Formal and Heuristic Approaches to Real-time Scheduling on Reconfigurable Systems

Thesis submitted to the
Indian Institute of Techenology Guwahati
for the award of the Degree

of

DOCTOR OF PHILOSOPHY

in

Computer Science and Engineering

Submitted by

Cherinet Kejela Addise

Under the guidance of

Dr. Arnab Sarkar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

January, 2022

Abstract

The dynamic partial reconfiguration (DPR) feature offered by modern FPGAs provides the flexibility of adapting the underlying hardware according to the needs of a particular situation during the runtime in response to application requirements. DPR has allowed the possibility of scheduling multiple real-time applications over both space and time so that the computation capacity of the FPGA floor may be efficiently harnessed. The scheduler generated/developed for the real-time tasks on FPGAs must not only handle all timing constraints, dependency constraints(if there is one), and FPGA based placement constraints but also correctly account for reconfiguration overheads involved in loading task bitstreams onto the configuration memory of the FPGA through the ICAP port. Hence, static off-line schedulers are often preferred for such a system in order to satisfy all these necessary constraints. In addition, off-line computation also allow exhaustive solution space enumeration to pre-compute optimal schedules at design time, thus ensuring lower design costs through higher resource utilization. *This thesis thus endeavors towards the exploration of new approaches and design of scheduling strategies for real-time tasks on partially reconfigurable platforms. Particularly, we present three static offline scheduler design approaches for reconfigurable systems: (i) a formal scheduler synthesis framework for the real-time tasks executing on an FPGA platform, using supervisory control of timed discrete event systems as the underlying formalism. (ii) an ILP based solution strategy for scheduling persistent real-time applications represented as precedence constrained task graphs on partially reconfigurable FPGAs and (iii) a heuristic solution methodology for scheduling persistent real-time applications represented as precedence constrained task graphs on partially reconfigurable FPGAs.*

Declaration

I certify that:

- The work contained in this thesis is original and has been done by me under the guidance of my supervisor.
- The work has not been submitted to any other Institute for any degree or diploma.
- I have followed the guidelines provided by the Institute in preparing the thesis.
- I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.
- Whenever I have used materials (data, theoretical analysis, gures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Cherinet Kejela Addise

Copyright

Signature of Author.....

Cherinet Kejela Addise

Certificate

This is to certify that this thesis entitled "***Formal and Heuristic Approaches to Real-time Scheduling on Reconfigurable Systems***", being submitted by **Cherinet Kejela Addise**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under my supervision and guidance. The thesis, in my opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of my knowledge, it has not been submitted elsewhere for the award of the degree.

.....

Dr.Arnab Sarkar

Associate Professor

Advanced Technology Development Centre

IIT Kharagpur

Dedicated to

Almighty GOD and all my respectd teachers

whose knowlwdge, blessing, love and inspiration paved my path of success

Acknowledgements

It is a genuine pleasure to express my deepest gratitude to my advisor Prof. Arnab Sarkar for the continued support of my Ph.D. study and research, for his patience, motivation, enthusiasm, and immense knowledge. You have inspired and motivated me during difficult times when I needed words of encouragement. I am also honored to have you as my guide and sincerely thank you for your understanding and thoughtfulness over the years. I would not be where I am today without your help and I will always have you to thank for it. I would also like to thank Prof. Chandan Karfa for his support, advice, punctuality, friendly approaches, and above all for being there for me when I needed him the most. Besides my advisors, I would like to thank the rest of my thesis committee members: Prof. H. K. Kapoor, Prof. John Jose, and Prof. Moumita Patra, for their insightful comments and encouragement.

My sincere thanks also go to Dr. Rajesh Devaraj and Dr. Sangeet Saha, without whom I would not have made it through my Ph.D. thesis. We had countless discussions and shared ideas that have changed my view of the research in many ways. I have learned so much from you about my field.

I would like to express my gratitude to the director, the deans, and other management of IIT Guwahati whose collective efforts have made this institute a place for world-class studies and education. I am thankful to all faculty and staff of Dept. of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work.

I would also like to give special thanks to the ministry of National Defence, Ethiopia, particularly Defence Engineering College for allowing me to pursue my Ph.D. research.

I am thankful to my friends Mohammed and Mahari for trusting in my capabilities and for your words of encouragement. I am so blessed to have a friend like you. I am also thankful to Dr. Abhijit, Dr. Sanjit, Dr. Piyoosh, Dr. Pranav, Partha Sarathi, and Palash for sharing their friendly advice during my stay in IITG.

Contents

List of Algorithms

List of Figures

List of Tables

List of Acronyms

ACC *Adaptive cruise control*

ALAP *As Late As Possible*

APP *Application*

ASAP *As Soon As Possible*

ASIC *Application Specific Integrated Circuit*

ASIP *Application Specific Integrated Products*

ATG *Activity Transition Graph*

BLC *bottom left corner*

BRAM *Block Random Access Memory*

CE *Cyclic Executive*

CLB *Configurable Logic Blocks*

COS *Complete Overlap Side*

CPU *Central Processing Unit*

CS *Classified Staffing*

CTI *Critical Task Indicating*

DAG *Directed Acyclic Graph*

DM *Deadline Monotonic*

DP *Deadline Partitioning*

DSP *Digital Signal Processing*

EDF *Earliest Deadline First*

FCFS *First Come First Serve*

FF *First Fit*

FPGA *Field Programmable Gate Arrays*

GPP *General Purpose Processor*

GPU *Graphical Processing Unit*

HB *Hard Blocks*

HDL *Hardware Description Language*

HNV *Half Normal Version*

HO *Horizontal Orientation*

ICAP *Internal Communication Access Port*

ILP *Integer Linear Programming*

ISE *Integrated System Environment*

JTAG *Joint Test Action Group*

KAMER *Keeping All MERs*

LLF *Least Laxity First*

LUT *Look Up Tables*

MAC *Multiply and Accumulate*

MER *Maximal Empty Rectangles*

MOP *Maximal Overlap Placer*

MUL *Multipliers*

MVDS *Multiple Variant Dag Scheduling*

NF *Next Fit*

NOR *Non Overlap Rectangle*

NAR *normalized achieved rewards*

NG *Negligible*

NP *Nonpolynomial*

NRCA *Non-pipelineRipple Carry Adder*

NV *Normal version*

PCAP *Processor Communication Access Port*

PF *Penalty Factor*

POS *Partial Overlap Side*

PRCA *Pipelined Ripple Carry Adder*

PRR *Partially Reconfigurable Region*

QoS *Quality of Service*

RC *Resource Constraints*

RCA *Ripple Carry Adder*

RM *Reconfigurable Module*

RPU *Reconfigurable Processing Units*

RT *Real-Time*

SC *Slot Constraints*

SCTDES *Supervisory Control of Timed Discrete Event Systems*

SHA *Shift-And-Add*

SJF *Shortest Job First*

SLR *schedule length ratio*

SMT *Satisfiability Modulo Theories*

SRPT *Shortest Remaining Processing Time*

SUR *Space Utilization Rate*

TDES *Timed Discrete Event Systems*

TNV *Twice Normal Version*

TRC *Top Right Corner*

VADS *variation aware dag scheduling*

VLS *Vertex List Set*

VO *Vertical Orientation*

WCET *Worst Case Execution Time*

List of Symbols

T_i	i^{th} task
I	$I = \{T_1, T_2, \ldots, T_n\}$ tasks
A_i	Arrival time of tasks T_i task
D_i	Relative deadline of tas T_i(with respect to its arrival)
P_i	Fixed (Minimum) inter-arrival time for periodic (sporadic) taskτ_i
$E_{i,j}$	Execution time of task T_i task
a_i	Arrival(event) of T_i
rs_i	Reserve slot/ tile k_i
$sd_{i,j}$	start of download of $T_{i,j}$
$cd_{i,j}$	completion of download of $T_{i,j}$
$se_{i,j}$	start of execution of $T_{i,j}$
$ce_{i,j}$	completion of execution of $T_{i,j}$
$su_{i,j}$	start of unload of $T_{i,j}$
$cu_{i,j}$	completion of unload of $T_{i,j}$
us_i	unreserve slot/tile k_i
Σ_{spe}	Set of perspective events
Σ_{rem}	Set of remote events
Σ_{con}	Set of Controllable events
Σ_{unc}	Set of Uncontrollable events
Σ_{for}	Set of forcible events
Σ_i	Set of events associated with task T_i
$tick$	Passage of one unit of the global clock
t	Short hand notation for $tick$
$Trun_i$	Execution time for task T_i

$Trun_i^{k_i}$	Execution time requirement for T_i's k_i^{th} version
$Trec_i$	Loading time for task T_i
$Trec_i^{k_i}$	Load time requirement for the k_i^{th} version of T_i
w_i	Width of task T_i
h_i	Height of task T_i
REW_i	Reward obtained for task T_i
D_{Dag}	Deadline of the task graph
ar_i^j	Area of T_i^{th} task j^{th} implementation
ζ_i	Number of versions of each task T_i
k_i	Task versions
Tl_i	Load start time of T_i
Te_i	Execution start time of T_i
REW_{sys}	Achived system-level reward
τ	set of topologically sorted tasks
U	Utilization
ER_j	An integer variable which holds the remaing execution requirement of T_j
ILR	an integer variable which holds the remaining time required to load the current task through ICAP

Chapter 1

Introduction

The increasing algorithmic complexity of embedded applications has led to a paradigm shift towards heterogeneous and highly integrated systems that include GPPs (General Purpose Processors), ASICs (Application Specific Integrated Circuits), ASIPs (Application Specific Instruction Set Processors), and even reconfigurable devices (such as FPGAs). Among these, FPGAs allow the advantage of adaptive high-performance application-specific hardware instantiation at lower costs. Many of the emerging commercial FPGAs also feature dynamic reconfiguration [3] so that hardware instantiations can be quickly reconfigured at runtime allowing even higher flexibility. Therefore, many of today's safety-critical real-time embedded systems including avionic and automotive systems, nuclear reactors, etc. [4], have begun to employ reconfigurable fabric within their architectures. FPGAs have also been used in synthetic vision, object tracking [5], cryptography [6], and digital audio/video [7] applications. For example, a given complex safety-critical system may employ FPGAs as a performance efficient reconfigurable backup platform for its real-time tasks. When one or more processors in the system fail, the FPGA may need to assume the responsibility of executing the real-time tasks that were previously running on the failed processors. However, efficiently executing dynamic hard real-time tasks on reconfigurable platforms require well-defined resource allocation and admission control mechanisms which not only guarantee the satisfaction of all timing constraints but also allow high resource utilization through the systematic management of time (effective scheduling) and space (effective mapping on the available 2D reconfigurable floor area). A reconfigurable resource basically consists of a 2-dimensional array of $W \times H$ Reconfigurable Processing Units (RPUs), commonly referred to as Configurable Logic Blocks (CLBs). A task T_i in such system is a relocatable digital circuit, typically rectangular in

shape, which may be configured to be executed consuming a sub-region $w_i \times h_i$ on the floor (having total area $W \times H$) of the reconfigurable device with the assumption that enough routing resources will be available to fulfill the needs of any placed task

Many embedded real-time applications are conveniently modeled as precedence constrained task graphs. A precedence constrained task graph $G = (T, E)$ is composed of a set T of task nodes and a set E of edges between task nodes. Each task node $T_i \in T$ represents a distinct functionality of an application. Each node $e_{ij} \in E$ denote precedence constraints between a distinct pair of task nodes T_i, T_j. As an example, let us consider the Adaptive cruise control (ACC) system in a modern car whose objective is to maintain a safe distance with other automobiles in the vicinity. ACC first determines the current distance and speed of an automobile in the neighborhood along with the speed at which it itself is running. Based on these parameters ACC determines the desired speed followed by the required braking force and throttle position (relative to current throttle position). Finally, it generates actuation outputs to actual physical controls over throttles and brakes. Ofcourse, this entire set of activities must be performed within stringent timing constraints which is modeled through an overall deadline for the completion of all activities of the application graph.

In literature, it has been shown that there are a few research works on dependent task scheduling on FPGAs [8, 9, 10]. However, these works did not consider one or more of the following; (i) Multiple alternative functional variants for hardware tasks (ii) Reconfiguration overheads, or (iii) Real-time constraints. The work in [11, 12] show that the unoccupied programmable resources on the 2D homogeneous DPR device can be represented by a maximal empty rectangle (MER) list. Since MER is a rectangle that cannot be completely covered by other rectangles, if there are enough resources, the targeted task can locate an open position by iterating through the MER list. However, such MER-based placement strategies lead to uncontrolled internal fragmentation. Therefore, a task allocation algorithm and resource management strategy for multi-variant tasks on 2D partially reconfigurable FPGAs must be proposed.

FPGA-based platforms are increasingly being looked upon as a lucrative and cheap alternative for executing many of the real-time safety-critical applications. Given an application, effectively organizing the executions of the application tasks while satisfying all timing constraints, is ultimately a scheduling problem. A majority of the existing scheduling techniques are online and employ heuristic sufficiency based schedulability conditions in order to provide scheduling decisions within reasonable time. In order to

control involved computation overheads, the online scheduler cannot include the consideration of all necessary conditions in the scheduler design and this makes online approaches inherently sub-optimal in nature. However, optimal solutions can make a fundamental difference towards enhancing the efficiency of resource-constrained safety-critical cyber physical systems (CPSs) in terms of reliability, performance, space, cost etc. Formal model-based design techniques have been found to be a lucrative alternative for many safety-critical CPSs including designs for reconfigurable platforms. This is because, formal synthesis mechanisms are *correct-by-construction*, which are typically more suitable compared to adhoc allocation techniques, especially for safety-critical CPSs [13].

While scheduling a set of real-time tasks on partially reconfigurable systems, dynamic reconfiguration of the FPGA requires to take decisions about the choice of new configurations and this may depend on factors such as: (i) The sequence of events during a particular run (ii) Decisions on the relative execution order of a set of functionalities over time (iii) Predictive knowledge about the behavior of the functionalities (iv) Placement decisions corresponding the functionalities on the available reconfigurable real-estate (v) Reconfiguration overheads etc. Efficient scheduling decisions related to the selection of new configurations is therefore a very complex design issue because of the sheer exponential nature of the combinatorics of possible choices, and is therefore difficult to accomplish online. Hence, Off-line formal approaches are often preferred in the design of reconfiguration controllers (i.e., *scheduler*) that are *correct-by-construction* as well as optimal in terms of usage of resource.

When FPGAs become the target platforms for the execution of a task, we can appreciate the fact that a task (node) could have different versions based upon their implementation techniques. For example, let us assume that a task contains a certain conditional *"loop"* within its computational steps and one can adopt two distinct implementation strategies to carry out the execution. Firstly, unroll the "loop" partially and execute it by exposing spatial parallelism. Secondly, execute the task in its compact form without unrolling the "loop". It is worth mentioning that the first implementation will definitely consume less amount of time than the second implementation, but it is obvious that the first implementation would be more spatially expensive. However, from Amdahl's law, it is also obvious that this *"space vs throughput"* relation of a hardware circuit (task) cannot vary proportionally. From the above discussion, we can conclude that a hardware task could have multiple hardware variants (variation in throughput) based upon its spatial requirements. *In this thesis work, we consider the multiple hardware variants of tasks.*

1.1 Challenges

A scheduling mechanism which efficiently caters to diverse applications and serves the variety of processing platforms in today's safety-critical systems, must meet several challenges. We now enumerate a few such important challenges.

- A task T_i in a reconfigurable system is a relocatable digital circuit, logically represented through a corresponding bitstream (*.bit file*). This bitstream typically consumes a rectangular subregion of dimension $w_i \times h_i$ and may be configured to be executed anywhere on the floor (having total area $W \times H$) of the reconfigurable device. Parallel execution (spatial) of a given subset of tasks $T_p = \{T_{p1}, T_{p2}, ...T_{p|T_p|}\}$ is achieved through their simultaneous placement on the FPGA floor such that no task subregion overlaps with the device boundaries or with other subregions. The vacant region within a set of already placed tasks whose area is not large enough to allow the feasible placement of any other task, is considered to be wasted due to fragmentation. One of the principal goals of any placement strategy is to reduce the total unutilized area lost due to fragmentation. However, the generic problem which attempts to either minimize the total area consumed by a given set of tasks, or maximize the number of tasks that may be feasibly accommodated within a given floor area, is known to be NP-hard [14].

- The floor of the FPGA may contain hard embedded blocks within the uniform fabric of CLBs. These heterogeneous hard components may include pre-fabricated blocks of BRAMs, MULs, etc. Due to their efficient implementation, the use of these inbuilt hard blocks (HBs) may often help improve performance. However, usage of these HBs restricts the placement of a task to those specific subregions / positions where the HBs are located and makes the placement techniques more challenging.

- Providing high resource utilization is typically not possible only by harnessing spatial parallelism on the FPGA floor. Rather, this demands scheduling strategies that also allow proportional fairness in the rates of progress for all co-scheduled tasks, similar to optimal online general purpose multiprocessor scheduling. However, such stipulated rates of progress for all tasks may only be maintained by appropriately multiplexing task executions over time using a preemptive scheduling policy (similar to DP-Fair) that incurs a significant number of context switches.

The design of a seamless context switch/preemption mechanism is another big challenge imposed by an FPGA's architecture. Task contexts are stored in state-holding elements like Flip-flops and LUT-RAMs of CLBs and other Hard Blocks (BRAMs, MULs, etc.). Switching context in any specific region of the FPGA involves: i) capturing the contexts of tasks that were executing in the region prior to a switch, ii) updating the contexts of these captured tasks and saving them in external memory, iii) forming a new bitstream comprising of tasks that should execute in the region subsequent to the switch, iv) restoring the new bitstream in the region to re-initiate execution after preemption. Literature [15, 16, 17, 18, 19, 20] has shown that significant improvements in both context capture/extraction and context updation times may be obtained through a selective bitstream read-back and context manipulation mechanism. These improvements have been able to effect a drastic reduction in overall context switch overheads from more than 10ms to hundreds of microseconds to a few milliseconds. Such low overheads have now made preemptive scheduling more affordable in partially reconfigurable systems.

- Dynamic partial reconfiguration (DPR) is a key differentiating capability associated with field programmable gate arrays (FPGAs). While DPR has been studied extensively in academic literature, it finds limited use in deployed systems [21]. Among other issues, most design tools available today are incapable of allowing full DPR features for an application. Though some FPGA vendor tool suppliers incorporate basic DPR functionality (i.e., dynamic configuration swapping),these tools are still limited as they do not typically allow dynamic merging of neighboring regions for the flexible 2D area model. The lack of advanced tools still restricts us from harnessing the full power of DPR.

1.2 Research Objectives

The principle aim of this dissertation has been to investigate the theoretical and practical aspects of offline scheduling strategies for reconfigurable systems while keeping in view the challenges/hurdles discussed in the previous section. In particular, the objectives of this work may be summarized as follows:

1. Design and implement a formal scheduler synthesis framework that generates an optimal schedule for a set of non-preemptive periodic real-time tasks executing on

a FPGA platform, using supervisory control of timed discrete event systems as the underlying formalism.

2. Design and implement an ILP-based solution strategy for scheduling persistent real-time applications represented as a precedence-constrained task graph on partially reconfigurable FPGAs.

3. Design and implement novel heuristic algorithms for scheduling persistent real-time applications represented as a precedence-constrained task graph on partially reconfigurable FPGAs.

1.3 Summary of work done

This research work conducts three offline scheduling approaches for reconfigurable systems. These approaches are summarized in three contributions as discussed below.

1.3.1 Contribution 1

The contributions of this work can be summarized as:

- Development of DES models for individual system components (i.e., non-preemptive tasks, reconfiguration port) and system specifications (i.e., resource constraint, timing constraint), to synthesize an optimal scheduler for the scheduling of a set of real-time non-preemptive periodic tasks on dynamically reconfigurable FPGAs.

- Demonstration of the practical viability of the proposed scheme through a proof-of-concept implementation. This has enabled the validation of the correctness of the proposed scheduling strategy on real platforms.

Now, We briefly discuss the scheduler synthesis framework of our proposal and then the implementation decription followed.

1. Formal approaches to real-time scheduling for reconfigurable systems.
   The pictorial representation of the proposed framework for the formal approaches is presented in figure 1.1. The proposed framework receives information regarding the task set, processing platform, constraints, and modeling style, as input parameters. A task set is a set of n independent real-time periodic applications/tasks:

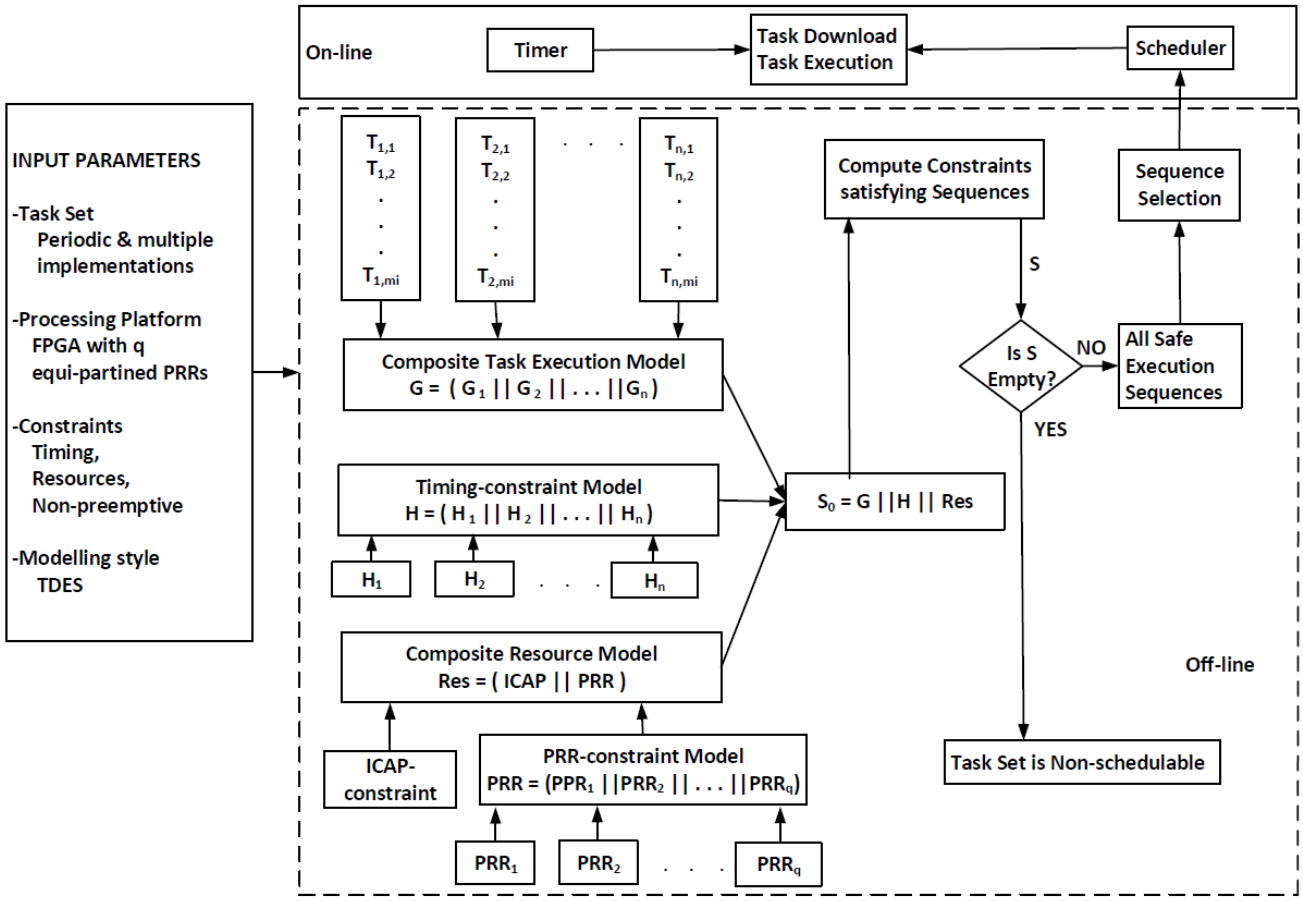


Fig. 1.1: Pictorial representation of the proposed scheduler synthesis framework

$T = \{T_1, T_2, \ldots, T_n\}$ that needs to be scheduled on a set of q equi-sized Partial Reconfigurable Region (PRR),$\{k_1, k_2, \ldots, k_q\}$. These tasks are nonpreemptive in nature, i.e., once the execution of a task is started on its assigned PRR, the PRR cannot be reconfigured until the completion of the task. In a reconfigurable system, a task or application is represented as a relocatable digital circuit, logically represented through a corresponding bitstream (also referred to as Reconfigurable Module (RM))) and are stored and maintained in a repository residing in memory. We assume that each task T_i can have m_i distinct implementations: $\{T_{i,1}, T_{i,2}, \ldots, T_{i,m_i}\}$. The j^{th} implementation of T_i is characterized by, (i) spatial requirement in terms of the required PRRs, (ii) temporal requirement in terms of the worst-case Execution Time (WCET) and the associated reconfiguration loading times. The reconfigurations of FPGA are performed under the supervision of our developed reconfiguration controller module by loading bitstreams from the repository (SPI flash) into the configuration memory of the FPGA through its ICAP port.

Given the task set, a set of executional PRRs and constraints, our framework first constructs the Timed Discrete Event Systems (TDES) models for each task, Partial Reconfigurable Regions (PRRs), Reconfigurable Port (ICAP), and the timing constraint. Given n individual TDES models $G_1, G_2, \ldots, G_n$, corresponding to $T_1, T_2, \ldots, T_n$ tasks in the system, a synchronous product [22] denoted by composition $G = (G_1 \parallel G_2 \parallel \ldots, G_n)$ on the models gives us the composite model representing the concurrent execution of all tasks. Similarly, the composite resource model(i.e. $Res = ICAP \parallel PRR$) and timing specification model ($H = H_1 \parallel H_2 \parallel, \ldots H_n$) can be obtained from individual models that captures the constraints such as resource and timing. The composite system model is obtained from individual models to find all the sequences that satisfy the constraints.

That is, $S_0 = G \parallel H \parallel Res$. The model S_0 may consist of deadlock states in it (which will block the execution of system) and hence, to obtain a non-blocking sub-part of S_0, we apply supervisor synthesis algorithm [23]. The resulting model S contains all feasible scheduling sequences that satisfy the given constraints. It may happen that the state set of S to be empty, which implies that the given task set is non-schedulable under the given set of constraints.

2. Supervisor Implementation

   The models developed in the above sections are realized on the FPGA using the

partial reconfiguration design tools. The tools must allow the implementation of scalable reconfigurable systems with various partial modules loaded to different locations of the device at runtime. The partial reconfiguration tools comprise several complex tasks, including FloorPlanning, communication architecture synthesis, physical constraints generation, physical implementation, timing verification, and the final bitstream generation [24, 25].

The synthesized supervisor was developed on the flexible 2D area model. Realizing the flexibilities of the 2D area model was the major drawback of the vendor tool manufacturers, like Xilinx ISE/Vivado or Altera Quartes. Even though there are academic tools (e.g. GoAhead [26] or OpenPR [27]) that can support the flexible 2D area model, they require detailed technical expertise about the FPGA fabrics. In our supervisor implementation, we used the academic tool, GoAhead for creating the partial regions and generation of the communication architectures. Additionally, the ISE design suit and the VHDL language, were used for writing the code of the various functional units of the supervisor (i.e., Reconfigurable controller, UART module, Tick_generator, etc.). We used the Atlys development board to test and verify the working of the scheduler on selected hard task examples.

1.3.2 Contribution 2

In this work, we developed an ILP-based solution strategy for scheduling persistent real-time applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs. The generated schedule must ensure that the execution of nodes in the task graph is completed within the given deadline while satisfying all dependency and resource-related constraints. While scheduling dependent tasks with varied implementations on an FPGA, multiple constraints must be satisfied simultaneously. That is,

1. ***Unique Load Time Constraint:*** During a reconfiguration, exactly one version of each task must start loading through the ICAP on the FPGA floor at a unique time step.

2. ***Single Load Channel (ICAP) Constraint:*** Only one task can be loaded through the single available ICAP port at a given time.

3. ***Load-execution Dependency Constraint:*** A task T_i can commence its execution only after its loading finishes.

4. ***Execution execution Dependency Constraint:*** Corresponding to each directed edge ($T_i, T_j \in E$) in the DAG, the execution of task T_j must commence only after the completion of its predecessor, T_i.

5. ***Placement Constraints:*** For a given temporal schedule of the tasks along with their selected versions, the placement constraints attempt to ensure that the tasks having overlapping life times on the FPGA floor do not spatially overlap with each other at any instant over the schedule length. Additionally, these constraints also guarantee that the tasks do not overlap with the FPGA boundaries.

6. ***Deadline Constraint:*** In order to ensure that the application G meets its end-to-end absolute deadline D_G, the sink node $T_{|T|}$ must complete execution by D_G.

The ILP solutions have been generated using the IBM CPLEX tool in OPL format. The simulation was performed on Intel(R) Core(TM) i5-1035G1 CPU @1.00GHZ 1.19GHz and 8GB installed memory(RAM). The Normalized Achieved Reward (NAR), the utilization (U), and cumulative schedules execution time($Sched\_exe\_Time$) are the metrics used to evaluate the performance of the proposed ILP formulation. We varied the application load (APP_{load}) and the number of tasks on the x-axis. The test result confirmed that the normalized reward remains comparable with the changes in APP_{load}. The utilization and the cumulative scheduler execution time were shown to increase as the number of tasks increased.

1.3.3 Contribution 3

The main objective of contribution 3 is to design and implement a Spatio-temporal schedule that maximizes the aggregate rewards through the judicious selection of task versions for a given runtime partial FPGA platform. The generated schedule must ensure that the execution of nodes in the task graph is completed within the given deadline while satisfying all dependency and resource-related constraints. The pictorial representation of the proposed system model is shown in Figure 1.2. It represents the Spatio-temporal scheduler for a given application. The system model is classified into three functional units as described below.

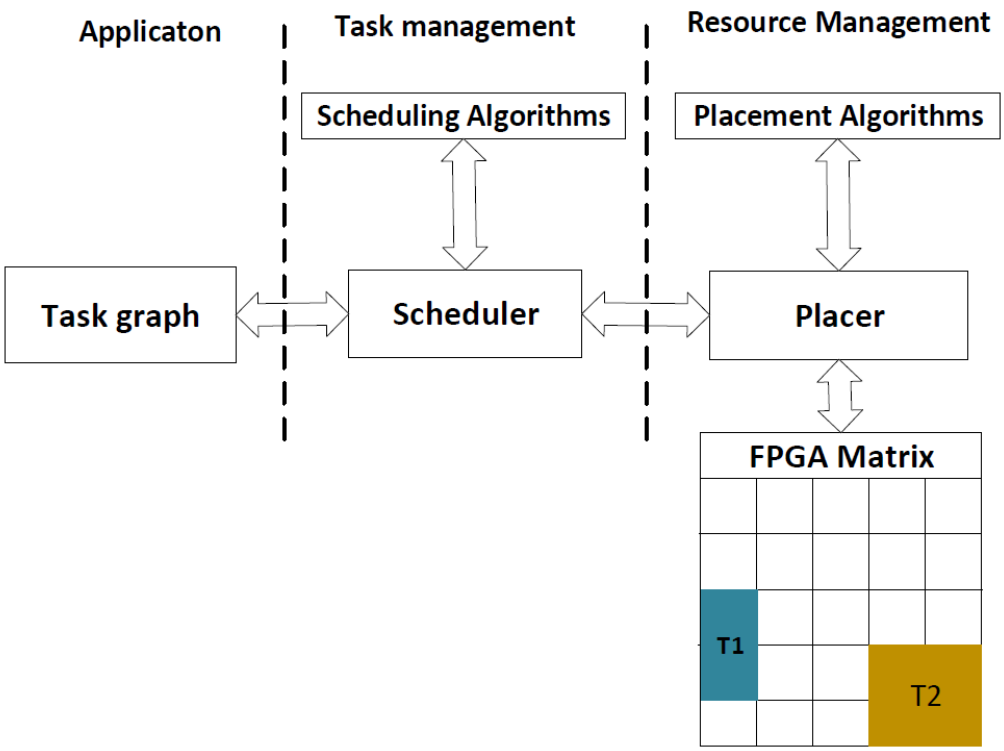


Fig. 1.2: System Model

1. **Application**

   We model a real-time application as a precedence constrained Directed Acyclic Graph (DAG) $G = (T, E)$, where T is a set of hardware tasks ($T = \{T_i \mid 1 \leq i \leq |T|\}$) and E is a set of directed edges ($E = \{\langle T_i, T_j \rangle \mid 1 \leq i, j \leq |T|; i \neq j\}$) representing precedence relations between distinct pairs of tasks. An edge $\langle T_i, T_j \rangle$ refers to the fact that task T_j can begin execution only after the completion of T_i. It is further assumed that a hardware task T_i may have k_i different versions / implementations; that is, $T_i = \{T_i^1, T_i^2, \ldots, T_i^{k_i}\}$. Although, all versions of a task produce the same output, their execution times, area requirements and accuracy of results may vary. Different versions of a task essentially mean different hardware circuit implementations corresponding to the same functionality [28].

2. **Scheduler**

   The scheduler manages the task execution sequences. Efficient task scheduling algorithms consider task dependencies and resource utilization to exploit the dynamically reconfigurable systems.

3. **Placer**

   For a specific version of each task as selected by the scheduler, a deadline meeting temporal schedule might be possible. However, this temporal schedule can be deemed to be feasible only if it is spatially schedulable on the FPGA floor. The placement strategy attempts to generate such a feasible spatial schedule. Specifically, for a subset of tasks (say, τ) having overlapping times in the temporal schedule, the placement strategy attempts a sub-region of size $w_i \times h_i$ for each task $T_i \in \tau$ such that no sub-region overlaps with the device boundaries or with other sub-regions. The spatial schedule consists of placed tasks along with vacant regions. A vacant region within a set of already placed tasks whose area is not large enough to allow the feasible placement of any other task is considered to be wasted due to fragmentation. One of the principal goals of any placement strategy is to minimize the total unutilized area lost due to fragmentation.

4. We performed a simulation-based experiment to test and verify our heuristic algorithm. The algorithms are implemented in C language and the data set generation through Matlab. The simulation was performed on Intel(R) Core(TM) i5-1035G1 CPU @1.00GHZ 1.19GHz and 8GB installed memory(RAM). We evaluated the performance of our proposed heuristic algorithms using the following metrics; The Normalized Achieved Reward (NAR), Schedule Length Ratio (SLR), Utilization ratio (U), and cumulative schedules execution time($Sched\_exe\_time$). We varied the application load (APP_{load}) against the NAR and SLR. The test result shows that the normalized reward remains comparable with the changes in APP_{load} and the SLR increases with increasing APP_{load}. For the utilization, we varied the number of tasks to be placed on the FPGA and we found out that utilization increases as the number of tasks increases. We also tested the effect of varying the reconfiguration time factor with respect to NAR and SLR. The result shows that NAR decreases, whereas the SLR increases as the reconfiguration time factor increases.

1.4 Organization of the Thesis

The thesis is organized into six chapters. A summary of the contents of each chapter is as follows:

Chapter 2: *Background on Real-time Systems, Supervisory control and Spatio-temporal Scheduling on FPGAs*
This chapter presents background on real-time systems and supervisory control of timed discrete event systems. Then, we discuss the Spatio-temporal scheduling of FPGAs.

Chapter 3: *A Supervisory Control Approach for Scheduling Real-time Periodic Tasks on FPGAs*
In the third chapter, we present a formal scheduler synthesis framework for a set of non-preemptive periodic real-time tasks executing on a FPGA platform. First, task execution on reconfigurable FPGA platforms is discussed. Then, the scheduler synthesis framework using supervisory Control of DES for tasks executing on FPGAs is presented. The chapter also discusses the detailed implementations of the synthesized supervisor (scheduler) practically on FPGAs.

Chapter 4: *An ILP-based Approach to Real-Time Scheduling of Task Graphs on Partially Reconfigurable FPGAs*
In this chapter, we present the design and implementation of an ILP-based Spatio-temporal schedule for a precedence task graph on partially reconfigurable FPGAs. First, we discuss the formalization of the precedence-constrained Spatio-temporal Scheduling Problem. Then, we detail an ILP-based solution to the DAG scheduling problem.

Chapter 5: *Heuristic Approach to Real-Time Scheduling of Task Graphs on Partially Reconfigurable FPGAs*
Research conducted in the fifth chapter deals with the heuristic approaches of a Spatio-temporal schedule for a precedence task graph on partially reconfigurable FPGAs. First, the discussion of the scheduler algorithms is presented, followed by the placer heuristics.

Chapter 6: *Conclusion and Future Work*
The thesis concludes with this chapter. We discuss the possible extensions and future works that can be done in this area.

Chapter 2

Background: Real-time Systems, Supervisory Control, Spatio-Tempral Scheduling on FPGAs

This dissertation is oriented towards the formal and heuristic approaches to real-time scheduling on reconfigurable systems. The previous chapter provided an overview of the complexity of modern embedded systems and discussed how FPGAs may be adapted to flexibly co-execute multiple performance-critical real-time functionalities on such systems. In this chapter, first, we present a background on real-time systems and scheduling with their different flavors and trends. Then, the chapter discusses the supervisory control of timed discrete event systems. Subsequently, we discuss the evolution of reconfigurable platforms with a particular emphasis on reconfiguration techniques. Next, a discussion on the evolution of FPGA-based architectures emphasizing mechanisms used in application loading and mapping, reconfiguration techniques, use of heterogeneous components (hard blocks), supporting computing infrastructures, etc. The chapter concludes by presenting a review of various formal and heuristic real-time scheduling strategies for FPGAs.

2.1 Real-Time Systems

A real-time systems are characterized by the necessity to satisfy two notions of correctness, functional and temporal. Therefore, such systems must not only produce correct results, the results should be produced before a stipulated time bound called deadline. Real-time systems span a wide range of domains including industrial control systems, automotive

and aviation systems, multimedia systems, consumer electronics, telecommunications, etc. A typical example of a real-time system is provided by a temperature controller in a chemical plant that is required to switch off the heater within 30 milliseconds when the temperature reaches 250, to avoid an explosion.

2.1.1 Hard vs Soft Real-Time

The hardness of a real-time systems is determined by the criticality of missed deadlines. Missing a deadline in a hard real-time system may lead to catastrophic consequences. Some examples of hard real-time systems are: fly-by-wire controllers for airplanes, monitoring systems for nuclear reactors, car navigation, robotics etc. On the other hand, a soft real-time system is less restrictive; it tolerates deadline misses at the cost of the quality of results, as long as they remain within certain temporal limits beyond which the system becomes useless. Obviously, Quality of Service (QoS) degrades as delay in response increases beyond deadline. Examples of such systems include streaming video, voice over IP, interactive gaming etc.

2.1.2 A Real-Time Tasks Model

A task is composed of a set of instructions to be executed on a processor. Typically task executes repeatedly and each such execution instance is referred to as a job. Important parameters which characterize a real-time task are:

1. **Arrival time** a_i is the time at which a task becomes ready for execution. It is also referred as *release time* or *request time* and indicated by r_i.

2. **Start time** s_i is the time at which a task starts its execution.

3. **Computation time** or *Execution time* C_i is the time necessary to the processor for executing the task without interruption.

4. **Finishing time** f_i is the time at which a task finishes its execution.

5. **Deadline** is the time before which a task should be completed. If its measured w.r.t. system start time (at 0), it will be called as *absolute deadline*(d_i) . If its measured w.r.t. request time (r_i), it will called as *relative deadline*(D_i).

6. **Response time** R_i is the difference between the finishing time and the request time: $R_i = f_i - r_i$.

7. **Worst-case execution time** e_i is the largest computation time of a task among all its possible execution.

8. **Lateness** L_i is the delay of a task completion with respect to its deadline: $L_i = f_i - d_i$.

9. **Tradiness** or *Exceeding time* E_i is the time a task stays active after its deadline: $E_i = max(0, L_i)$.

10. **Laxity** or *Slack time* X_i is the maximum time a task can be delayed on its activation to complete within its deadline: $X_i = D_i - C_i$.

11. **Priority** P_i is the importance given to a task in context of the schedule at hand.

A real-time task τ_i can be classified as periodic, aperiodic and sporadic based on regularity of its activation.

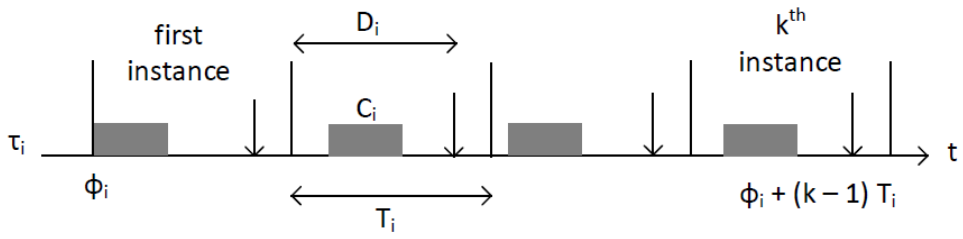


Fig. 2.1: Periodic task τ_i

1. **Periodic** tasks consist of an infinite sequence of identical activities, called instances or *jobs*, that are regularly activated at a constant rate. The activation time of the first periodic instance is called *phase* (ϕ_i). The activation time of the k^{th} instance is given by $\phi_i + (k-1)T_i$, where T_i is the activation period of the task.

2. **Aperiodic** tasks also consist of an infinite sequence of identical jobs. However, their activations are not regularly interleaved.

3. **Sporadic** tasks consist of an infinite sequence of identical jobs with consecutive jobs are seperated by a minimum inter-arrival time.

There are three levels of constraint on task deadline:

1. *Implicit Deadline:* all task deadlines are equal to their periods ($D_i = T_i$).
2. *Constrained Deadline:* all task deadlines are less than or equal to their periods ($D_i \leq T_i$).
3. *Arbitrary Deadline:* all task deadlines may be less than, equal to, or greater than their periods.

Processor Utilization Factor U: Given a set of tasks, $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, U is the fraction of the processor time spent in the execution of the task set. $U = \sum_{i=1}^{n} C_i/T_i$

Hyperperiod: It is the minimum interval of time after which the schedule itself repeats itself. If H is the length of such an inteval, then the schedule in $[0, H]$ is the same as that in $[kK, (k+1)K]$ for any integer $k > 0$. For a set of tasks activated simultaneously at $t = 0$, the hyperperiod is given by the least common multiple of the periods: $H = lcm(\tau_1, ..., \tau_n)$.

2.1.3 A real-time Scheduler

Scheduling appears in any domain where there is a need to allocate limited available resources in order to serve a certain number of tasks. It is then necessary to coordinate the use of such resources so that the tasks may run to completion as efficiently as possible. This efficiency means optimizing one or many criteria. Such criteria could be to minimize the schedule length (makespan), maximize resources utilization, minimize the number of tasks that must be rejected due to insufficient resources etc. The problem of scheduling can be described by a triplet $\{\alpha, \beta, \gamma\}$ where α represents the set of available resources, β the set of applications to be executed on the resources along with their time constraints and γ the objective function to be optimized. The generic multi-resource scheduling problem has been shown to be NP complete and hence, many scheduling heuristics of lesser complexity have been proposed.

The set of rules that, at any time, determines the order in which tasks are exeucted is called a *scheduling algorithm*. Given a set of tasks, $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion. A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints. A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule. Scheduling algorithms can be classified based on preemption.

1. **Preemptive**: tasks can be interrupted at any time (so that the processor may be assigned to another task) and resumed later.

2. **Non-preemptive**: once activated, a task must be continuously executed until completion.

In addition to the above classification, depending on whether the schedule is generated statically at design-time or dynamically at run-time, scheduling algorithms are differentiated as *off-line* or *online*. In offline scheduling, the scheduler has a priori knowledge of the task set and its constraints, such as arrival times, execution times, precedence constraints, etc. The schedule is generated and stored at design time and dispatched later during runtime of the system. Offline scheduling is also referred to as static. Offline scheduling is usually performed to find the optimal solution of tasks. Contrary to offline scheduling, the execution sequence is not known in advance for online scheduling. Online scheduling algorithms make their scheduling decisions at runtime based on the information about the tasks that have arrived so far. Online scheduling happens to be more flexible than offline scheduling since it can be used for the cases where the sequence of tasks dynamically changes at run-time. However, they may incur significant overheads because of runtime processing. Usually, online scheduling algorithms try to produce an "approximate" solution, but cannot guarantee the optimal solution. A scheduling algorithm is said to be *optimal* if it is able to find a feasible schedule, if one exits. An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A *heuristic* algorithm tends toward the optimal schdule, but does not guarantee finding it.

2.2 Real-time Scheduling for Uniprocessor Systems

Scheduling theory has been intensively studied over the years and uniprocessor scheduling takes the lion's share in the rich literature review. Many optimal uniprocessor scheduling algorithms have been proposed along with their schedulability analysis. Here below are some scheduling algorithms.

2.2.1 Rate Monotonic (RM)

The RM algorithm in [29] is a preemptive static priority scheduling scheme for periodic and independent tasks systems. In the static priority scheme, tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made

ready to run, the active task with the highest priority commences or resumes execution, preempting the currently executing task, if need be. In RM, shorter the period of a task, higher becomes its priority. Liu and Layland (1973) [29] have proven RM to be an optimal static priority scheduler for preemptive task systems.

An important shortcoming of the RM algorithm (as shown by Liu and Layland) is that even on uniprocessor systems no more than 69% of the processor may be utilized to ensure scheduling feasibility of a set of tasks under rate-monotonic priority assignment, in the worst case.

2.2.2 Deadline Monotonic (DM)

DM [30] is a static priority scheduling algorithm that gives the highest priority to the task with the least relative deadline d_i. DM could be used with periodic, aperiodic and sporadic tasks systems.

2.2.3 Earliest Deadline First (EDF)

EDF [29] is a dynamic priority scheduling scheme where the highest priority is assigned to the task with the closest absolute deadline. Priorities are reassessed and updated at runtime if necessary (e.g. on each task arrival). EDF has been proven to be optimal for preemptive periodic tasks. Later, EDF has also been shown to be optimal in the case of non-periodic tasks. EDF scheduling outperforms RM and produces less preemption compared to RM.

2.3 Real-time Scheduling for Multiprocessor Systems

An increasing number of real-time systems require more than one processors to achieve their performance goals. The problem of scheduling tasks on multiple processors cannot be seen as a simple extension of the uniprocessor scheduling due to additional constraints such as: task migration overheads, inter-task communication overheads etc. Traditionally, there are two classes of scheduling algorithms for multiprocessor platforms: partitioned scheduling and global scheduling.

2.3.1 Global Vs. Partitioning Scheduling

In global scheduling, all ready tasks are stored in a single priority queue among which scheduler selects the highest priority task at each invocation irrespective of which processor is being scheduled. In a purely partitioned approach on the other hand, the set of tasks is partitioned into as many disjoint subsets as there are processors available, and each such subset is assigned to a distinct processor [31]. After this mapping is obtained, all instances / jobs of a given task are executed only on the processor to which it is associated.

Due to the allowance of task migrations, global scheduling methodologies are typically able to achieve higher schedulability compared to partitioned scheduling. However, task migrations and preemptions come at the cost of increased runtime overheads. Therefore attempts have been made to devise schemes which are able to restrict migrations while satisfying the performance goals.

Pros and Cons:

The main advantage of partitioning is that it allows the multiprocessor scheduling problem to be reduced to a set of uniprocessor ones. Within each processor, a separate well known uniprocessor scheduler like Rate Monotonic (RM), Earliest Deadline First (EDF), etc. may be easily applied. In addition, the overhead of inter-processor task migrations is smaller than global scheduling. Finally, because task-to-processor mapping (which task to schedule on which processor) need not be decided globally at each scheduling event, the scheduling overhead associated with a partitioned strategy is lower than that associated with a global strategy [[32], [31]]. However, partition based scheduling approaches may often be plagued by low resource utilization. Oh et.al. in [33] showed that on homogeneous multiprocessor systems where no task migration between processors is allowed and each processor schedules tasks preemptively employing the Rate Monotonic policy, the maximum utilization that may be achieved is just 41%. When **EDF**, a well known optimal scheduler for uniprocessor systems, is applied to multiprocessors using a fully partitioned approach disallowing task migrations, the worst case utilization bound reduces to 50% [34].

On the other hand, even though the generic global scheduling methodology may have higher scheduling complexity and cause an unrestricted number of migrations and cache misses, it possesses many attractive features like flexible resource management, dynamic load distribution, fault resilience, high system utilization, etc. [35].

2.3.2 Recent Trends in Real-time Multiprocessor Scheduling

Recent multiprocessor scheduling techniques for general purpose processors such as ER-fair [36] and DP-Fair [37] have shown that it is possible to achieve optimal resource utilization irrespective of the skewness in task weights / periods, by maintaining proportional fair execution progress for all tasks. However, such proportional fairness is achieved at the cost of higher preemptions / migrations.

ERfair Scheduler: ERfair schedulers mandate execution of each task T_i to proceed proportionally at a rate lower bounded by a parameter called its weight (wt_i) which is defined as the ratio of its execution requirement (e_i) and period (p_i). To maintain ER-fairness at the end of any given scheduling quanta or time slot t, $s_i < t < s_i + p_i$, at least $\frac{e_i}{p_i} \times (t - s_i)$ of the total execution requirement of e_i must be completed for each task T_i, where s_i is the start time of task T_i. ERfair ensures schedulability if the summation of weights of the n tasks is atmost the number of processors $\sum_{i=1}^{n} e_i/p_i \leq m$.

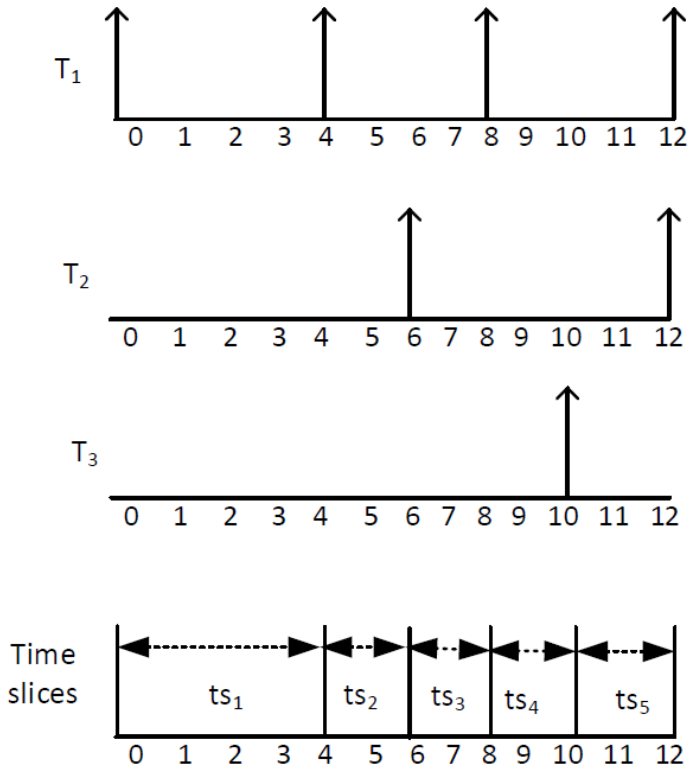


Fig. 2.2: Time Slices in DP-Fair

DP-Fair Scheduler: Although ERfair is an optimal scheduler allowing full resource utilization, it suffers from high scheduling overheads as well as inter-processor task migration overheads. Recently Deadline Partitioning-Fair (DP-Fair), a lower overhead optimal scheduling strategy with a more relaxed proportional fairness constraint has been proposed. DP-Fair sub-divides time into slices demarcated by the deadlines of all tasks. Time slice ts_i denotes the interval between the $(i1)^{th}$ and ith task deadlines. For example, as shown in Figure 2.2, let us assume three ready periodic tasks T_1, T_2, T_3 at time 0 with periods / deadlines 4, 6, 10, respectively. Therefore, there are distinct deadlines at 4, 6, 8, 10 and 12. Hence, we have five distinct time slices ts_1, ts_2, ts_3 ts_4 and ts_5.

Within each time slice, each task is allocated a "per-slice workload" which is equal to its weight times the length of the time slice. If tsl_i be the length of the i^{th} time slice, then task T_i should complete its allocated workload $(\frac{e_i}{p_i} \times tsl_i)$ within the time slice. Although the exact intra-time slice task scheduling policy may vary, any adopted strategy must obey the following three rules:

1. Always run a task with zero local laxity
2. Never run a task with no workload remaining in the slice
3. Do not voluntarily allow more idle processor time than $(m - \sum \frac{e_i}{p_i}) \times$ (length of time slice)

DP-Fair is optimal multiprocessor algorithm. In comparison to ERfair, DP-Fair enforces a more relaxed rate constraint and preserves scheduling optimality by guaranteeing ER-fairness constraints to be satisfied only at task period / deadline boundaries. Obviously, for such a criterion to be guaranteed in a system of m processors, we must have $\sum_{i=1}^{n} e_i/p_i \leq m$.

2.3.3 Combined Scheduling of Periodic and Aperiodic Tasks

The generic problem of the combined scheduling of periodic and aperiodic tasks has spun-off in different directions primarily based on the types of real-time tasks that a system requires to handle. For example, based on task arrival times, scheduling algorithms have been designed considering either, both periodic and aperiodic tasks to be arbitrarily arriving at runtime [38], both task types to be statically known offline [39], or considering periodic tasks to be persistent and aperiodic tasks to be dynamically arriving [40]. On the

basis of task deadlines on the other hand, scheduling strategies have been considered for scenarios where both periodic and aperiodic tasks are hard [41], and also where periodic tasks are hard and aperiodic tasks are soft [42]. In the former case, sound acceptance tests must be employed to guarantee that all accepted tasks meet their deadlines. In the later case, the main objective is to guarantee all deadlines for periodic tasks while achieving good average response times for the soft aperiodic tasks.

For task sets containing persistent periodic tasks along with dynamic aperiodic tasks, typically, an initial periodic task schedule is created offline and then aperiodic tasks are scheduled within the remaining capacity. A hard aperiodic task acceptance strategy, namely Critical Task Indicating (CTI) algorithm, has been discussed by Lee et al. in [43]. CTI generates an offline scheduling table to keep an account of residual resources within a single hyper-period after assigning the periodic tasks in an As Late As Possible (ALAP) fashion. Later, this table is used by the online scheduler to find slacks where hard aperiodic tasks may be accommodated. Fohler et al. discussed a similar approach called Slot-Shifting, in [44]. An extension of the Slot-Shifting algorithm was considered by Schorr et al. in [45], in order to incorporate non-preemptive aperiodic tasks into an arbitrary feasible schedule.

On multiprocessor systems, the offline resource allocation step must not only consider scheduling but also mapping of the static periodic tasks onto available processing elements such that residual resources necessary to allocate the dynamic aperiodic tasks may be maximized. In [46], Kato et al. reported that the average response time of aperiodic tasks may be improved by allowing online migration of some of the periodic tasks. Saez et al. in [47] introduced a global scheduling strategy where they checked the dynamic state of each processor and migrated aperiodic tasks to take advantage of the spare time on each processor and thus allow high resource utilization. Andersson et al. [48] proposed necessary and sufficient feasibility conditions to achieve low overhead exact admission control in EDF scheduled systems consisting of either, only aperiodic tasks, or a combination of persistent periodic tasks and dynamic aperiodic tasks. By extending this approach, Nie et al. [38] developed an on-line EDF based admission controller, called capacity-based admission control, for dynamically arriving periodic as well as aperiodic tasks.

2.4 Timed Discrete Event Systems [1, 2]

2.4.1 A Timed DES and it's behavior

DES is a discrete state space, event driven system that evolves with the occurrence of events, such as the arrival or completion of a task. In timed model of DES, both logical behavior and timing information are considered. Supervisory control of a TDES is timely disablement or enforcement of certain events in the transition structure of the TDES such that it's behavior meets certain specifications. Following the SCTDES, individual components, i.e., tasks of the system are modeled by an automaton:

$$G = (Q,\ \Sigma,\ \delta,\ \Gamma,\ q_0,\ Q_m),$$

where Q is the finite state space. Σ is the set of events. $\delta : Q \times \Sigma \longmapsto Q$ is the partial state transition function. $\Gamma : Q \to 2^{\Sigma}$ is the active event function. For all states $q \in Q$, $\Gamma(q)$ is the set of all events $\sigma \in \Sigma$ for which $\delta(q, \sigma)$ is defined and it is called the active event set of G at q. $q_0 \in Q$ is the initial state. $Q_m \subseteq Q$ is the set of marked states representing the completion of tasks. The event set Σ is partitioned into following disjoint subsets:

$$\Sigma = \Sigma_c \cup \Sigma_{uc} \cup \Sigma_{for} \cup \{t\},$$

where Σ_c is the set of *controllable events*: these are the events that can be prevented (or disabled) from happening by supervisor. Σ_{uc} is the set of *uncontrollable events*: these are the events that cannot be prevented from happening by supervisor. Event t denotes the *passage of one unit time*, or one *tick* of the global clock. Σ_{for} is the set of *forcible events*: these are the events that can preempt a *tick* of the global clock by forcing action of supervisor, and a forcible event itself may be either controllable or uncontrollable.

Let us denote by Σ^+ the set of all finite sequence of events of Σ, of the form $\sigma_1\sigma_2...\sigma_k$ where $k \geq 1$, $k \in N$ and $\sigma_i \in \Sigma$. Let $\epsilon \notin \Sigma$ be the empty event and define $\Sigma^* = \{\epsilon\} \cup \Sigma^+$. Transition function δ can be extended to Σ^* by defining $\delta(q, \epsilon) = q$ and $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$ for all $s \in \Sigma^*$ and $\sigma \in \Sigma$. The behavior of TDES G is described by a pair of languages, $L(G)$ and $L_m(G)$, where $L(G)$ is the set of all strings that the TDES can *generate* (i.e., the strings s such that $\delta(q_0, s)$ is defined). $L_m(G) \subseteq L(G)$ is the language of *marked* strings that is used to represent the completion of tasks (i.e., the strings s such that $\delta(q_0, s)\ \in\ Q_m$).

The *prefix-closure* of a language $L \subseteq \Sigma^*$ is denoted by $\overline{L}$ and consisting of all the prefixes of all the strings in L. $\overline{L} = \{s \in \Sigma^* : (\exists t \in \Sigma^*)\ [st \in L]\}$. L is said to be prefix-closed if $L = \overline{L}$. The prefix closure of a language L is relevant to control problems because it contains the evolutionary history of words in L. By definition, $L(G)$ is prefix-closed, i.e., $L(G) = \overline{L(G)}$, since a string is only possible if all its prefixes are also possible. However, $L_m(G)$ need not be prefix-closed, since not all states of TDES G need be marked. The *post-language* of L after s, denoted by L/s, is the language $L/s = \{t \in \Sigma^* : st \in L\}$. For $s \in \Sigma^*$, let $le(s)$ denote the last event of the string s, e.g., $le(abc) = c$ for $a, b, c \in \Sigma$ and $abc \in \Sigma^*$ [49].

2.4.2 Accessability and Co-accessability

A state $q \in Q$ is reachable or *accessible* if there is a string $s \in \Sigma^*$ with $\delta(q_0, s)$ is defined and $\delta(q_0, s) = q$. We denote the operation of deleting all the states of G that are not accessible by $Ac(G)$. Since Ac operation has no effect on $L(G)$ and $L_m(G)$, without loss of generality, we assume that an automaton is always accessible ($G = Ac(G)$) in this article. A state $p \in Q$ is co-reachable or *co-accessible* if there is a string $s \in \Sigma^*$ such that $\delta(p, s) \in Q_m$. TDES G that is accessible as well as co-accessible is said to be *trim*. G is *non-blocking* if every accessible state is co-accessible. It says that any string that can be generated by G is a prefix of (i.e. can always be completed to) a marked string of G, or equivalently $L(G) = \overline{L_m(G)}$. Likewise, G is said to be *blocking* if $L(G) \neq \overline{L_m(G)}$ and consequently *deadlock* can happen. G could reach a state q where there are no outgoing transitions and $q \notin Q_m$. This is called a deadlock because no further event can be executed. If deadlock happens, then necessarily $\overline{L_m(G)}$ will be a proper subset of $L(G)$, since any string in $L(G)$ that ends at state q cannot be a prefix of a string in $L_m(G)$.

2.4.3 Composition of TDES

There are two composition operations defined on TDES: parallel, denoted by $\|$, and product composition, denoted by $\times$. To construct an execution model of a composite task describing the concurrent behavior of individual tasks, we employ the *parallel composition* operation. Given two TDESs $G_i = (Q_i,\ \Sigma_i,\ \delta_i,\ \Gamma_i,\ q_{0i},\ Q_{mi})$ for $i = 1, 2$, the parallel composition of G_1 and G_2 is defined as follows:

$$G_1 \parallel G_2 = Ac(Q_1 \times Q_2,\ \Sigma_1 \cup \Sigma_2,\ \delta, \Gamma_{1\|2},\ (q_{01}, q_{02}),\ Q_{m1} \times Q_{m2}),$$

where

$$\delta((q_1,q_2),\sigma) = \begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ (\delta_1(q_1,\sigma),q_2) & \text{if } \sigma \in \Gamma_1(q_1) \setminus \Sigma_2 \\ (q_1,\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2) \setminus \Sigma_1 \\ undefined & \text{otherwise} \end{cases}$$

and $\Gamma_{1\|2}(q_1,q_2) = [\Gamma_1(q_1) \cap \Gamma_2(q_2)] \cup [\Gamma_1(q_1) \setminus \Sigma_2] \cup [\Gamma_2(q_2) \setminus \Sigma_1]$.

Product composition of $G1$ and $G2$ is defined as follows:

$$G_1 \times\ G_2 = Ac(Q_1\ \times\ Q_2,\ \Sigma_1 \cup \Sigma_2,\ \delta,\ \Gamma_{1\times 2},\ (q_{01},q_{02}),\ Q_{m1} \times\ Q_{m2}),$$

where

$$\delta((q_1,q_2),\sigma) = \begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ undefined & \text{otherwise} \end{cases}$$

and $\Gamma_{1\times 2}(q_1,q_2) = \Gamma_1(q_1) \cap \Gamma_2(q_2)$. Intersection of two languages can be obtained by performing product composition of their automaton representations, because $L_m(G_1 \times G_2) = L_m(G_1) \cap L_m(G_2)$[2].

2.4.4 Supervisor

Let automaton G represents the composed model of individual components (or tasks) in the system. Behavior of this G is expressed by languages $L(G)$ and $L_m(G)$. In addition to this, we have a set of *specifications* (or goals) that are subset of $L_m(G)$ and must be satisfied by G. Let us assume that each specification is modelled by an automaton and composed model is represented by K such that $K \subseteq L_m(G)$ and $K \neq \emptyset$. In order to ensure that the behavior of G stays within the given specification, we introduce an another automaton called *supervisor*, which is denoted by S. The transition function of G can be controlled by S in the sense that the controllable events of G can be dynamically enabled or disabled by S based on the observation of the events generated by system G. However, the supervisor does not generate events, it only provides the information about what are the events allowed for G to execute from its current state. If G follows the event set information provided by S, then $L_m(G)$ will be restricted such that it will never generate a string that is out of given specification K.

The abstract definition of a supervisor S is a function from the language generated by

G to the power set of Σ. Formally, $S : L(G) \to 2^{\Sigma}$. For each $s \in L(G)$ generated so far by G (under the control of S), $S(s) \cap \Gamma(\delta(q_0, \sigma))$ is the set of enabled events that G can execute at its current state $\delta(q_0, \sigma)$. Supervisor S is *admissible* if for all $s \in L(G)$,

$$\Sigma_{uc} \cap \Gamma(\delta(q_0, s)) \subseteq S(s)$$

which means that S is not allowed to ever disable a feasible uncontrollable event. Here $S(s)$ is the control action at s. We shall consider *admissible* supervisor in this paper. Given G and admissible S, the resulting closed-loop system is denoted by S/G (read as *"S controlling G"*). The controlled system S/G is a TDES, and we can characterize its generated and marked languages. The language generated by S/G is defined recursively as follows:

1. $\epsilon \in L(S/G)$
2. $[(s \in L(S/G))\ and\ (s\sigma \in L(G))\ and\ (\sigma \in S(s))] \Leftrightarrow [s\sigma \in L(S/G)]$
3. no other strings belong to $L(S/G)$

The language marked by S/G is defined as follows: $L_m(S/G) = L(S/G) \cap L_m(G)$. The $L_m(S/G)$ consists exactly of the marked strings of G that survive under the control of S. Supervisor S controlling TDES G is blocking if S/G is blocking and supervisor S is non-blocking if S/G is non-blocking. Since marked strings represent completed tasks, a blocking supervisor results in a controlled system that cannot terminate the execution of the task at hand.

2.4.5 Controllability

It is well known that given an uncontrolled system with behavior $L(G)$ and a desired specification $K \subseteq L_m(G)$, there exists a supervisor S such that $L(S/G) = \overline{K}$, which restricts the system behavior to the desired behavior by dynamically disallowing some of the *controllable* events while never preventing any of the *uncontrollable* events from occuring. For $s \in \Sigma^*$, the set of eligible events in G after processing string s is defined as $Elig_G(s) = \{\sigma \in \Sigma | s\sigma \in L(G)\}$. Similarly for K, $Elig_K(s) = \{\sigma \in \Sigma | s\sigma \in \overline{K}\}$. Then K is *controllable* (with respect to G) if for all $s \in \overline{K}$

$$Elig_K(s) \supseteq \begin{cases} Elig_G(s) \cap (\Sigma_{uc} \cup \{t\}) & \text{if } Elig_K(s) \cap \Sigma_{for} = \emptyset \\ Elig_G(s) \cap \Sigma_{uc} & \text{if } Elig_K(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

Thus K controllable means that an event σ may occur in K if σ is currently eligible in G and either (i) σ is uncontrollable, or (ii) σ = tick and no forcible event is currently eligible in K. The effect of the definition is to allow the occurence of tick (when it is eligible in G) to be ruled out of K only when a forcible event is eligible in K and could thus be relied on to preempt it. However, that a forcible event need not preempt the occurrence of completing non-tick events that are eligible simultaneously [1]. If the controllability condition is satisfied, then the supervisor that achieves exactly the required behavior, $\overline{K}$, is:

$$S(s) = \{\sigma \in \Sigma_c | s\sigma \in \overline{K}\} \ \cup \ [\Sigma_{uc} \cap \ \Gamma(\delta(q_0, s))].$$

This S is called as an *implicit supervisor* [50]. If this supervisor S is adjoined with the system G, then the behvaior of the resulting closed-loop system will be $L(S/G) = \overline{K}$. When K is not controllable, let $C(K)$ denote the family of controllable sublanguages of K.

$$C(K) = \{K' \subseteq K \mid K' \textit{ controllable } w.r.t. \ L(G)\}$$

$C(K)$ is always non-empty, since $\emptyset$ is controllable. The main result of [51] on controllability is that $C(K)$ has a unique largest controllable sublanguage $supC(K)$ such that $supC(K) \subseteq K$. So we design a *minimally restrictive* (or maximally permissive) supervisor which restricts the system behavior to the *supremal controllable sublanguage* of K, denoted by $supC(K)$. If this minimally restrictive supervisor which is synthesized by SCT is adjoined with the plant (or system), then the resulting controlled system will dynamically reconfigure itself to ensure $L(S/G) = \overline{supC(K)}$.

2.5 Field Programmable Gate Arrays (FPGAs) Its Evolution and Conceptual Background

2.5.1 Introduction to FPGAs

FPGA is an electronic device which consists of a matrix of reconfigurable logic circuitry, typically referred to as configurable logic blocks (CLBs), surrounded by a periphery of I/O blocks, as shown in Figure 2.3. When a FPGA is configured, the internal circuitry is electrically connected in a way that creates a hardware implementation of the desired application. Unlike processors, FPGAs use dedicated hardware (using CLBs as building blocks) for processing logic. Hence, FPGAs are truly parallel in nature and do not

have to compete for the same resources when processing different operations of an application, as is the case with a software implementation using processors. As a result, performance of different components of the application become mutually independent; additional processing incorporated in one component do not affect the performance of other components.

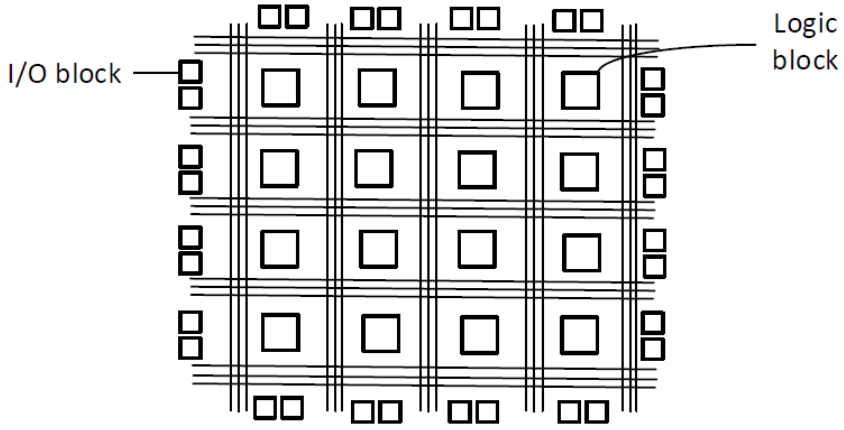


Fig. 2.3: Simple FPGA Internal Architecture

However, unlike single-purpose hardware designs (also referred to as, *Application Specific Integrated Circuits, ASICs*) with dedicated and fixed hardware functionalities, FPGAs can literally re-wire their internal circuitry to allow post-fabrication hardware programmability and thus, these devices are the enabling technology for reconfigurable computing. By incorporating programmability, FPGAs are able to combine the flexibility of software based implementations along with performance efficiencies often close to that of single purpose hardware or ASICs. A side effect of the power of reconfigurability is the ability to correct / modify design errors even after the implementation phase. This is unlike ASICs where it is almost impossible to rectify design errors post fabrication. This restriction increases the design, verification and testing overheads of ASICs by multiple folds, in terms of both cost and time-to-market. With the objective of alleviating such overheads, FPGA based implementations are often preferred over ASICs in situations where the intended device is not targeted for huge mass production. Due to the same reason, FPGAs are also popular as prototyping platforms before the mass manufacturing phase. However, power and area related overheads for FPGAs are typically higher than those for ASICs and this limits the design size.

2.5.2 Closer look into CLBs

A CLB, which may be considered as the unit building blocks of an FPGA, is essentially composed of a LUT (Look Up Table), register and decoder as shown in Figure 2.4. The register shown in the figure is used to synchronize the LUT output with a clock, if necessary. A LUT with 2^n locations along with a $2^n \times 1$ decoder is capable of implementing any n-input function. For example, Figure 2.4 shows how the LUT-decoder combination

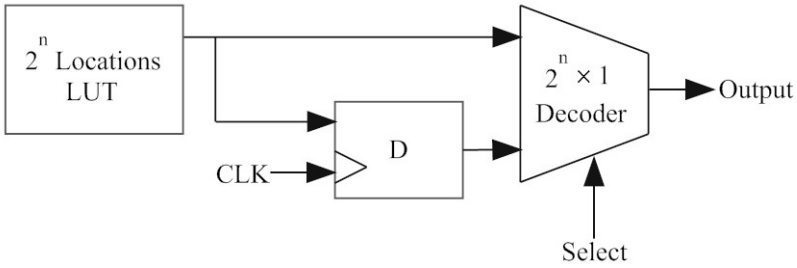


Fig. 2.4: Configurable Logic Block

can be used to implement the Boolean function $F = a \times b + \overline{a} \times c$ over inputs (a, b, c), by storing the appropriate output values in a LUT with 2^3 locations. It is clear that 2^{2^n} different n - input Boolean functions may be implemented by programming and storing appropriate values within a LUT with 2^3 locations and irrespective of the function implemented, the delay to produce an output remains same.

2.5.3 Heterogeneous FPGAs

Today, FPGAs are increasingly used in many computationally intensive applications with stringent performance requirements. In order to satisfy performance demands, these FPGAs often include specialized embedded Hard Blocks (HBs) such as memory blocks and DSP units within the uniform matrix of homogeneous CLBs, as shown in Figure 2.6. Inclusion of these HBs are making todays FPGA platforms more heterogeneous in nature. The most common embedded HBs are:

- Memory Blocks: In applications like image processing, huge amounts of intermediate data need to be frequently and temporarily stored during their processing. Use of embedded memory blocks in FPGAs have become crucial for the efficient implementation of these applications with reduced memory access delays. These memory blocks, often called Block RAMs (BRAMs) provide dedicated storage capacity up to about 1MB in many modern FPGAs.

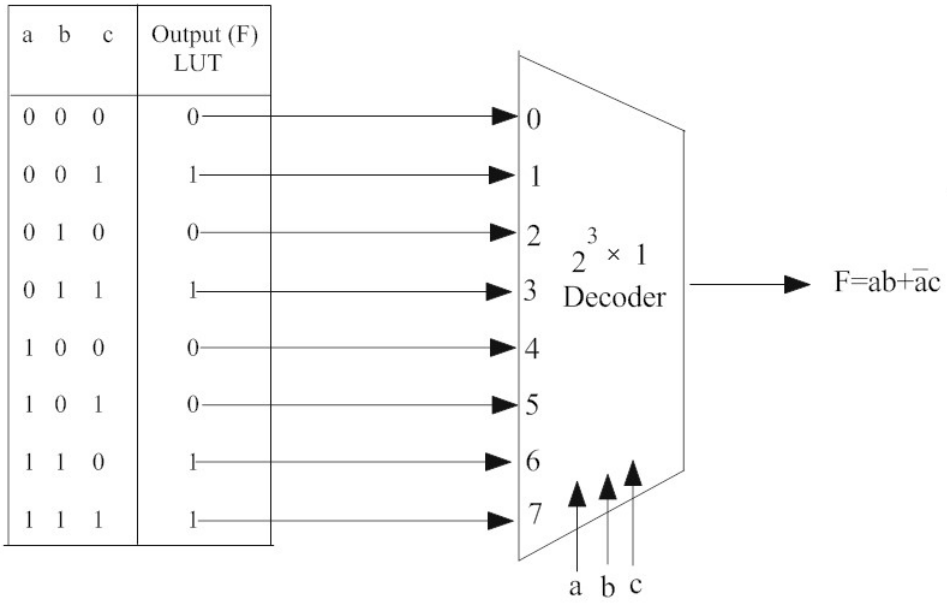


Fig. 2.5: Combination of a 2^3-location LUT with a $(2^3 \times 1)$ decoder to implement the function $F(a, b, c) = ab + \overline{a}c$, within a simple CLB

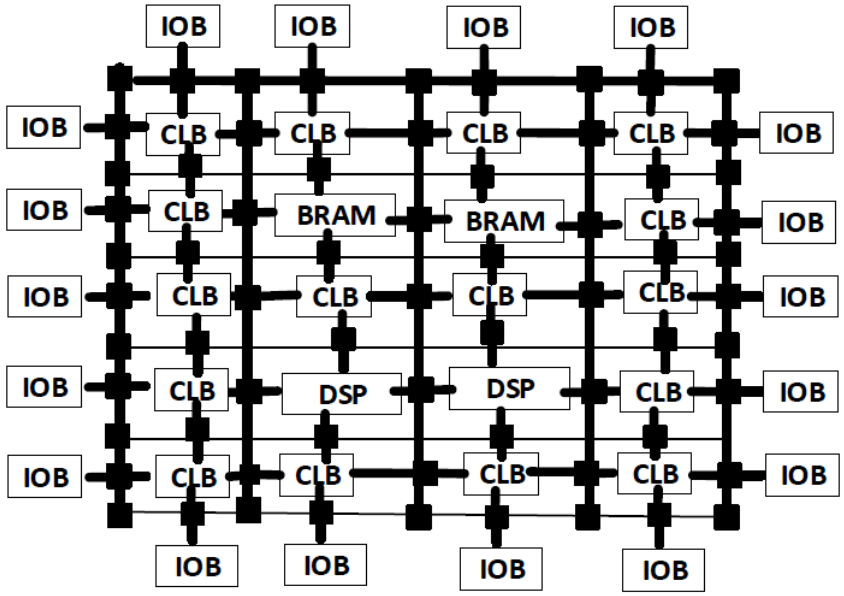


Fig. 2.6: Hetrogeneous FPGAs: A Conceptual Block Diagram

- Embedded DSP Blocks: Embedded DSP blocks in FPGAs commonly pro-vide many MAC (Multiply and Accumulate) units. Associated with the aforementioned BRAM embedded memory modules, they can be used to easily implement digital processing functions such as filters.

2.5.4 FPGA Design Flow

The FPGA design flow begins with the behavioural description of the intended application using a Hardware Description Language (HDL) like VHDL / Verilog or a schematic capture environment. This step of the design flow is interspersed with periods of functional simulation to verify the correctness of the intended design. After this step, the designer can at least be sure that his logic is functionally correct before going on to the next stage of development. At the next step, the design gets synthesized into an intermediate representation called netlist. The generated netlist is then passed through a translation process called place route. This step involves mapping the logical structures described in the netlist onto actual macrocells, interconnections, and input / output pins. The result of the place & route process is a bitstream which is the configuration data to be loaded in the configuration memory of the FPGA to implement the desired design. The generated bitstream may be downloaded into the FPGA through configuration interfaces like JTAG, SelectMap or Slave Serial ports and ICAP. These ports enable numerous reconfiguration techniques including compressed, encrypted and partial bitstream download and readback.

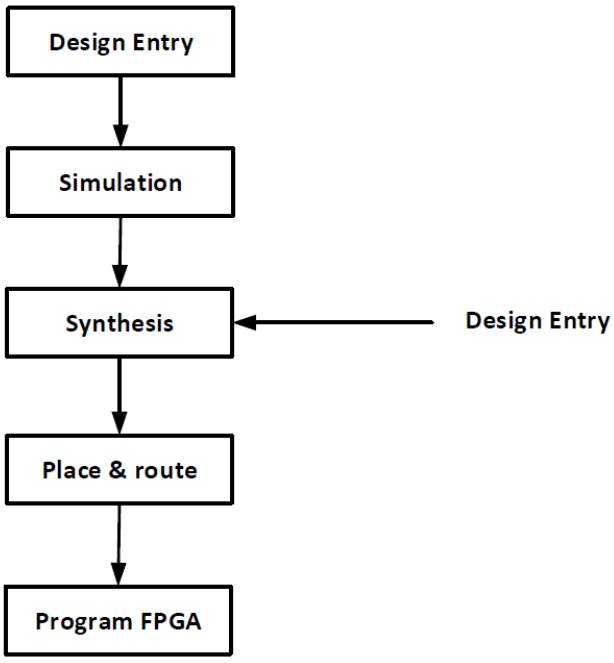


Fig. 2.7: Programmable Logic Design Process

2.5.5 Dynamic and Partial Reconfiguration

FPGAs are most commonly categorized as fully reconfigurable or partially reconfigurable. Initially, FPGAs came only as fully reconfigurable platforms where all logic resources on the entire FPGA floor at a given time must be reconfigured as a single atomic operation. Thus, all concurrently executing applications instantiated on different sub-regions of the FPGA floor must simultaneously halt at each reconfiguration event. Such a constraint restricted space-shared dynamic co-execution of independent applications running in different sub-regions within the area of the floor. This restriction was overcome through the advent of commercial platforms which also feature dynamic partial reconfiguration enabling a hardware instantiation running in a sub-region to be reconfigured while allowing instantiations in other sub-regions to continue execution uninterruptedly. One of the biggest companies offering such partial reconfiguration capabilities is Xilinx with their Virtex families. Xilinx FPGAs usually adopt the module-based style for partial reconfiguration. A module in a module-based reconfiguration is determined by a separate partial bitstream, also called a "task". Such a module in a particular FPGA sub-region may be reconfigured without interrupting the execution of other modules, by downloading the module's bitstream through the Internal Configuration Access Port (ICAP) into the configuration memory of the sub-region.

2.5.6 Real-time Hardware Tasks

A software task is defined as a set of instructions (a piece of code) and data (i.e. handled by those instructions) which is executed on a processor. Important parameters that characterize a real-time software task include execution time, deadline, response time, period, latency etc. Similarly, with respect to application execution on FPGAs, we bring in the concept of a hardware task. FPGAs contain a reconfigurable resource which basically consists of a 2-dimensional array of $W \times H$ Reconfigurable Processing Units (RPUs), commonly referred to as Configurable Logic Blocks (CLBs). A hardware task T_i in such system is a re-locatable digital circuit, typically rectangular in shape, which may be configured to be executed anywhere consuming a sub-region $w_i \times h_i$ on the floor (having total area $W \times H$) of the reconfigurable device. In addition to these geometric features, all parameters which characterize a real-time software task, also apply to real-time hardware tasks.

2.5.7 Preemption of Hardware Tasks

As discussed in section 2.1.3, a non-preemptive task only releases the CPU voluntarily. On the other hand, preemptive tasks can be interrupted (to let another task execute) and resumed later. One advantage of preemptive scheduling is its capability to schedule tasks with widely different periods. For example, let us consider a typical periodic task T_1, from the domain of automotive control systems, having an execution time of 10ms and period of 100ms. Even if this task is always assigned the highest priority in a non-preemptive system, execution time of all other tasks must not be greater than 90ms to ensure that no instance of T_1 ever misses its deadline in the worst case. This enforces manual partitioning of tasks having large execution times into small subtasks, making the design and implementation of the system very complex. Another advantage of preemptive scheduling is the possibility of admission control using utilization bounds and achieving an efficient resource utilization, up to 100% for a preemptive scheduler like Earliest Deadline First (EDF) [29].

In spite of the scheduling flexibility and higher resource utilization as possible in preemptively scheduled systems, there has been very little work in this regard for reconfigurable platforms. This is primarily due to the challenges involved in saving the state of a partially completed hardware task and restoring saved states to re-initiate execution. Hardware task contexts are stored in state-holding elements like Flip-flops and LUT-RAMs of CLBs and other Hard Blocks (BRAMs, MULs etc.). Switching context in any specific region of the FPGA involves:

1. Capturing the contexts of tasks that were executing in the region prior to a switch.
2. Updating the contexts of these captured tasks and saving them in external memory.
3. Forming a new bitstream comprising of tasks that should execute in the region subsequent to the switch.
4. Restoring the new bitstream in the region to re-initiate execution after preemption.

Authors in [52] have employed a mechanism called Scan Path generation to allow task context extraction / insertion on reconfigurable platforms. However, to enable hardware preemption, the methodology necessitates addition of task specific components known as scan-chains to hardware tasks, thus incurring significant spatial overheads. An improvement over this strategy is the bitstream readback methodology [53]. In [53], the au-

thors realized hardware context switching on Virtex-4 FPGAs using bitstream read-back through ICAP (Internal Configuration Access Port) along with some additional combinational logic inside the reconfigurable region. Now, traditional technologies for context switch, which required to save entire task bitstreams, incurred about $\approx$ 7ms to extract the context of a task having size $\approx$ 500 KB [53]. Needless to say, that such extraction time is unaffordably high. However, recent literature [16, 17] has discussed that the instantaneous state of a task is contained within at most $\approx$ 8% of its entire bitstream and it is sufficient to extract only this part of the bitstream during context switch. Through such selective bitstream read-back scheme, context switch overheads between tasks of typical sizes ($\approx$ 500 KB) may be drastically reduced to only $\approx 700\mu sce$ [17] using this intelligent selective extraction mechanism. Thus, the actual extraction overhead may be reduced to at most $1/10^{th}$ of the full bitstream read-back time. Hence, the overall context switch overhead also reduced. A detailed discussion on the quantification of hardware context switching overheads for a specific FPGA family can be found in next section.

2.6 Spatio - Temporal Scheduling of Hardware tasks

Parallel execution (spatial) of a given subset of tasks $Tp = \{T_{p1}, T_{p2}, \ldots T_{p|T_p|}\}$ is achieved through their simultaneous placement on the FPGA floor such that no task sub-region overlaps with the device boundaries or with other subregions. The vacant region within a set of already placed tasks whose area is not large enough to allow the feasible placement of any other task, is considered to be wasted due to fragmentation. One of the principal goals of any placement strategy is to reduce the total unutilized area lost due to fragmentation. A Placer assumes the responsibility of placing a task onto the FPGA floor after verifying that sufficient residual spatial resources are available to accommodate the task. If a feasible sub region to place the task is found, then a Loader downloads the task s bitstream through ICAP onto the configuration memory of the FPGA.

Due to limited resources of the FPGA, all tasks cannot be accommodated simultaneously. It is then necessary to additionally multiplex tasks over time on the available spatial resources. Scheduler's responsibility in such a system therefore, is to decide both where and when, to execute tasks in the system. Thus, the job of a real-time scheduler for FPGAs is to efficiently manage both space and time while appropriately accounting for reconfiguration related overheads, such that all deadlines are met. Hence, a spatio-temporal scheduling algorithm simultaneously conduct both temporal allocation along

with placement. To specific issues of importance for a dynamic scheduler may be pointed out here.

- Which tasks should be selected for execution at a scheduling point ?
- What are the best feasible regions available to place those tasks ?

2.7 Various Task Placement strategies for FPGAs

Given a set of tasks = { $T_1, T_2, ...T_n$ } to be executed at a given time, the problem of placement is to find a sub-region of size in $w_i \times h_i$ for each task T_i such that no sub-region overlaps with the device boundaries or with other subregions. The vacant region within a set of already placed tasks whose areas are not large enough to allow the feasible placement of any other task, is considered to be wasted due to fragmentation. One of the principal goals of any placement strategy is to reduce the total unutilized area lost due to fragmentation. However, the generic problem which attempts to either minimize the total area consumed by a given set of tasks, or maximize the number of tasks that may be feasibly accommodated within a given floor area, may be proved to be NPcomplete in the strong sense [54].

Therefore, researchers have delved towards devising various heuristic placement approaches [55, 56, 28]. One stream of work [57, 58, 56] has attempted placement using different restricted area models.

2.7.1 Task Placement For lD Area Model

In [58, 59], placement strategies have been discussed for one dimensional reconfigurable resources where the reconfigurable width W of the floor is divided into a constant number of equal sized columns (termed as tile), as shown in Figure 2.8. The placement strategy in this case therefore becomes a 1D bin packing problem. Steiger et al. [56] introduce the horizon and stuffing techniques for lD model. The "horizon" technique attempts to schedule new tasks when there are no overlapping in both time and space dimension with other scheduled tasks. The "stuffing" always places an arriving task on the leftmost column. However, with equal sized columns, resulting placements are often plagued by severe fragmentation.

Chen et a1. [60] propose a task placement method called classified stuffing technique to reduce the fragmentation on a 1D structure. The arriving tasks are placed on the

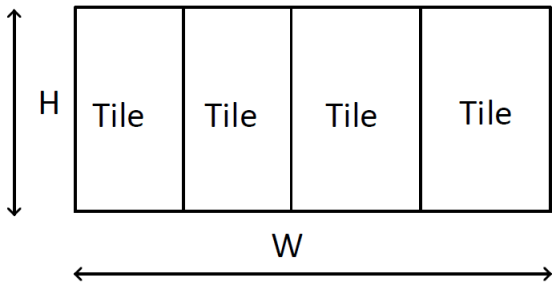


Fig. 2.8: 1D Area Model

basis of a parameter called Space Utilization Rate (SUR), which is defined as the ratio between the area requirement and the execution requirement of a task. Tasks with high SUR ($SUR > 1$) are placed in the leftmost available columns of the FPGA, while low SUR tasks ($SUR < 1$) are placed in the right most available columns.

Hubner et al. [61] proposed partitioning of the floor area of Virtex-II into vertical slots. A task can be placed in any slot. They also proved the possibility of placing different tasks on top of each other if the sum of the heights of these tasks do not exceed the height of the slot. This paper emphasizes on the issue of memory configuration and communication among tasks.

2.7.2 Task Placement For 2D Slotted Area Model

An improvement over the 1D slotted area approach is the 2D partitioned area model where both the width W and height H of the floor is partitioned into equal intervals to obtain a fixed number of equal sized rectangular tiles as shown in figure 2.9. Tiles are placeholders for hardware tasks and each tile can accommodate no more than one hardware task at a time. Many research works [62, 63, 64] have used the 2D slotted area model. Further relaxations on the 2D slotted area model lead us to generalized flexible 2D area model which allows the unrestricted freedom to place tasks onto any arbitrary region of the FPGA. As stated earlier, although this freedom brings in the possibility of more compact placement, it comes at the cost of much higher computational complexity. In general, relaxations on the 2D slotted area model (in order to provide higher resource utilization), lead to two important drawbacks:

- Even though they improve average performance for a set of best-effort tasks, they often degrade scheduling predictability which is of utmost importance, especially in real-time systems. This is because, it becomes more complex in this case to

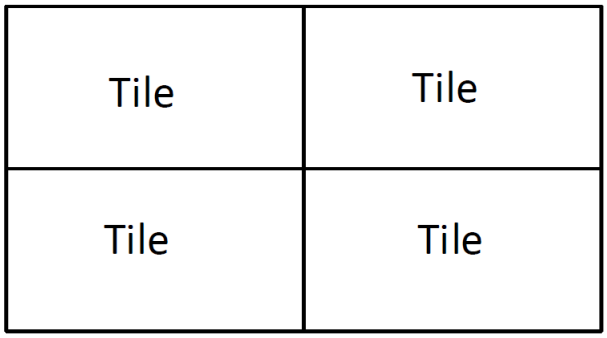


Fig. 2.9: 2D Area Model

deterministically account for the spatio-temporal capacity available in the worst-case, within a given time interval in future.

- These models tend to incur much higher overheads at each spatial scheduling point, making the complexity of the overall spatio-temporal scheduling problem very expensive towards online application.

2.7.3 Task Placement For Flexible 2D Area Model

This is the most flexible model that allows allocation of tasks anywhere on the device as shown in Figure 2.10. The advantage of this model is high device utilization as tasks can be placed tightly. However, the high flexibility of this model makes scheduling and placement more difficult. The 2D area model has the problem of external fragmentation: if tasks are placed on arbitrary positions the remaining free area is fragmented. Many recent research works [65, 66], have attempted heuristic approaches involving mechanisms to achieve two principal objectives: (i). Devising a strategy to maintain a record of all empty regions among already placed tasks and (ii). Finding an appropriate empty region as well as location within that region to place a task which dynamically arrives at run-time. Such dynamic addition and deletion of tasks may again lead to a fragmented floor area resulting in poor resource utilization.

In [67], Bazargan et al. proposed KAMER (Keeping All MERs), a mechanism which proceeds by maintaining a list of Maximal Empty Rectangles (MERs) that cannot be covered fully by a set of other empty rectangles. Whenever a task arrives, the algorithm places it in the bottom-left corner of the largest available MER in a worst-fit manner. After placement, the remaining empty area of the region is partioned either a vertical or horizontal split to produce two empty rectangular subregions. Although this method

produces good placement quality, a drawback is that a wrong splitting decision could cause the rejection of an otherwise feasible task. Walder et al. [68] proposed an enhancement version of the Bazarian partitioner which postpones the vertical/horizontal splitting decision until the arrival of a new task in order to overcome the possibility of wrong decision.

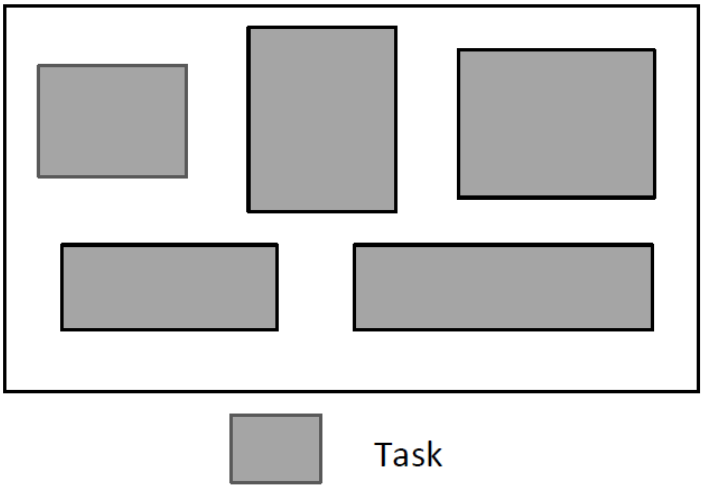


Fig. 2.10: 2D Flexible Area Model

The authors in [69] used FirstFit and BestFit placement strategies and claimed to achieve lower fragmentation and task rejection rates compared to KAMAR [67] and Enhanced Bazargon [68] methods. However, this work maintains the occupancy status of each CLBs of an FPGA in a 1D array and therefore, is susceptible to high computational overheads. A few other placement approaches aimed towards the efficient management of empty regions include algorithms by Handa et al. [70] (which employs a Vertex List Set(VLS), where a given free space fragment is presented by a list of vertices). Ahmadinia et al. [71] proposed management of occupied instead of free area as they observe that records for occupied spaces grow at a much slower rate than those for free spaces, making data management for the accommodation of new tasks simpler. Olakkenghil and Baskaran [72] propose a new data structure based on run-length encoding to manage free areas. In their work, the FPGA surface is modelled by a matrix coded according to reflected binary gray curve. Summary of the different placement strategies are shown in table 2.1. *The contributions presented in this thesis use the flexible 2D area model.*

Table 2.1: Summary of placement strategies

Placement Techniques	Area partition	Task allocations	Advantage	Disadvantage
1D stuffing	Partition the 1D area model into equal sized columns(Tiles)	Place arriving tasks on the leftmost column	Easy	Fragmentation
1D classified stuffing	Partition the 1D area model into equal sized columns(Tiles)	Placement is done based on the Space Utilization Ratio(SUR)	Minimize fragmentation for 1D stuffing	Computation demanding compared to 1D stuffing
2D area model	Partition the 2D area model into equal sized rectangular tiles	Task allocation is fixed to a particular rectangle(Tile)	Improve average performance	Incur higher overheads at each spatial scheduling pts.
Bazargan KAMER (Flexible 2D area model)	The remaining empty region is partitioned either a horizontal or vertical split	Place tasks on the bottom-left corner of the largest available Maximal Empty Rectangle (MER)	-good placement quality	wrong splitting decision could cause the rejection of an otherwise feasible task
Enhanced Bazargon (Flexible 2D area model)	The remaining empty region is partitioned either a horizontal or vertical split when task arrive	Place tasks on the bottom-left corner of the largest available MER)	overcome the possibility of wrong split	Fragmentation

2.7.4 Task Placement For 2D Heterogeneous FPGAs

Many FPGAs may contain not only CLB blocks but also embedded static components (as BRAM blocks, multipliers and DSPs) in a certain disposition and this heterogeneity imposes stricter placement constraints for the task. A task including static components cannot be placed anywhere on the FPGA because their feasible positions are limited by the locations of static components on the FPGA. Few algorithms deal with task placement on 2D heterogeneous architecture.

In [73], authors proposed an algorithm to deal with heterogeneous hardware constraints while placing a task. Feasible placement positions of the given hardware tasks are determined offline. At run-time, a task is placed at the first available free position. A generalization over this problem model is proposed in [74], where a task can have multiple instances with the scheduling objective being minimization of task rejection ratio. Authors in [62] used a 2D slotted area model and classified tasks according to their resource requirements. The tasks with the maximum required resources in a given class represents a slot in the FPGA. Hence this slot can accommodate any task belonging to the class. A new online placement algorithm which utilizes the symmetry in arrangement of HBs (BRAMs) on the FPGA floor has been discussed in [75]. The algorithm maintains the positions of the BRAMs along with the separating distance between BRAMs. On the arrival of a new task, the algorithm scans the unoccupied BRAMs from left to right and top to bottom and appropriately places the task.

2.7.5 Real-time Preemptive scheduling: Uniprocessors Vs Multiprocessors Vs FPGAs

Significant amount of work has already been done in on-line scheduling of real-time tasks on reconfigurable architectures. The work proposed in [76] [77][78] [79] [80] demonstrate the preemptive and non-preemptive periodic task scheduling on the reconfigurable systems. Preemption is an important technique that allows real-time systems to achieve high resource utilization by lending it the flexibility to co-schedule tasks having different behaviours (in terms of execution times, periodicity, task recurrence etc.) in varying execution environments (ranging from uniprocessors, homogeneous / heterogeneous multiprocessors and even reconfigurable processing cores). Non-preemptive tasks are characterized by the fact that once allocated a processor at the beginning of execution, the processor cannot be relinquished from the task until its completion. Preemptive tasks

on the other hand, can be interrupted (to possibly allow other tasks to execute) before completion and resumed later. An important advantage of preemptive scheduling is its capability to schedule tasks with widely varying periods. For example, let us consider a typical period task T_1, from the domain of automotive control systems, having an execution time of 10ms and period of 100ms. Even if this is always assigned the highest priority in a non-preemptive system, execution time of all other tasks must not be greater than 90ms to ensure that instance of T_1 ever misses its deadline in the worst case. To ensure this, tasks having large computation times must be manually partitioned into small subtasks, and this makes the scheduler design very tedious especially in large and complex systems. In comparison, by allowing preemption, dynamic unicore scheduling strategies like Earliest Deadline First (EDF) [81] achieves full resource utilization without imposing any restriction on tasks execution times and / or deadline / periods. These preemptive systems are also practically realizable in most cases because context switching overheads involved in each preemption is typically very low and may usually be neglected.

So in a nutshell, a major drawback of non-preemptive scheduling approaches is that it may severely restrict resource utilization especially in scenarios where the individual task utilizations or their periods are skewed [82]. The situation becomes worse as we shift from unicore to multicore systems with numerous processing elements [83]. Traditionally, there have been two principal approaches towards real-time scheduling of tasks on multicore systems global scheduling and partitioned scheduling [84]. In fully partitioned scheduling, all tasks are first assigned to dedicated cores. Tasks allocated to a given core are executed on that core until completion. The main advantage of partitioned scheduling is that the multicore scheduling problem can be amicably reduced to a set unicore problems and scheduled using well-known approaches like EDF. However, a critical drawback of this approach is that even by employing preemptive scheduling approaches like EDF (which offer 100% resource utilization on unicore systems), not more than 50% of the system capacity may be utilized in the worst case [34]

In global scheduling on the other hand, all tasks are maintained in a single ready queue. At each scheduling event, the m (denotes the number of cores) highest priority tasks are selected from this queue. By allowing the flexibility of intercore task migrates during execution. global schedulers can typically achieve significantly superior resource utilization compared to partitioned schemes. Pfair and its work conserving variant ERfair [84], both fully global schemes, were the first multicore scheduling strategies that allowed optimal resource utilization irrespective of the skewness in task weights / periods. This

optimally was achieved by maintaining proportional fair execution progress for all tasks at each time instant throughout the length of the schedule. However, in order to provide such accurate proportional fairness, these schemes incur unrestricted migrations and preemptions which leads to high context switch related overheads even in closely-coupled multicores with shared caches.

The need to control context switch overheads led to the design of more recent semi-partitioned approaches like DP-Fair and Bfair. These approaches partition time into slices, demarcated by the arrivals and departures of all the jobs in the system. Within a time slice, each task is allocated a work load equal to its proportional fair share. The task shares within each core are usually scheduled using EDF-like strategies and completed by the end of the time slice. By deviating from the need to maintain strict proportional fairness at all times, these strategies are able to guarantee resource utilization optimality while incurring at most $m1$ migrations within time slices. Thus, context switch overheads in these semi-partitioned schemes are significantly reduced in comparison to global approaches like ERfair. However, although the semi-partitioned DP-Fair scheme may be considered a prominent state-of-the-art scheduler for dynamically arriving task sets in multi-core systems, it cannot be directly employed in platforms such as FPGAs. This is due to the inherent architectural constraints in FPGAs which lead to non-negligible reconfiguration overheads in the order of a few to tens of milliseconds [57]. Thus, in case of FPGAs, context switches come at a premium and they must be judiciously handled as well as have to be correctly accounted for within any given interval of time. Otherwise, correct estimation of available system capacity will not be possible and this may lead to huge task deadline misses. Due to this fact, algorithms for general purpose multicore systems are bound to fare poorly on FPGAs.

2.8 Formal and heuristic scheduling for FPGAs: A Survey

In this section, a brief survey of formal and heuristic (in the context of spatio-temporal scheduling) approaches to real-time Scheduling on reconfigurable systems is presented.

2.8.1 Formal approaches for Scheduling tasks on FPGAs

In literature, it has been experimentally shown that DPR in combination with accelerators results in: (i) better utilization of the FPGA resources, (ii) performance that is comparable to non-reconfigurable solutions, and (iii) tighter WCET (Worst Case Execu-

tion Time) bounds [85]. Biondi et al. proposed a programming framework, named *FRED*, to support the development of real-time applications upon heterogeneous platforms, providing a predictable infrastructure that can ensure bounded delays when requesting a dynamically-reconfigured hardware accelerator [86]. However, this work presented only a proof-of-concept of FRED on top of *FreeRTOS*. Later, Pagani et al. proposed an implementation of the *FRED* framework on the Linux operating system addressing several challenges such as, architectural support for the accelerators, reconfiguration and communication mechanisms, implementation of FRED scheduler, and synchronization mechanisms between software and hardware tasks [87]. Seyoum et al. developed DART, a tool that fully automates the design flow in a real-time DPR-based system that comprises both software and hardware components [88]. Casini et al. proposed a holistic framework to help designers partition real-time applications on heterogeneous platforms with hardware accelerators [89]. Goossens et al. studied the hard real-time scheduling of multi-mode applications on reconfigurable heterogeneous hardware platforms [90]. Valente et al. proposed an approach to calculate DPR time and analyzed when it is really useful to exploit DPR capabilities, for real-time applications running on heterogeneous platforms [91].

The work in [92] proposed a combined offline-online scheduling algorithm of preemptive real-time tasks on partially reconfigurable systems. As electronic systems design can no longer be seen as an isolated hardware design activity, unified modelling language (UML) becomes of significant interest as a unification language for system descriptions combining both hardware and software components [93]. The UML design for dynamically reconfigurable embedded systems is shown in [93, 94, 95]

Offline formal approaches towards the design of reconfigurable controllers/schedulers have become more suitable to solve the problem of scheduling tasks on reconfigurable systems than the manual encoding and analysis approach. Aylward et al [96], L. Gong et al [97], addressed the formal methods that can provide attractive verification techniques that apply to recongurable embedded-system designs. Singh and Lillieroth [98] present a typical study addressing the correctness of recongurable cores, such as a 64b adder and an 8b counter. They consider a formalization based on propositional logic and integer arithmetic. They use a theorem-prover at runtime to check whether the dynamically calculated circuits are correct.

The work in X. An et al [99] explored the model-based design of correct controllers for dynamically recongurable architectures. They formalize the behaviours of the DPR FPGAs as automata, following a modelling methodology, distinguishing the different lev-

els of hardware architecture, task implementation, and application software. They used a tool (BZR language and compiler) to implement the models, solve the control problems and generate an executable code. Authors in [100, 101] have proposed a dynamic partial reconfiguration of concurrent control systems implemented in field programmable gate array (FPGA) devices. They apply Petri nets and a unified modelling language state machine diagrams respectively as a specification of the system. Guillet et al [102] presented a reconfiguration controller for a Dynamic Partial Reconfiguration. They used a tool (MARTE and BZR language and compiler) to implement the models and synthesize them using the Discrete Controller Synthesis formal technique.

2.8.2 Spatio-Temporal Scheduling for FPGAs

Diessel and Elgindy [103], combined placement along with a temporal scheduling mechanism for non-real-time preemptible task sets. Tasks are allocated from a ready queue starting from the bottom-left position using the first-fit strategy. When the allocator fails to accommodate the next pending task, it attempts to create additional space by preempting and locally repacking the currently running tasks utilizing spare logic resources created due to the partial completion of execution of these tasks.

Spatio-temporal scheduling of tasks with preassigned priorities have also been considered for reconfigurable systems, where priority could be based on the temporal (deadline, laxity) or geometrical (size, aspect-ratio) properties of the tasks (or both) [104]. Walder and Platzuer [68] used non-preemptive online scheduling schemes like *First Come First Serve (FCFS)* and *Shortest Job First (SJF)* for block-partitioned 1D reconfigurable devices. In [60], authors present a scheduling mechanism called *Classified Staffing (CS)*, where both geometrical and temporal parameters have been used to obtain priorities. The tasks are first inserted into a ready queue on their temporal priorities (For no real-time tasks, these priorities are obtained using the *Shortest Remaining Processing Time (SRPT)* strategy, while for non-real-time tasks, the *Least Laxity First (LLF) strategy is followed*). The tasks in the ordered ready are then placed based on a Spatio-temporal parameter called *Space Utilization Rate (SUR)*, which is defined as the ratio between the area requirement and the execution requirement of a task. Using a 1D area model, tasks with high SUR ($SUR > 1$) are placed in the leftmost available columns of the FPGA, while low SUR tasks ($SUR < 1$) are placed in the rightmost available columns.

Due to the rightmost involved in the hardware tasks preemption on FPGAs. non-

preemptive scheduling strategies have usually been employed. In [78], authors presented a preliminary work on non-preemptive FPGA-based real-time task scheduling and placement using Clairvoyant EDF [105] with some modifications. However, one of the major drawbacks of real-time non-preemptive scheduling approaches is that they may severely restrict resource utilization, especially in scenarios where the individual task utilizations or their periods are skewed [82, 83]. Although few, there exist works that have employed preemptive scheduling techniques for reconfigurable platforms.

Danne and Platzner [53] consider the problem of scheduling preemptive periodic real-time tasks on FPGAs by using a flexible 2D area model. They proposed the EDF-Next Fit (EDF-NF) algorithm, a variant of EDF [81], which uses the concept of master processes (servers) to reserve the area and execution time of tasks. However, as the paper reveals, being based on EDF, this algorithm cannot achieve a resource utilization of more than 50% in the case of generic systems.

Assuming the 1D area model, Danne and Platzner [80] proposed the EDF-First-k-Fit (EDF-FkF) and EDF-Next Fit (EDF-NF) algorithms for scheduling preemptive periodic real-time tasks on FPGAs. Both the algorithms start by generating a list Q of all active tasks sorted in the earliest deadline first order. EDF-FkF selects the maximum number of k of consecutive tasks, starting from the first task in Q, which can be accommodated within the area of FPGA. On the other hand, EDF-NF conducts a linear search from the beginning of Q, selects the next task provided it can be feasibly accommodated within the area of the FPGA, and continues the search until no more tasks can be accommodated. Guan et al. [77] extended the approaches presented by Danne and Platzner by incorporating actual reconfiguration overheads corresponding to vertex-4 FPGA in the algorithms.

To achieve higher resource utilization and minimize task rejections, a different dynamic scheduling and placement approach has been discussed in [28]. In this article, the author assumes that a hardware task could have multiple hardware variants (varying size) such that, the larger the size of a task, the faster is its performance. The selection of appropriate hardware task variants is a trade-off between the maximal utilization of reconfigurable resources versus the timing requirements of the tasks. A similar algorithm that considers multiple hardware task shapes has been proposed in [106]. The authors showed that resource utilizations using conventional scheduling algorithms like EDF and LLF may be significantly improved by using the flexibility of multiple shapes, against a rigid task scenario. This work neglected reconfiguration overheads and assumed tasks to be soft

real-time in nature. However, reconfiguration overhead is a major constraint that may adversely affect the temporal performance of tasks if not handled appropriately. In [107], authors described the *reuse and partial reuse* approach which allows a single configured task on the reconfigurable floor to be shared and reused among multiple applications, thus reducing the number of reconfigurations required.

Works on scheduling for systems that include aperiodic tasks on fully and partially reconfigurable FPGA platforms, are very few. This is partly due to the fact that the design of scheduling methodologies for FPGAs involves considerable engineering challenges, especially in the face of the architectural and temporal constraints that must be satisfied. A dynamic aperiodic task handling mechanism for partially reconfigurable platforms has been presented in [108]. Here, Farag et al. proposed a utilization-bound oriented acceptance test for aperiodic tasks by modifying the Synthetic Utilization based admission control policy discussed by Abdelzaher et al. in [109].

Among few of the existing research works based on dependent task scheduling on FPGAs, authors in [75] discussed an ILP formulation and heuristic policy for the Spatio-temporal scheduling of dependent non-real-time tasks on partially reconfigurable FPGAs. Here the authors considered the 2D flexible area model and thus, dealt with strict placement constraints. The scheduling objective is to minimize the overall makespan time corresponding to the execution of the task graph. The work in [110] presents a reconfigurable operating system (ROS) framework for the incoming task graph that aids the designer from the early design stages to the actual hardware implementation. They proposed an Island Based Genetic Algorithm flow that optimizes several objectives including performance, area, and power consumption. Another research work in [8] proposes an online task scheduling algorithm that targets the 2D FPGA area partitioning model and takes into account the data dependency and the data communications. They proposed a heuristic policy named *"Communication Aware online task Scheduling Algorithm (CASA)"* to achieve their purpose. Khuat et al. [75] discussed an ILP formulation and heuristic policy for the Spatio-temporal scheduling of dependent non-real-time tasks on partially reconfigurable FPGAs. Here the authors considered the 2D flexible area model and thus, dealt with strict placement constraints. The scheduling objective is to minimize the overall makespan time corresponding to the execution of the task graph. The work in [9] has proposed a methodology for a dependent hardware task mapping, based on the availability of multiple architectural variants for each hardware task. It is shown that their proposed approach significantly improves the total execution time, by the use

of trade-offs in resource consumption and data throughput for each hardware task. All the above works lack real-time constraints. The work in [10] proposes an operating System for runtime reconfigurable multiprocessor systems. The hardwareOS, called CAP-OS (Configuration Access Port-Operating System) was primarily developed to manage the usage of the available Internal Configuration Access Ports of the recently released Xilinx FPGAs. The partial bitstreams, the software executable, and the task graphs of the applications are required by the CAP-OS. The CAP-OS is also responsible for the runtime scheduling of the configurations of the different tasks, allocating the tasks to the processing elements, and resource management.

2.9 Summary

This chapter started with a brief overview of real-time systems followed by scheduling algorithms on uniprocessors and multiprocessors. Then, we have pointed out the fundamental definitions of the supervisory control of timed discrete event systems. Subsequently, we have discussed the evolution of FPGA architecture, the various task placement strategies for FPGAs, and the formal and heuristic scheduling for FPGAs. In the next chapter, we present the scheduler synthesis scheme for a set of non-preemptive periodic real-time tasks executing on a FPGA platform.

Chapter 3

A Supervisory Control Approach for Scheduling Real-time Periodic Tasks on FPGAs

In the last chapter, we discussed various real-time scheduling algorithms on uniprocessor and multiprocessor systems. Also, the scheduler synthesis using Supervisory Control of Timed Discrete Event Systems (SCTDES) was presented. The evolution of the FPGA architectures and the different Spatio-temporal scheduling strategies have been discussed. As mentioned earlier, this dissertation is oriented towards the formal and heuristic approaches to real-time scheduling on reconfigurable systems. With this objective, we present a formal scheduler synthesis framework for the set of real-time non-preemptive periodic tasks executing on an FPGA platform, using supervisory control of timed discrete event systems as the underlying formalism. We show the practical viability of our proposed framework by synthesizing schedulers for small applications and implementing them on the Atlys development board.

3.1 Supervisory Control of DES

In this section, we briefly discuss the concepts related to SCDES, the underlying formalism employed in this work.

A DES and its Behavior [111]: First, individual system/specification components are modeled by an automaton: $Y = (Q,\ \Sigma,\ q_0,\ Q_m,\ \delta)$, where, Q is the finite set of *states*, Σ is the set of *events*, $q_0 \in Q$ is the *initial state*, $Q_m \subseteq Q$ is the set of *marked*

states, $\delta : Q \times \Sigma \longmapsto Q$ is the (partial) state *transition function*. Here, the events in Σ are further categorized as follows: (i) Σ_c: *controllable*; supervisor has an ability to prevent the occurrence of these events. (ii) Σ_{uc}: *uncontrollable*; supervisor cannot prevent the occurrence of these event. We use $q \xrightarrow{\sigma} q'$ to denote $\delta(q, \sigma) = q'$.

The *closed* behavior of Y: $L(Y) = \{s \in \Sigma^* | \delta(q_0, s) \text{ is defined}\}$. The *marked* behavior of Y: $L_m(Y) = \{s \in \Sigma^* | \delta(q_0, s) \in Q_m\}$. $L_m(Y) \subseteq L(Y)$ represents all finite sequences that lead to the satisfaction of a phenomena.

The *prefix-closure* of a language $L \subseteq \Sigma^*$ is denoted by $\overline{L}$ and $\overline{L} = \{s \in \Sigma^* : (\exists x \in \Sigma^*)\ [sx \in L]\}$.

Reachability, Co-reachability and Trim [2]: A state $q \in Q$ is *reachable*, if $\delta(q_0, s) = q$, for some $s \in \Sigma^*$. DES Y is said to be reachable, if q is reachable for all $q \in Q$. A state $p \in Q$ is *co-reachable*, if $\delta(p, s) \in Q_m$, for some $s \in \Sigma^*$. A DES Y is said to be co-reachable, if p is co-reachable for every $p \in Q$. A DES Y that is reachable as well as co-reachable is said to be *trim*. Y is said to be *non-blocking*, if $L(Y) = \overline{L_m(Y)}$. Otherwise, Y is said to be *blocking*.

Synchronous Product [111]: Given the DES models of individual system/specification components, their composite model can be obtained using *synchronous product*. Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ with $\Sigma = \Sigma_1 \cup \Sigma_2$ and the natural projection $P_i : \Sigma^* \to \Sigma_i^*$ defined by: (i) $P_i(\epsilon) = \epsilon$, (ii) $P_i(\sigma) = \epsilon$ if $\sigma \notin \Sigma_i$, (iii) $P_i(\sigma) = \sigma$ if $\sigma \in \Sigma_i$ and (iv) $P_i(s\sigma) = P_i(s)P_i(\sigma), s \in \Sigma^*, \sigma \in \Sigma$. The *inverse projection* of P_i is, $P_i^{-1} : 2^{\Sigma_i^*} \to 2^{\Sigma^*}$. The *synchronous product* of L_1 and L_2, denoted by $L_1||L_2$, is defined as $L_1||L_2 = P_1^{-1}L_1 \cap P_2^{-1}L_2$. Synchronous product may be employed to construct an execution model of a composite task describing the concurrent co-execution behavior of its constituent individual tasks.

Supervisor and Controllability: Let automata Y and K represent the composite models of individual components in the system and specification, respectively, such that $L_m(K) \subseteq L_m(Y)$ and $L_m(K) \neq \emptyset$. Given the system Y and its specification $K \subseteq L_m(Y)$, $K \neq \emptyset$, supervisor $S = Y||K$ is computed. A *supervisory control* for Y is a map $S : L(Y) \to 2^{\Sigma}$. While doing this, the supervisor must respect controllability of the events. That is, the supervisor must restrict the system behavior to the desired specification by disallowing *controllable* events that might lead to undesired behavior, while never preventing any of the *uncontrollable* events from occurring.

3.2 Proposed Scheduler Synthesis Scheme

In this section, we present the design of the scheduler synthesis mechanism. We start with the presentation of the system model and assumptions considered in this work.

3.2.1 THE MODELS

***System Model*:** This work considers a system model which consists of a dynamically reconfigurable FPGA platform, a separate General Purpose Processor (GPP) and a memory. Tasks or applications are represented as bitstream images (also referred to as Reconfigurable Module (RM)) and are stored and maintained in a repository residing in memory. Reconfigurations of the FPGA are performed under supervision of the GPP or through the reconfigurable controller module, by loading bitstreams from the repository into the configuration memory of the FPGA through its ICAP port. Once a task arrives, GPP kick-starts its execution on the FPGA by loading its implementation. After the completion of execution on the FPGA, computed results are returned back.

The architecture of the FPGA has been assumed to be similar to that of the Xilinx Virtex series of FPGAs. These FPGAs basically consist of a 2D array of CLBs. The floor area (denoted by K) of the FPGA is partitioned into q equi-sized tiles (also referred to as Partially Reconfigurable Region (PRR)), $\{k_1, k_2, \ldots, k_q\}$. A PRR can hold one RM only at any point of time. However, a task (RM) can consume multiple PRRs based on its resource requirements. This RM-PRR binding takes place at compile-time and are stored as bitstream images. Thus, the combinations of one or more RRs can be configured to execute tasks by loading predefined *bitstreams*. We consider run-time partially reconfigurable systems. Hence, a RM in a particular PRR may be reconfigured without interrupting the execution of other RMs.

***Application Model*:** This consists of a set of n independent real-time periodic applications/tasks: $T = \{T_1, T_2, ..., T_n\}$. Tasks in T needs to be scheduled on a set of q equi-sized PRRs. These tasks are non-preemptive in nature, i.e., once the execution of a task is started on its assigned RR, the RR cannot be reconfigured until the completion of the task. In a reconfigurable system, a task T_i denotes a relocatable digital circuit, logically represented through a corresponding *bitstream*.

We assume that each task T_i can have m_i distinct implementations: $\{T_{i,1}, T_{i,2}, \ldots, T_{i,m_i}\}$. The j^{th} implementation of T_i is characterized by, (i) spatial requirement $\mathcal{I}_{i,j}$ in terms of the required PRRs to place $T_{i,j}$, (ii) temporal requirement in terms of the Worst Case Execu-

tion Time (WCET) of $T_{i,j}$ ($E_{i,j}$) along with associated reconfiguration loading/unloading time $RD_{i,j}$. It may be noted that the time required for loading/unloading a task on a partially reconfigurable device is directly proportional to the task's bitstream size [92].

A periodic task T_i is represented as, $\langle A_i,\ D_i, P_i,\ \langle \mathcal{I}_{i,1},\ RD_{i,1},\ E_{i,1}\ \rangle,\ \ldots,\ \langle \mathcal{I}_{i,m_i}, RD_{i,m_i},\ E_{i,m_i}\ \rangle \rangle$, where

- A_i is the *arrival time* of the task T_i.
- D_i is the *relative deadline* of task T_i.
- P_i denotes the *period* of T_i and $P_i \geq D_i$.
- $\mathcal{I}_{i,j}$ $(j = 1, 2, \ldots, m_i)$ is the set of slots required to place the j^{th} implementation $T_{i,j}$ of T_i.
- $RD_{i,j}$ $(j = 1, 2, \ldots, m_i)$ is the *reconfiguration download time* associated with $T_{i,j}$.
- $E_{i,j}$ $(j = 1, 2, \ldots, m_i)$ is the *WCET* of $T_{i,j}$.

All these parameters are assumed to be *discrete* and *finite*.

Table 3.1: Task characteristics

Tasks	$\{\mathcal{I}_{i,1}, RD_{i,1}, E_{i,1}\}$	$\{\mathcal{I}_{i,2}, RD_{i,2}, E_{i,2}\}$	A_i, D_i, P_i
T_1	$\{\{k_1\}, 1, 2\}$	$\{\{k_1, k_2\}, 2, 1\}$	$\langle 0, 5, 5\rangle$
T_2	$\{\{k_2\}, 1, 2\}$	$\{\{k_2, k_3\}, 2, 1\}$	$\langle 0, 4, 5\rangle$
T_3	$\{\{k_3\}, 1, 2\}$	$\{\{k_2, k_3\}, 2, 1\}$	$\langle 0, 7, 10\rangle$
T_4	$\{\{k_1\}, 1, 2\}$	$\{\{k_1, k_2\}, 2, 1\}$	$\langle 0, 9, 10\rangle$

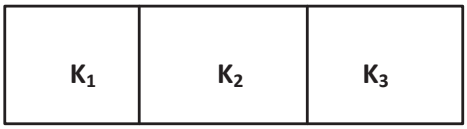


Fig. 3.1: Three Slots

***Example*:** Figure 3.1 shows an example reconfigurable platform consisting of three slots: k_1, k_2, k_3. We consider the execution of four tasks T_1, T_2, T_3 and T_4 whose execution parameters for the given platform is captured in Table 3.1. We assume $D_i = P_i$. It can be seen that the implementations $T_{1,1}$ and $T_{1,2}$ of task T_1 require the slots/tiles $\mathcal{I}_{1,1} = \{k_1\}$

and $\mathcal{I}_{1,2} = \{k_1, k_2\}$, for feasible placement. The reconfiguration loading time of $T_{1,1}$ is 1 time unit. The execution of $T_{1,1}$ and $T_{1,2}$ are 2 and 1 time units respectively.

Problem Statement: *Given a set of real-time non-preemptive periodic tasks* ($T = \{T_1, T_2, ..., T_n\}$) *to be executed on a reconfigurable computing platform* ($K = \{k_1, k_2, \ldots, k_q\}$), *design a supervisor (i.e., scheduler) which contains a feasible schedule satisfying all timing and resource constraints (if such a schedule actually exists).*

In order to synthesize a scheduler by employing the Supervisory Control Theory of TDES [111]. First, we discuss the various stages involved in the execution of tasks on reconfigurable FPGAs.

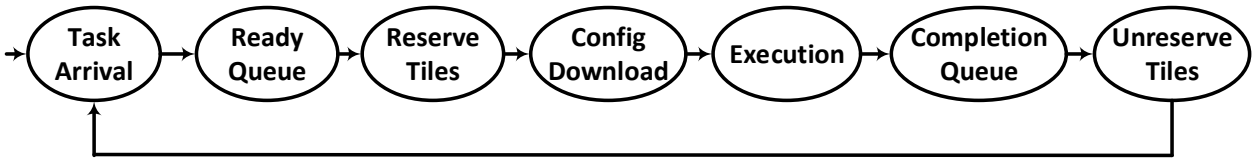


Fig. 3.2: State transition diagram capturing task execution on reconfigurable FPGA platform

3.2.2 Task execution on reconfigurable FPGA platforms

First, we discuss the various stages involved in the execution of tasks on reconfigurable FPGAs. The typical execution flow of a task T_i on a reconfigurable FPGA platform is captured as a state transition diagram shown in Figure 3.2. Whenever a task T_i is released/ready for execution, it is moved to the ready queue. Next, one of the implementation choices $T_{i,j}$ of T_i is selected and the associated partially reconfigurable regions are reserved. Then, the bitstream corresponding to $T_{i,j}$ is loaded into the reserved PRRs. Then, the execution of T_i occurs on the allocated PRRs. Once the execution is completed, T_i is moved to the completion queue until the PRRs corresponding to T_i are unreserved. The above steps are repeated whenever a task T_i is released for execution.

3.2.3 The Event Set

Based on the state transition diagram discussed in the previous sub-section, let us define the set of events that capture the execution of tasks on a reconfigurable computing platform. Table 3.2 summarizes the list of events associated with a task T_i. The event set Σ_i = $\{a_i, \{\cup_{j=1}^{m_i} rs_{i,j}, us_{i,j}, sd_{i,j}, cd_{i,j}, se_{i,j}, ce_{i,j}\}\}$. The total event set $\Sigma = \cup_{i=1}^{n} \Sigma_i \cup \{t\}$,

Table 3.2: Description of event sets

Event	Description
a_i	Arrival of the current instance of T_i
$rs_{i,j}$	Reserve PRRs associated with $T_{i,j}$; $rs_{i,j} = \{\cup_{k \in \mathcal{I}_{i,j}} rs_{i,j,k}\}$
$us_{i,j}$	Unreserve PRRs associated with $T_{i,j}$; $us_{i,j} = \{\cup_{k \in \mathcal{I}_{i,j}} us_{i,j,k}\}$
$sd_{i,j}$	Start of download of $T_{i,j}$
$cd_{i,j}$	Completion of download of $T_{i,j}$
$se_{i,j}$	Start of execution of $T_{i,j}$
$ce_{i,j}$	Completion of execution of $T_{i,j}$
t	Progress of one time unit

where t is a special event which captures the progress of one time unit. The event set is categorized as follows:

- It may be noted that with respect to a particular implementation $T_{i,j}$ of task T_i, the following events are considered to be *controllable*, i.e., $\Sigma_{con} = \cup_{i=1}^{n} \cup_{j=1}^{m_i} \{rs_{i,j}, sd_{i,j}, se_{i,j}\}$.
- The events such as task arrival, completion of download operation, completion of task execution and, progress of time cannot be prevented by the supervisor. Hence, the set of *uncontrollable* events are: $\Sigma_{unc} = \{\cup_{i=1}^{n}\{a_i\}\} \cup \{\cup_{i=1}^{n} \cup_{j=1}^{m_i} \{us_{i,j}, cd_{i,j}, ce_{i,j}\}\{t\}\}$.

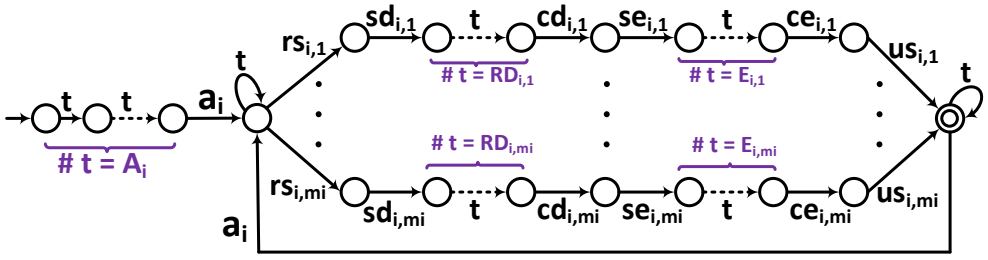


Fig. 3.3: DES model G_i for Periodic task T_i

3.2.4 Task Execution Model

The DES model G_i for the execution of task T_i on the reconfigurable platform K is depicted in Fig.3.3. From Fig. 3.3, it may be observed that the arrival of task $T_i's$ first

instance a_i occurs at the A_i^{th} tick from the system's start time. Subsequently, T_i will be kept in the ready queue (modelled using a self-loop on event t) or an implementation of T_i will be selected immediately. An implementation is captured by m_i outgoing branches on events $\{rs_{i,1}, rs_{i,2}, \ldots, rs_{i,m_i}\}$.

Let us assume that the implementation $\mathcal{I}_{i,1}$ is selected for the current instance of T_i. The corresponding operation is represented by $rs_{i,1}$. Next, the bitstream corresponding to implementation $\mathcal{I}_{i,1}$ will be downloaded (which is captured by $sd_{i,1}$) and it consumes $RD_{i,1}$ tick events. On the occurrence of completion of the download operation $cd_{i,1}$, task T_i starts its execution (captured by $se_{i,1}$) and consumes $E_{i,1}$ ticks. Subsequent to the completion of execution, T_i moves to the completion queue which is captured by the transition on $ce_{i,1}$. Finally, the PRR regions associated with $T_{i,1}$ will be unreserved (captured by the event $us_{i,1}$) and this leads to the completion of execution of the current instance of T_i. Next, the model G_i waits (captured by the self-loop on t) for the arrival of the next instance of task T_i. The above steps are repeated, when the next instance of T_i arrived for execution.

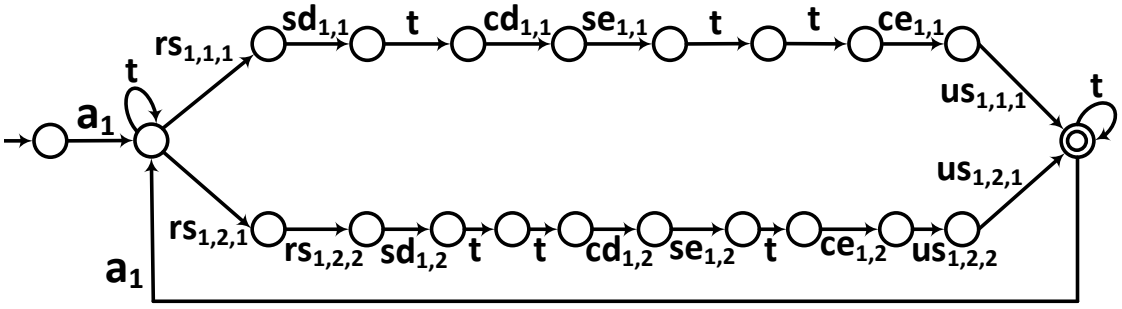


Fig. 3.4: DES model G_1 for task T_1

***Example(continued)*:** Figure 3.4 shows the DES model G_1 for task T_1. Similarly, the DES models G_2 for task T_2, G_3 for task T_3 and G_4 for task T_4, respectively, can be constructed.

3.2.5 Composite Task Execution Model

It may be inferred that the marked behavior $L_m(G_i)$ corresponding to the execution of task T_i satisfies distinct execution times of T_i for a given implementation $T_{i,j}$. Given n individual models $G_1, G_2, \ldots, G_n$ corresponding to $T_1, T_2, \ldots, T_n$, the synchronous product $G = G_1 \parallel \ldots \parallel G_n$ on the models gives us the composite model of the tasks executing *concurrently*. As individual models do not share any common event except *tick*, all models

synchronize only on the *tick* event. Since individual models satisfy distinct execution times of T_i on a given implementation $T_{i,j}$, $L_m(G)$ also satisfies them. However, the sequences in $L_m(G)$ *may violate resource (i.e., ICAP and PRR) and timing (i.e., deadline and fixed-inter arrival time) constraints.* In the following sub-sections, we develop models to capture these constraints.

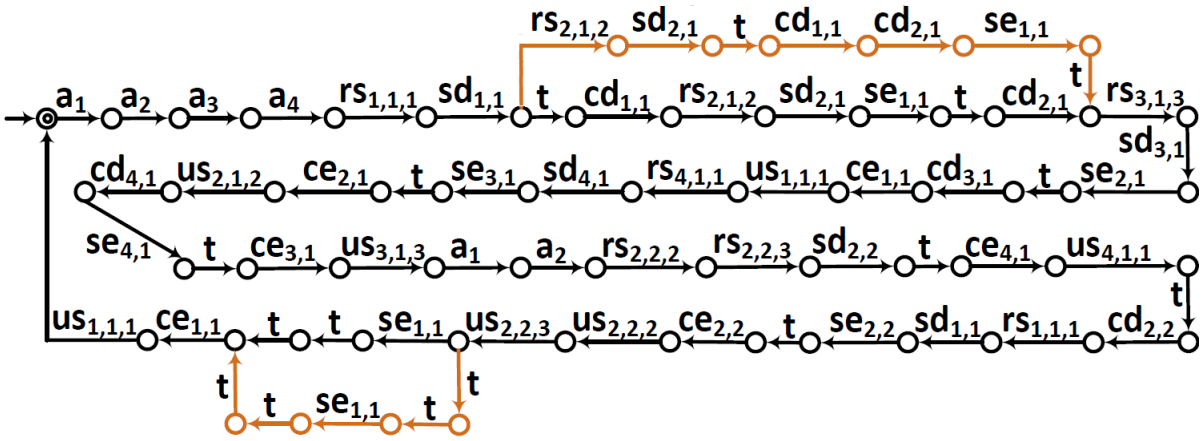


Fig. 3.5: Example: Composite Task Execution Model (partial diagram)

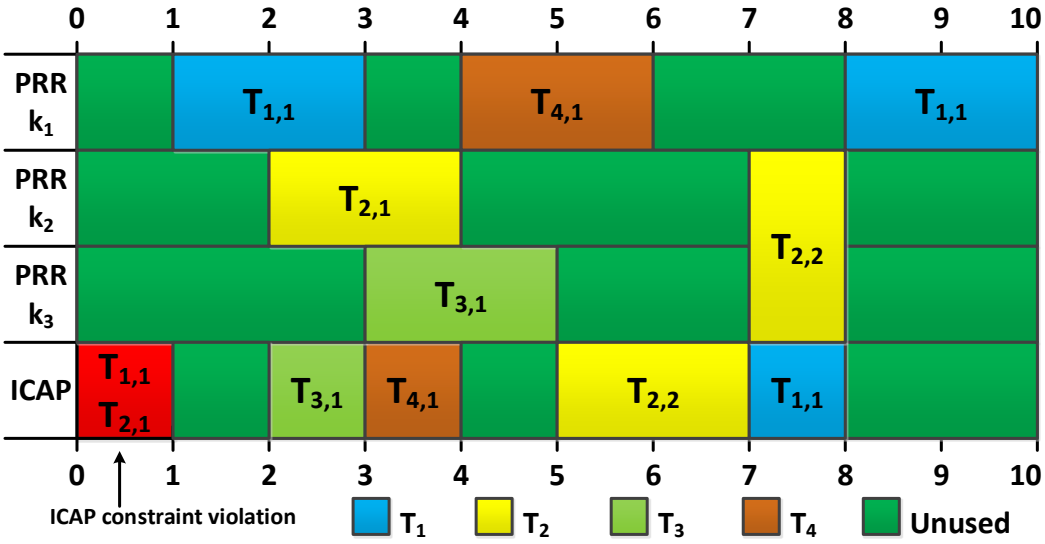


Fig. 3.6: Example: Gantt chart representation of seq_1

Example(continued) **:** Fig. 3.5 shows the (partial) composite task execution model G $(= G_1 \parallel G_2 \parallel G_3 \parallel G_4)$. To illustrate that G contains both resource and timing constraint satisfying and violating sequences, let us consider the sequences seq_1, seq_2, $seq_3 \in L_m(G)$:

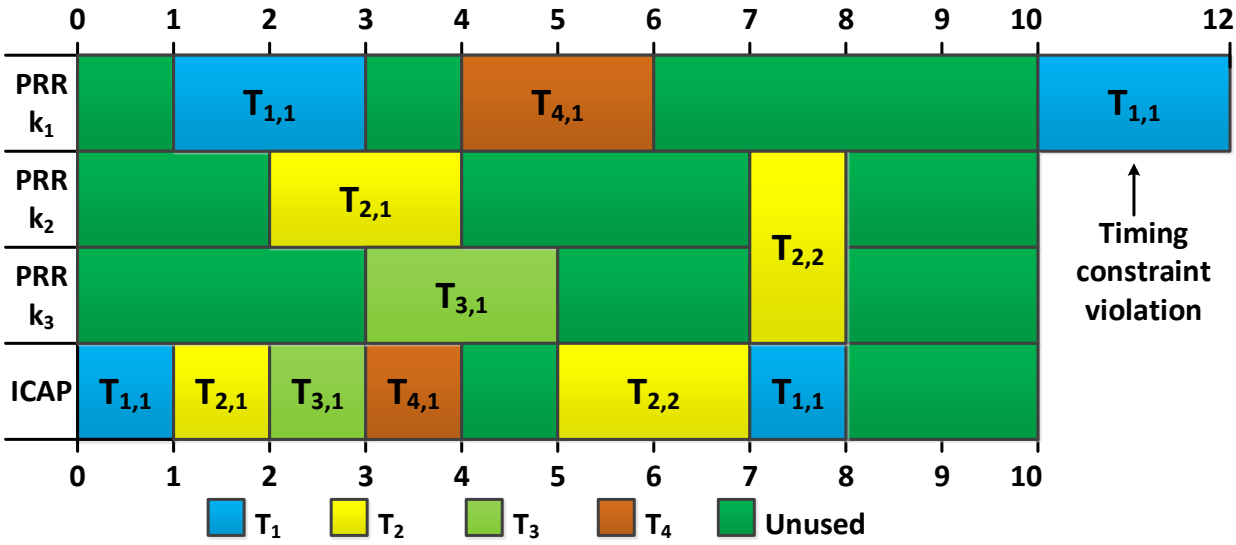


Fig. 3.7: Example: Gantt chart representation of seq_2

- $seq_1 = a_1a_2a_3a_4rs_{1,1,1}sd_{1,1}trs_{2,1,2}sd_{2,1}tcd_{1,1}cd_{2,1}se_{1,1}t\ rs_{3,1,3}sd_{3,1}se_{2,1}tcd_{3,1}ce_{1,1}us_{1,1,1}$ $rs_{4,1,1}sd_{4,1}se_{3,1}tce_{2,1}\ us_{2,1,2}\ cd_{4,1}se_{4,1}tce_{3,1}us_{3,1,3}a_1a_2rs_{2,2,2}rs_{2,2,3}sd_{2,2}tce_{4,1}\ us_{4,1,1}t\ cd_{2,2}$ $rs_{1,1,1}sd_{1,1}se_{2,2}tce_{2,2}us_{2,2,2}us_{2,2,3}se_{1,1}\ ttce_{1,1}us_{1,1,1}$

- $seq_2 = a_1a_2a_3a_4rs_{1,1,1}sd_{1,1}tcd_{1,1}rs_{2,1,2}sd_{2,1}se_{1,1}tcd_{2,1}\ rs_{3,1,3}\ sd_{3,1}se_{2,1}tcd_{3,1}ce_{1,1}us_{1,1,1}$ $rs_{4,1,1}sd_{4,1}se_{3,1}tce_{2,1}\ us_{2,1,2}\ cd_{4,1}se_{4,1}tce_{3,1}us_{3,1,3}a_1a_2rs_{2,2,2}rs_{2,2,3}sd_{2,2}tce_{4,1}\ us_{4,1,1}\ tcd_{2,2}$ $rs_{1,1,1}sd_{1,1}se_{2,2}tce_{2,2}us_{2,2,2}us_{2,2,3}ttse_{1,1}tt\ ce_{1,1}us_{1,1,1}$

- $seq_3 = a_1a_2a_3a_4rs_{1,1,1}sd_{1,1}tcd_{1,1}rs_{2,1,2}sd_{2,1}se_{1,1}tcd_{2,1}\ rs_{3,1,3}sd_{3,1}se_{2,1}tcd_{3,1}ce_{1,1}us_{1,1,1}$ $rs_{4,1,1}sd_{4,1}se_{3,1}tce_{2,1}\ us_{2,1,2}cd_{4,1}se_{4,1}tce_{3,1}us_{3,1,3}a_1a_2rs_{2,2,2}rs_{2,2,3}sd_{2,2}t\ ce_{4,1}\ us_{4,1,1}\ tcd_{2,2}$ $rs_{1,1,1}sd_{1,1}se_{2,2}tce_{2,2}us_{2,2,2}us_{2,2,3}\ se_{1,1}tt\ ce_{1,1}us_{1,1,1}$

The gantt chart representation of seq_1 is shown in Fig. 3.6. It may be seen that seq_1 respects the timing constraints for all tasks. For example, the implementation $T_{1,1}$ is selected for task T_1 and it is downloaded via ICAP at the first time slot (time slot: 0). Subsequent to the download of the bitstream, $T_{1,1}$ gets executed on PRR k_1 for two units of time (time slots 1 and 2). Further, it can be observed that seq_1 violates the ICAP constraint. That is, only one bitstream can be downloaded at a time using ICAP. However, both $T_{1,1}$ and $T_{2,1}$ are downloaded simultaneously via ICAP at time slot 0. The gantt chart representation of seq_2 is shown in Fig. 3.7. Here, seq_2 does not satisfy the timing constraint associated with T_1. For example, the deadline/period gets violated when its second instance gets executed. In particular, the second instance of T_1 arrives

at time instant 5 and its execution needs to be completed within its relative deadline of 10. However, the execution of T_1 gets completed at 12 which leads to the violation of deadline. On the other hand, the sequence seq_3 respects both the resource as well as timing constraints for all tasks (shown in Fig. 3.16).

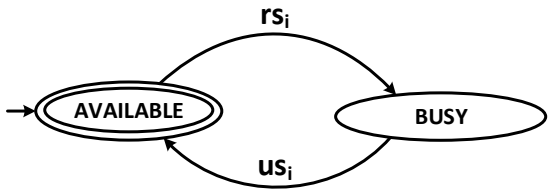


Fig. 3.8: TDES Model for PRR/Slot k_i

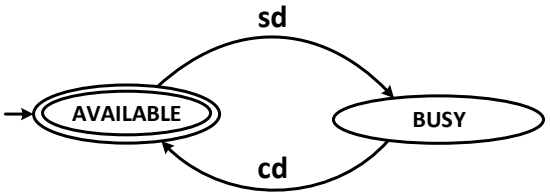


Fig. 3.9: TDES Model for ICAP

3.2.6 Resource-Constraint Model

In this sub-section, we develop the models to capture the resource constraint specifications for PRRs and ICAP. Specifically, we develop DES models to capture the following constraints: (i) Once a task T_i is allocated to a PRR/slot k_λ, it remains allocated until its completion. Meanwhile no other task $T_j (\neq T_i)$ is allowed to start execution on slot k_λ. (ii) A reconfiguration port ICAP must be exclusively used by a single task when performing bitstream download operation.

The resource constraint associated with the PRR/slot k_i is captured by the DES model SC_k and its transition structure is shown in Figure 3.8. It can be seen that PRR is initially available for use. Once any task $T_i \in T$ reserves PRR k_i, then SC_k transits from $AVAILABLE$ to $BUSY$ state. This implies that PRR k_i is currently being used by T_i and it becomes unavailable for the other tasks in T. Once T_i unreserves PRR k_i, the model SC_k transits back to $AVAILABLE$ state. Now, PRR k_i is availble for execution for any task in T. Similarly, the DES model for $ICAP$ is shown in Figure 3.9.

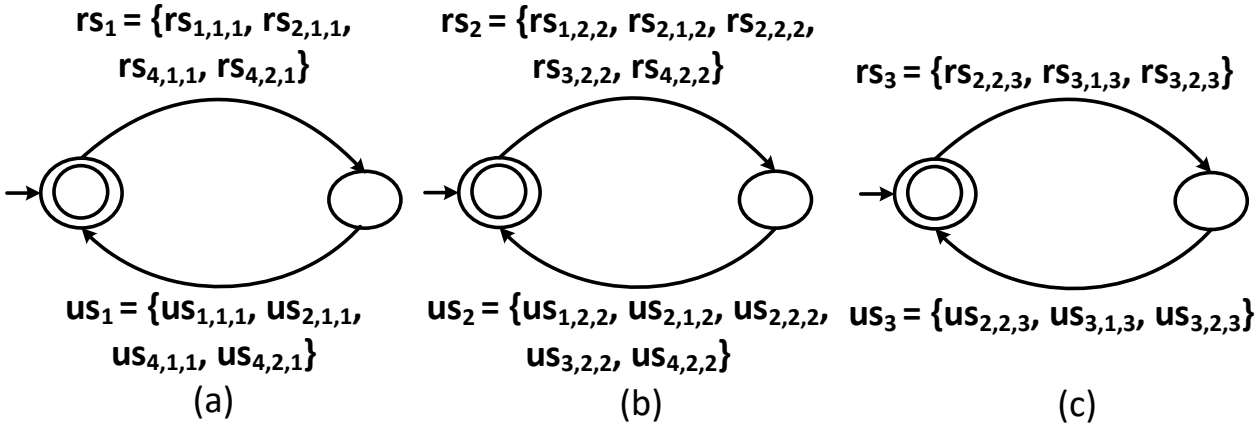


Fig. 3.10: Example: TDES Model for PRR/Slot (a) k_1, (b) k_2, (c) k_3

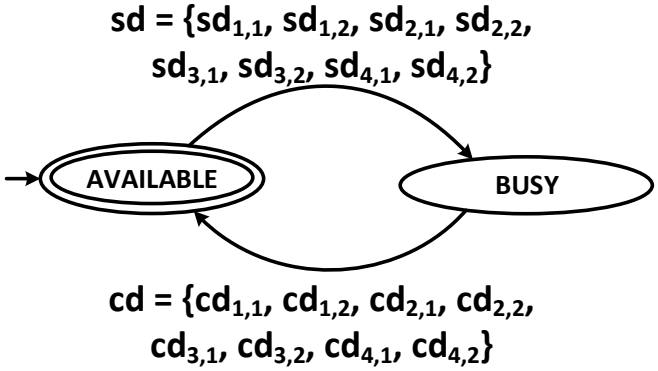


Fig. 3.11: Example: TDES Model for ICAP

Example (continued) **:** Figures 3.10a, 3.10b and 3.10c show the DES models SC_{k_1} for PRR/slot k_1, SC_{k_2} for PRR/slot k_2 and SC_{k_3} for PRR/slot k_3, respectively. Figure 3.11 illustrates the DES model of the ICAP.

3.2.7 Composite Resource-constraint Model

Given the q DES models for PRR/slot constraints $SC_{k_1}, SC_{k_2}, \ldots, \ldots SC_{k_q}$ which corresponds to the available slots in the given problem, we can compute the composite slot-constraint model SC as: $SC_{k_1} \parallel SC_{k_2} \parallel \ldots \parallel SC_{k_q}$. Hence, $L_m(SC)$ represents the language that disallows the concurrent execution of multiple tasks on the same PRR/slots. Similarly, $L_m(ICAP)$ contains sequences that allow only one task to make use of the ICAP port at any given time. Now we construct the composite resource-constraint model RC $= SC||ICAP$. Although, all sequences in $L_m(RC)$ satisfy the slot and reconfiguration port constraints, sequences in $L_m(RC)$ may not correctly capture timing properties such

as execution and deadlines associated with the tasks.

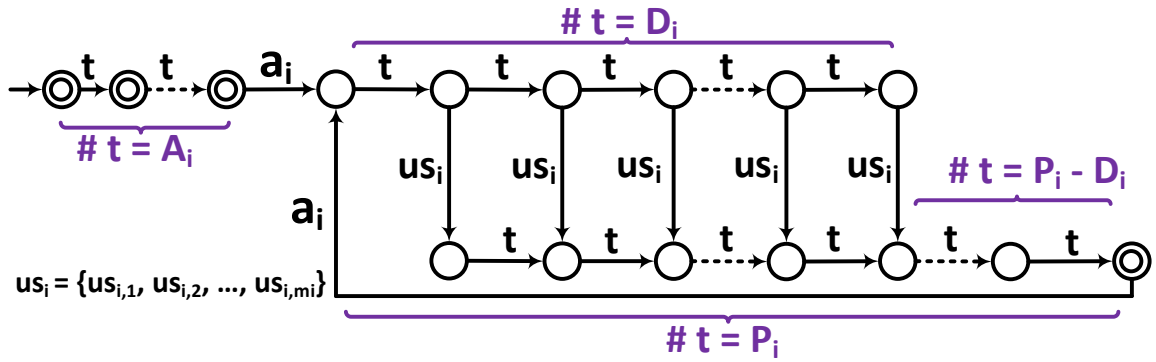


Fig. 3.12: TDES Model H_i for deadline and period of T_i

3.2.8 Timing Constraint Model

In the previous sub-sections, we have introduced and discussed models for resource-constraints associated with the PRR/slot and ICAP port. In this sub-section, we introduce a DES model to capture the timing constraints associated with each task $T_i \in T$. Figure 3.12 shows the DES model H_i that captures the deadline and period associated with task T_i. It can be seen that once T_i arrives after A_i ticks from system start, T_i has to complete its execution within its relative deadline D_i. Such a constraint is modelled by the transitions on the event us_i up to D_i ticks from the arrival of the current instance of T_i. After the completion of execution of the current instance of T_i, the next instance of T_i must arrive after $P_i - D_i$ ticks to ensure the fixed inter-arrival time between any two consecutive instances of the periodic task $T_i \in T$.

Example(continued) **:** Figures 3.13a and 3.13b show the DES models H_1 for T_1 and H_2 for T_2. Similarly, Figures 3.14a and 3.14b capture the DES models H_3 and H_4 for T_3 and T_4, respectively.

3.2.9 Composite Timing Constraint Model

The marked behavior $L_m(H_i)$ *contains all sequences that satisfy the deadline and period constraints associated with* T_i. Similarly, we can construct models for other tasks in the task set T. Given H_1, H_2, ..., H_n corresponding to T_1, T_2, ..., T_n, we can obtain the composite model $H = H_1||H_2||\ldots||H_n$. The marked behavior $L_m(H)$ *captures the timing constraints associated with all the tasks in* T.

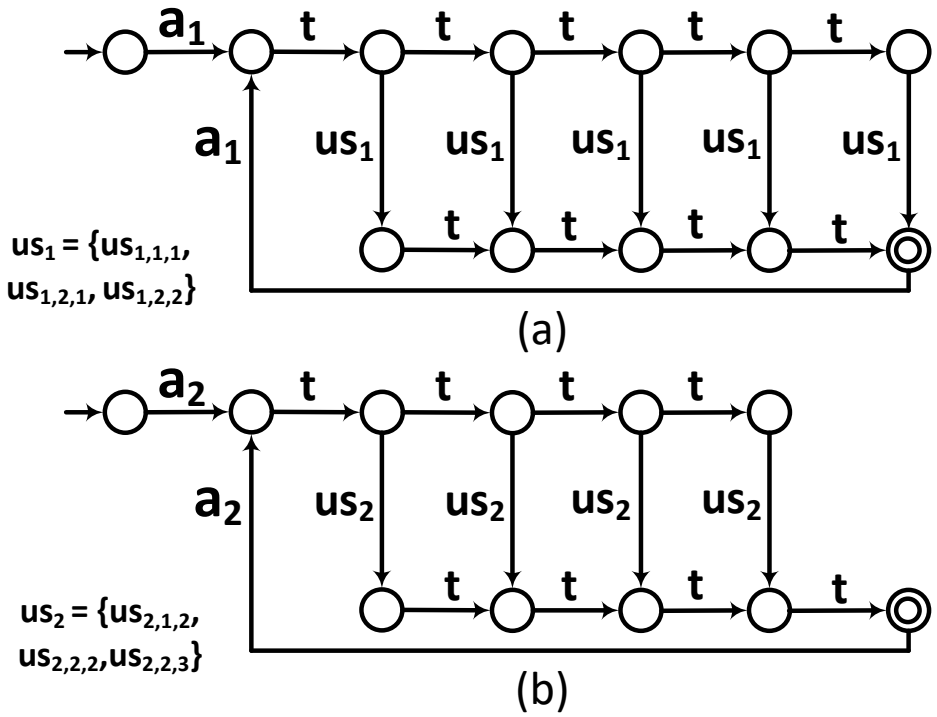


Fig. 3.13: Example: TDES Models (a) H_1, (b) H_2

3.2.10 Supervisor Synthesis

To find only and all sequences in $L_m(G)$ that satisfy the resource as well as timing constraints, we compute the supervisor as follows: $S = G||RC||H$. The resulting marked behavior $L_m(S)$ $(= L_m(G) \cap L_m(RC) \cap L_m(H))$ contains all feasible scheduling sequences that satisfy the system specifications considered in this work. Hence, we can use any sequence in $L_m(S)$ to construct the supervisor which can be used to govern task execution on a given FPGA. It may be noted that sequences in $L(S)$ which violate resource and/or timing constraints are not part of $L_m(S)$ and hence, such sequences lead to *deadlock* states in S. This implies $L(S) \neq L_m(S)$, i.e., S is *blocking*. However, to guarantee that all instances of all tasks meet their resource and timing constraints, during online execution, S must be made controllable with respect to the system specifications (RC and H). This is achieved by determining the controllable and non-blocking part of $L_m(S)$ using *Safe State Synthesis* [112].

It is possible to further appreciate that through a systematic search over the scheduling language $L_m(S)$, it is possible to filter-out those scheduling sequences which minimize migration related overheads. For example, given $L_m(S)$, the proposed framework can be extended so that feasible schedule(s) which performs best with respect to one or more

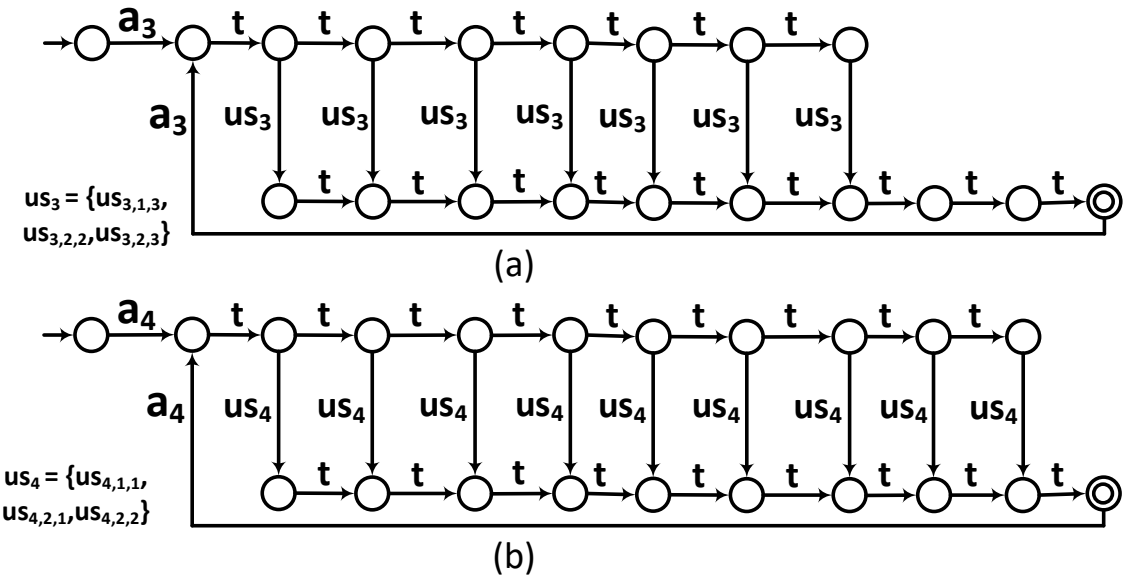


Fig. 3.14: Example: TDES Models (a) H_3, (b) H_4

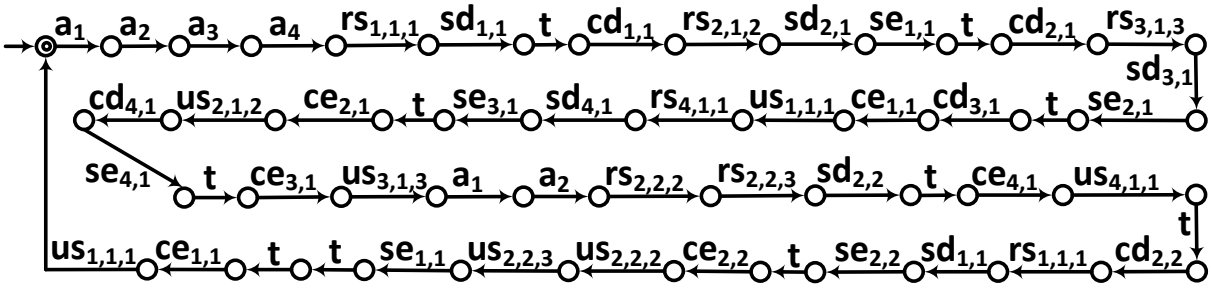


Fig. 3.15: Example: Supervisor (partial diagram)

metrics like resource usage, power consumption, fault-tolerance etc can be determined.

***Example(continued)*:** The (partial) diagram of the obtained supervisor is shown in Fig. 3.15. It can be seen that seq_3 satisfies all system specifications and it belongs to $L_m(S)$. Further, the gantt chart representation of seq_3 (shown in Fig. 3.16) also illustrates that there are no resource and timing constraint violations.

3.2.11 Complexity Analysis

A schematic diagram representing the overall flow of the proposed scheduler framework has been summurized in Figure 1.1. We now present a step-wise discussion on the complexity of the proposed synthesis scheme.

1. The state space complexity of G_i (shown in figure 3.3) is computed as follows:

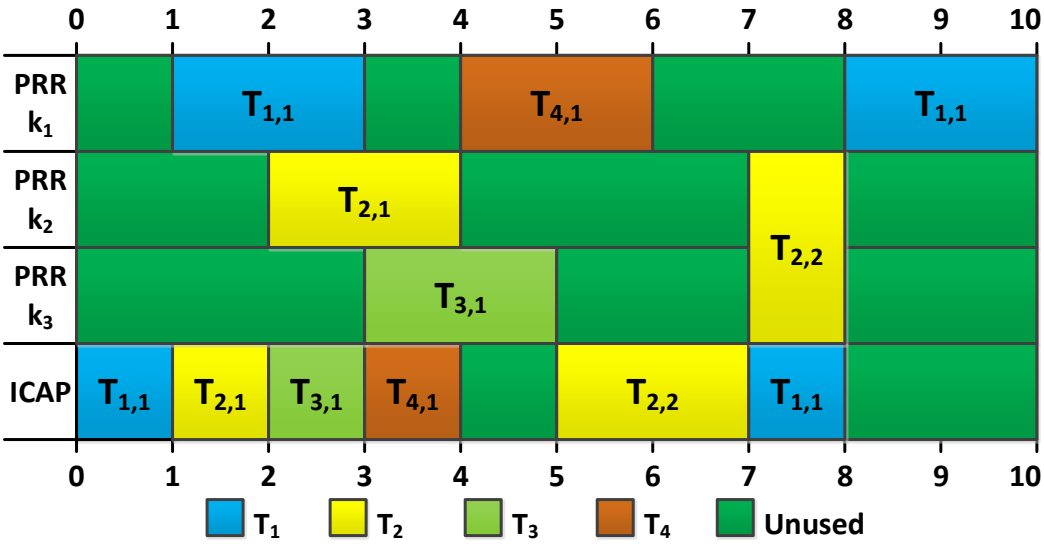


Fig. 3.16: Example: Gantt chart of sequence seq_3

There are A_i states to measure the arrival time of T_i. After the occurrence of the arrival event a_i, the DES G_i has m branches emanating from it based on the events $\{ rs_{i,1}, rs_{i,2}, \ldots, rs_{i,m_i}\}$ representing the start of task T_i's execution on anyone of the m PRRs/tiles. With the reconfiguration time RD_i and execution time E_i, each of the branches contain $(RD_i + E_i)$ states due to transition on tick events. Therefore, the state-space complexity of G_i becomes $\mathcal{O}(A_i + m(RD_i + E_i))$.

2. The state space complexity of the resource-constraint model SC_{k_i} (shown in figure 3.8) is $\mathcal{O}(q)$ because it contains distinct states connected to the initial state to represent the execution of each task on PRR/slot-k_i.

3. The state space complexity of H_i (shown in figure 3.12) is $\mathcal{O}(A_i+P_i)$ because distinct states are used to count the occurrence of each tick starting from the arrival to its period P_i.

4. The state space complexity of model ICAP (shown in figure 3.9) is a constant.

5. Given n individual DESs $\{G_1, G_2, \ldots, G_n\}$ corresponding to $\{T_1, T_2, \ldots, T_n\}$, an upper bound for the number of states in the composite task execution model G is $\prod_{i=1}^{n} |Q^{G_i}|$, where $|Q^{G_i}|(= \mathcal{O}(A_i + m(RD_i + E_i))$ is the total number of states in G_i. Similarly, the total number of states in the composite resource-constraint model RC is $\prod_{i=1}^{q} |Q^{RC_i}|$, where $|Q^{RC_i}|$ is the total number of states in RC_i. In the same

token, an upper bound for the number of states in the composite timing-constraints model H is $\prod_{i=1}^{n} |Q^{H_i}|$, where $|Q^{H_i}|$ is the total number of states in H_i.

Let $S_0 (= G \parallel SC \parallel H)$ and S eliminate the sequence in S_0 that may possibly terminate in a deadlock. The total number of states in S_0 becomes $\mathcal{O}(|G| \times |RC| \times |H|)$. The time-complexity for computing G, RC, H and S_0 are exponential. However, the computation of $supC(L_m(S))$ are polynomial time as it involves simple graph traversal [113]. It may be observed that the number of states in the Composite models G, RC, and H grows exponentially as the number of tasks and executional PRRs/slots increases.

3.3 Supervisor Implementation

In this section, we present the procedures for the real-world implementation of the supervisor/scheduler synthesized using our scheme on FPGAs with DPR support.

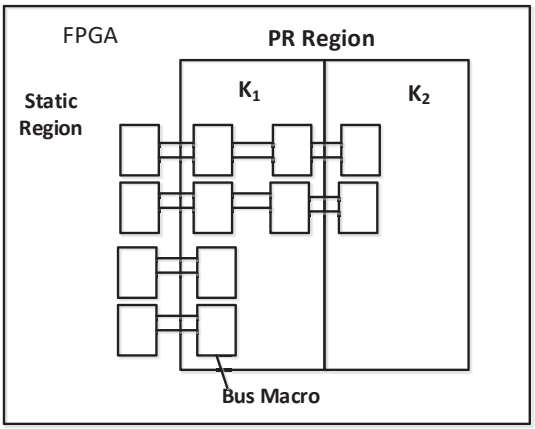


Fig. 3.17: Spartan FPGA with Static and PR Regions (k_1, k_2)

3.3.1 Experimental Setup

We used the Atlys development board to test and verify our synthesized scheduler. The Atlys development board is built around Spartan-6 LX45 FPGA in a 324-pin package, with a 100MHZ system clock. The platform also includes several input/output interfaces such as switches, buttons, and LEDs. For memory, the board provides 1 Gbit DDR2 memory and 128 Mbit of flash memory. Communication can be done with a com-port on the computer through a USB-UART bridge on the board. The Spartan-6 also includes

of an ICAP primitive that gives access to the FPGA configuration memory through the configuration registers.

We considered two independent tasks ($\{T_1, T_2\}$) for implementation. A 16-bit unsigned *ripple carry adder* (RCA; denoted by T_1) and a 4-bit binary (unsigned) sequential multiplier using the *shift-and-add* algorithm (SHA; denoted by T_2). Two VHDL implementations of RCA have been considered: non-pipelined (NRCA) and a two-stage pipeline (PRCA). The NRCA implementation completes the addition of two 16-bit numbers in every 2 clock cycles whereas the PRCA generates an output every clock cycle. Similarly, two VHDL implementations of the SHA algorithm have been prepared. SHA1 was coded in such a way that it has a single process with 8 states and it requires 8 clock cycles to complete one multiplication. SHA2 is coded using two processes and 4 states, as a result, it requires 4 clock cycles to multiply two 4-bit binary numbers. Thus, for task RCA, implementation $T_{1,1}$ refers to NRCA and implementation $T_{1,2}$ denotes PRCA. Similarly, implementation $T_{2,1}$ and implementation $T_{2,2}$ denote SHA1 and SHA2 respectively.

Figure 3.17 shows a representative architectural view of the implementation done using Spartan FPGA board with DPR support. The block diagram in Figure 3.17 shows the presence of (i) static region, (ii) two partially reconfigurable regions (PRRs k_1 and k_2), (iii) a set of bus macros across the boundaries of different regions.

All the PRRs obtain hard-coded input data from the static region. Similarly, computed results from PRRs are displayed to the outside world through the static region. The UART module and the LEDs on the Atlys board are used for computational verification. The UART module sends the output received from PRR k_1 to the com-port of the computer. The LEDs display the result of computation from PRR k_2.

Table 3.3: Reconfiguration and Execution times (in clock cycles)

Task Impl	**Rec.clks**	**Exe.clks**	**Diff.bitstream size**
NRCA ($T_{1,1}$)	$46,025$	$100,001$	$14KB$
PRCA ($T_{1,2}$)	$55,887$	$50,001$	$17KB$
SHA1 ($T_{2,1}$)	$53,440$	$80,001$	$16KB$
SHA2 ($T_{2,2}$)	$88,761$	$40,001$	$27KB$

In order to construct the DES models, we have profiled both implementations of tasks T_1 and T_2 by running them standalone on the Spartan-6 LX45 FPGA, and obtained information regarding the number of clock cycles required for their reconfiguration operation

and execution. The results are captured in Table 3.3. For this implementation, we have considered 50, 000 ticks of the internal clock to be the length of one time unit. Along with the tick generator, a tick-counter module is also used to count and generate an interrupt signal when a particular number of ticks have elapsed. Therefore, all event generations ($rs_{i,j}$, $sd_{i,j}\ldots$, etc) are sequenced by our tick-counter. The clock cycles that we obtained for the reconfiguration and execution process are converted into our defined tick as shown in Table-3.4.

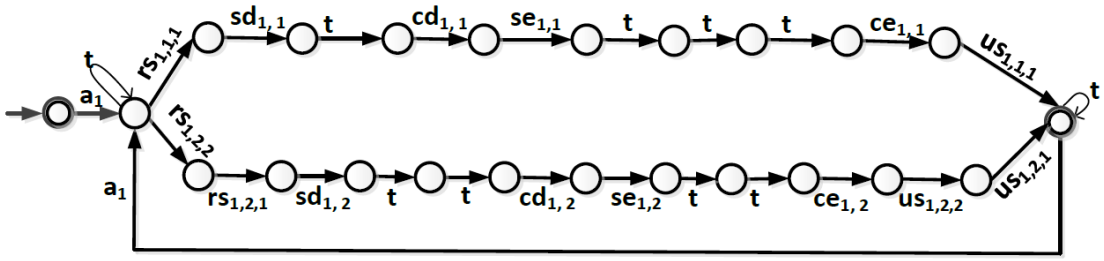
Table 3.4: Task Characteristics

	Implementation (PRRs, Reconfig Download Time, WCET)		
Task	$\{\mathcal{I}_{i,1},\ RD_{i,1},\ E_{i,1}\}$	$\{\mathcal{I}_{i,2},\ RD_{i,2},\ E_{i,2}\}$	$\langle A_i, D_i, P_i\rangle$
RCA	$\{\{k_1\}, 1, 3\}$	$\{\{k_1, k_2\}, 2, 2\}$	$\langle 0, 7, 7\rangle$
SHA	$\{\{k_2\}, 2, 2\}$	$\{\{k_1, k_2\}, 2, 1\}$	$\langle 0, 7, 7\rangle$

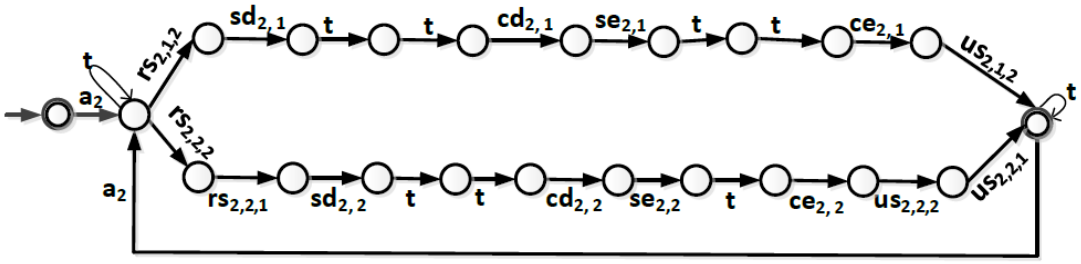
During the implementation of the whole system, two configuration files are produced by the final project: the static and the partial configuration files. The static configuration file (top.bit) consists of the static logic and an empty PR design. When all the necessary bitfile generations are completed, the top.bit will be the first bitstream to be downloaded to the board. Using the GOAHEAD, partial netlists have been obtained for each module implementation. Merging individual netlists with the static logic generates a static design with a PR module (merged.ncd). We use BitGen -r switch to create a differential bitstream from the differences between merged.ncd and top.bit. The differential bitstreams generated this way are our partial reconfiguration files (i.e., partial.bit) and stored on the flash memory of the Atlys board.

3.3.2 Supervisor Generation

We apply the modeling concept described in section 3.2 to capture the task implementation characteristics shown in table 3.4. The TDES model G_1 for RCA and G_2 for SHA are shown in fig 3.18a and 3.18b, respectively. Similarly the TDES model for the resource constraints; the reconfiguration port and PRRs/slots k_1 and k_2 are shown in fig 3.19a, 3.19b, and 3.20 respectively. The TDES models H_1 and H_2 for the period and deadline of T_1 and T_2 are shown in fig. 3.21a and 3.21b respectively. Next, we computed the supervisor $S = G \parallel RC \parallel H$. For the purpose of implementation, we consider the following sequence (say, seq_4) that belongs to $L_m(S)$: $seq_4 =$

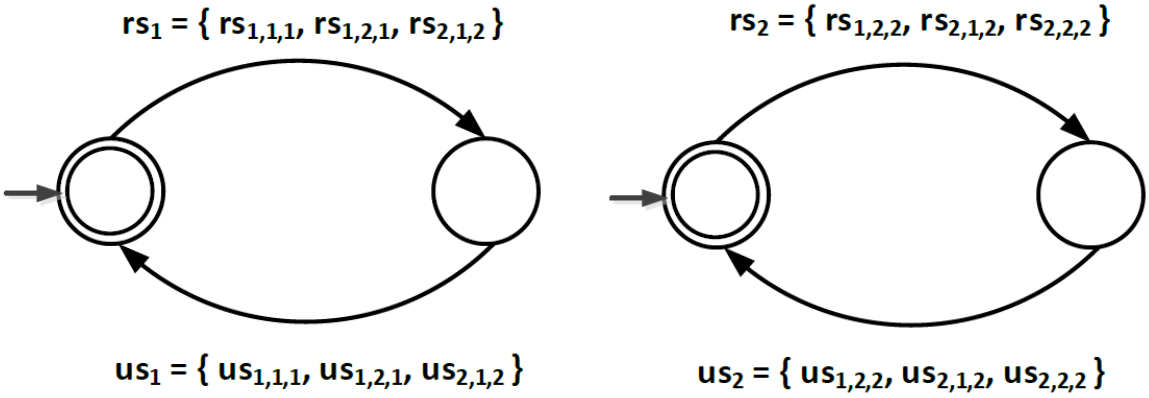


(a) G_1 for T_1



(b) G_2 for T_2

Fig. 3.18: TDES Models for RCA (i.e.,T_1) and SHA(i.e., T_2)



(a) SC_k_1 for K_1

(b) SC_k_2 for K_2

Fig. 3.19: TDES Models for PRRs/slots

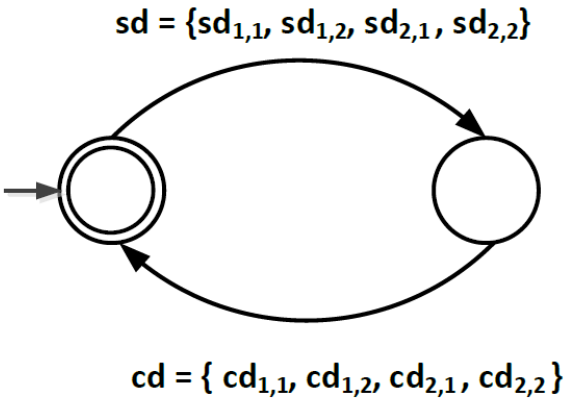
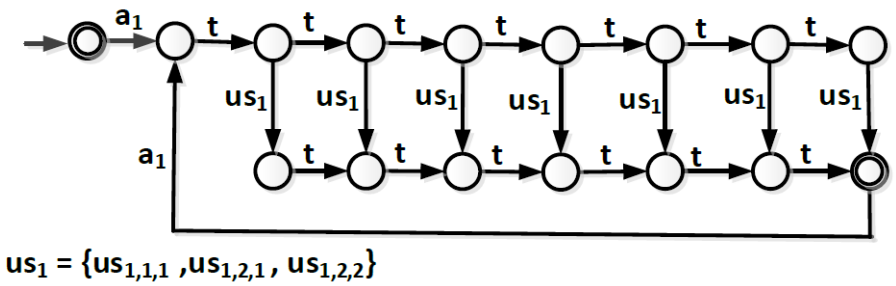
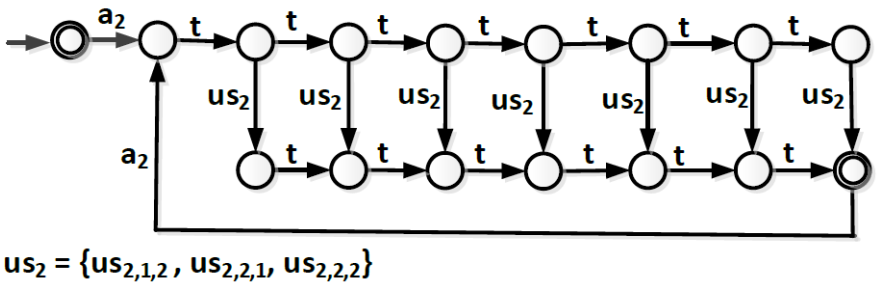


Fig. 3.20: TDES for ICAP



(a) TDES model H_1 for T_1



(b) TDES model H_2 for T_2

Fig. 3.21: TDES Model for deadline and period of T_1 and T_2

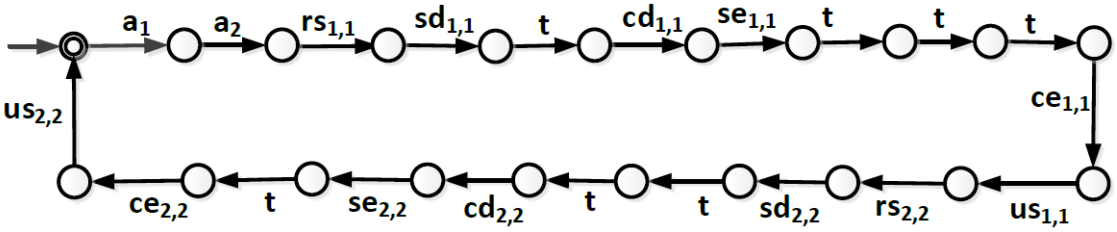


Fig. 3.22: supC($L_m(S_0)$)

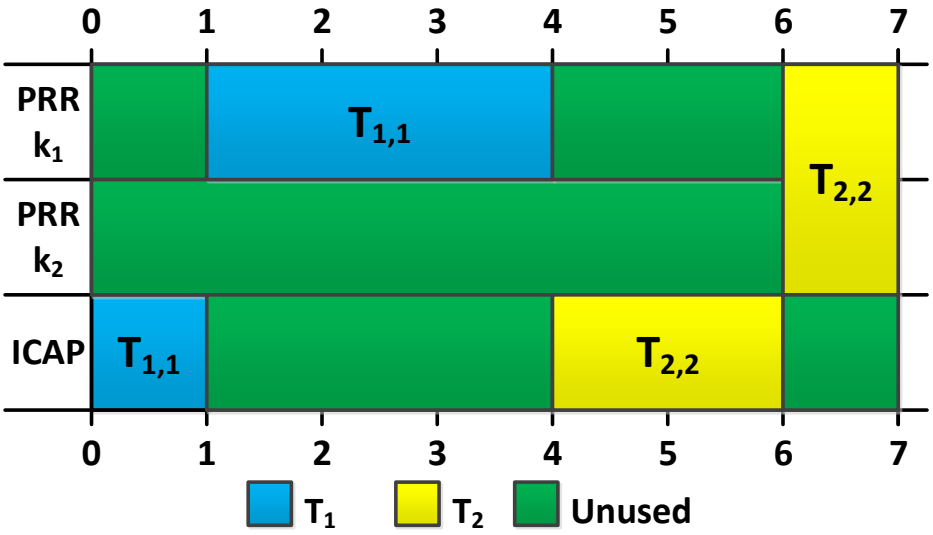


Fig. 3.23: Gantt chart of sequence $seq_4 \in L_m(S)$

$a_1 a_2 rs_{1,1} sd_{1,1} tcd_{1,1} se_{1,1} tttce_{1,1} us_{1,1} rs_{2,2} sd_{2,2} ttcd_{2,2} se_{2,2} t\ ce_{2,2} us_{2,2}$. The gantt chart representation of this sequence is shown in Figure 3.23. It can be seen that the implementation $\mathcal{I}_{1,1}$ is selected for T_1 and gets executed on PRR k_1. For task T_2, the implementation $\mathcal{I}_{2,2}$ is selected and it uses both the PRRs simultaneously for its execution. Finally, we have implemented a VHDL code corresponding to the selected sequence from the synthesized supervisor.

3.3.3 Task Management

The Supervisor is responsible for the timely module downloading, execution start and completion taking into consideration all the constraints placed upon it. Fig 3.24 illustrates the system implementation structure of our work. The structure consists of *ScheduleFSM module*, *Reconfiguration-controller module*, *Tick_counter & Tick_generator module*, *UART module*, PR Regions, SPI flash, LED, and button. The *ScheduleFSM* is the top module that encodes the synthesized supervisor discussed in section 3.3.2, it initializes the start address and bitstream size of the partial modules and is responsible to send the start/stop execution signals to other modules. The *Reconfiguration-controller module* requires the start-address and size information of the differential bitstream along with the "start_downloading" signal to function properly. The *Tick_counter & Tick_generator module* expects the "start_tick" signal from ScheduleFSM to generate sequence of tick_counter.

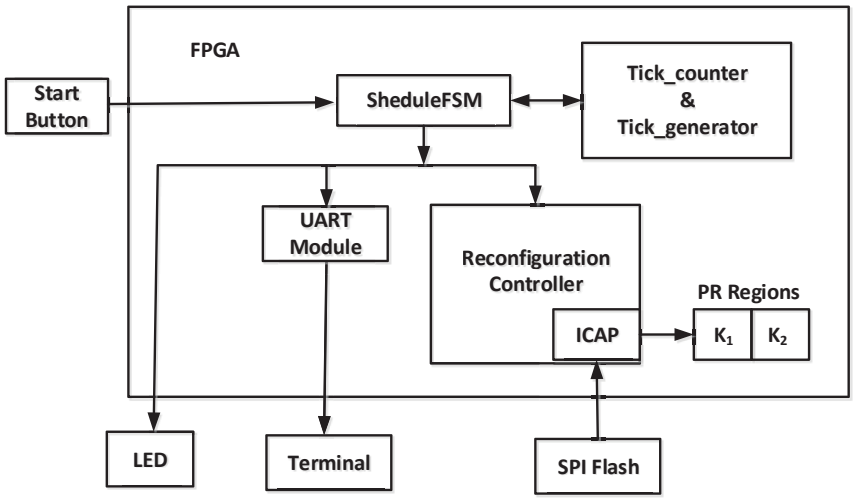
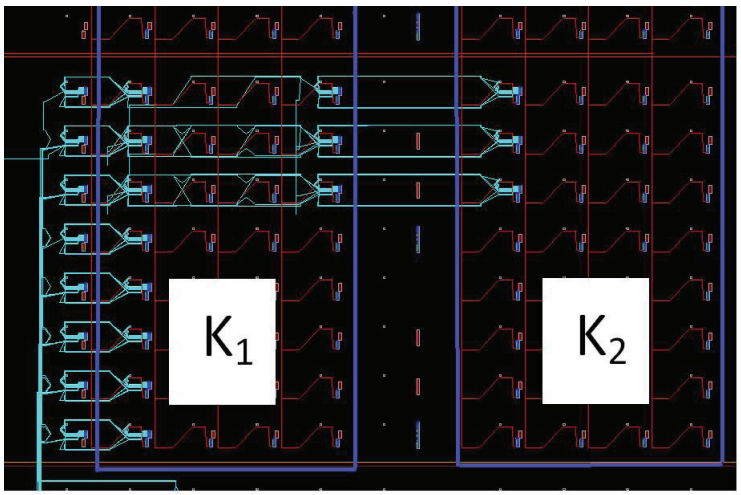


Fig. 3.24: System Implementation Structure

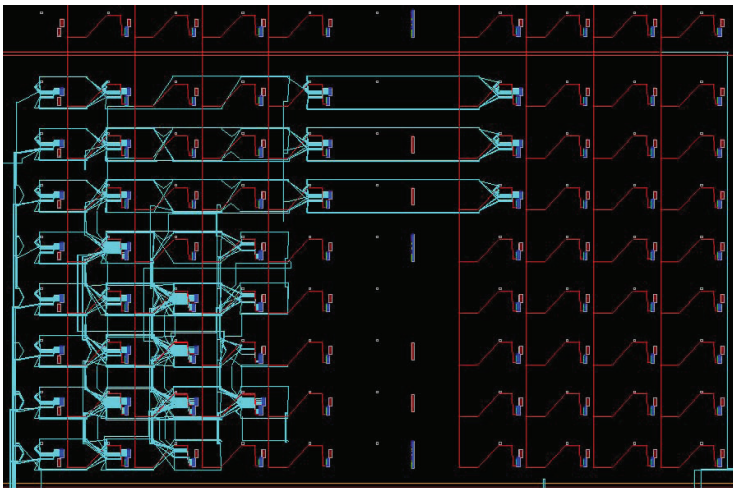
Pressing the GPIO button initializes the *SheduleFSM* module, which inturn initialize the various module's data and control signals. The "start_tick" is the first signal that

is set by the *SheduleFSM* module to enable the *Tick_counter & Tick_generator* module. This module is responsible for generating a tick event that is used by the *SheduleFSM* module to monitor event generation.

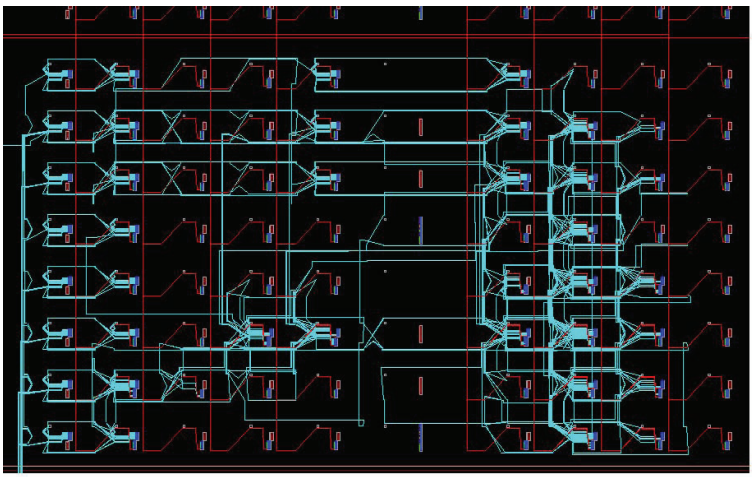
For example, referring figure 3.22, the final synthesized supervisor, at tick_count = 0, the *SheduleFSM* set the event $sd_{1,1}$ (start downloading $T_{1,1}$) and for the *Reconfiguration-controller* module. The *Reconfiguration-controller* module then selects the appropriate bitstream from the SPI flash and send it to the ICAP to reconfigure the associated partial Region. At tick_count = 1, events $cd_{1,1}$ (complete downloading $T_{1,1}$) and $se_{1,1}$ (start executing $T_{1,2}$) are asserted, so that the placed module starts executing. At the 4^{th} tick_count, events $ce_{1,1}$ and $us_{1,1}$ are asserted, indicating that $T_{1,1}$ complete execution and unreserve the corresponding slot. Next, task $T_{2,2}$ starts reserving the slots and continue the process.



(a) Empty PRRs at time slot 0



(b) NRCA placed on k_1 at time slot 1



(c) SHA2 placed on both PRRs at time slot 6

Fig. 3.25: Snap shot from FPGA editor for task Placements on PRRs

3.3.4 Results

The proposed system model was verified experimentally. Figure 3.25 shows the snap shot taken from the FPGA editor for task placements at different time interval. Figure 3.25a illustrates the configuration of PRRs at time instant 0, where both the PRRs are empty. This figure clearly shows the location of the bus macros at the boundary of static and partially reconfiguration regions k_1 and k_2. Figure 3.25b depicts the configuration of the FPGA at time instant 1 when NRCA ($T_{1,1}$) has been placed on PRR k_1. From this figure, we can observe that the bus macros which are used for data or control signal communications are wired with the NRCA logic. Many of the bus macros of k_1 (located at the bottom left) are seen wired with the placed logic. This figure also confirms that NRCA uses the resources that are available only in PRR k_1. The configuration of FPGA at time instant 6 is depicted in Fig. 3.25c when SHA2 ($T_{2,2}$) logic has been placed in the combined area of PRRs k_1 and k_2. As seen from the figure, the SHA2 logic consumes a large part of the resource available in PRR k_2 and the remaining small logics are placed on PRR k_1.

3.4 Summary

In this chapter, we presented an SCDES based offline scheduler synthesis approach for non-preemptive periodic real-time tasks on FPGAs with DPR support. The synthesis process started with the construction of individual DES models that can capture the *task execution* and *resource-constraints*. Using these individual models corresponding to system components and specifications, we obtained the supervisor that contains all feasible execution sequences and anyone of those sequences can be used to schedule the task set. These steps have been illustrated through a running example. An experimental proof-of-concept validation of our synthesized scheduler has been performed by implementing two independent real-time tasks on Atlys Spartan-6 FPGA platform.

Chapter 4

An ILP-based Approach to Real-Time Scheduling of Task Graphs on Partially Reconfigurable FPGAs

In the earlier chapter of this dissertation, we have considered the supervisory control of timed discrete event system formalization for scheduling independent hard real-time tasks on a FPGA platform. In this chapter, we developed an offline ILP-based solution strategy for scheduling persistent real-time applications represented as a precedence-constrained task graphs on partially reconfigurable FPGAs. The designed solution strategy must not only handle all timing constraints, dependency constraints and FPGA based placement constraints but also correctly account for reconfiguration overheads involved in loading task bitstreams onto the configuration memory of the FPGA through the ICAP port. It is important to correctly account for the reconfiguration times because they consume significant overheads often being comparable to task execution times. We have experimentally validated the proposed offline solutions and observed that in offline, optimal solution can be achieved with moderate computational overheads through CPLEX solver.

4.1 System Model

This work assumes a runtime partially reconfigurable FPGA platform with support for relaxed task placement based on the generic 2D flexible area model. Thus each task T_i,

which is assumed to be rectangular in shape having dimension $w_i \times h_i$, may be placed anywhere on the FPGA floor of dimension $W \times H$, such that it does not overlap with other tasks placed on the floor or with the FPGA boundaries. The FPGA is accompanied by a separate General Purpose Processor (GPP) and memory. Task bitstream images are stored and maintained in a repository residing in memory. All hardware reconfigurations are performed according to a schedule generated offline under supervision of the GPP by loading bitstreams from the repository into the configuration memory of the FPGA, through its ICAP port.

We model a real-time application as a precedence constrained Directed Acyclic Graph (DAG) $G = (T, E)$, where T is a set of hardware tasks ($T = \{T_i \mid 1 \leq i \leq |T|\}$) and E is a set of directed edges ($E = \{\langle T_i, T_j \rangle \mid 1 \leq i, j \leq |T|; i \neq j\}$) representing precedence relations between distinct pairs of tasks. An edge $\langle T_i, T_j \rangle$ refers to the fact that task T_j can begin execution only after the completion of T_i. It is further assumed that a hardware task T_i may have k_i different versions / implementations; that is, $T_i = \{T_i^1, T_i^2, \ldots, T_i^{k_i}\}$. Although, all versions of a task produce the same output, their execution times, area requirements and accuracy of results may vary. Different versions of a task essentially mean different hardware circuit implementations corresponding to the same functionality [28]. Task versions with varying degrees of *area Vs. execution time trade-offs* may be obtained by controlling the degree of parallelism for a set of implementations. Some known techniques to control the degree of parallelism are loop unrolling with different unrolling factors and realizing hardwares with different pipeline stages. Hence, without loss of generality, this work assumes that, higher the area ar_i^j ($= w_i^j \times h_i^j$) consumed by the j^{th} version of the i^{th} task T_i^j, lower becomes its required execution time $Trun_i^j$ ($ar_i^j > ar_i^{j'} \implies Trun_i^j < Trun_i^{j'}$). The loading time $Trec_i^j$ corresponding to T_i^j is a function of its bitstream size, which in turn is proportional to the area ar_i^j consumed by T_i^j. A reward REW_i^j is assigned to T_i^j on successful completion. REW_i^j is a function of the accuracy of results produced by the j^{th} version of T_i, its execution requirement $Trun_i^j$, area demand ar_i^j as well as the relative importance of T_i with respect to other tasks. The application represented by the real-time task graph G is associated with an overall deadline D_{dag} within which a distinct chosen version of each task node in G must complete execution while satisfying all constraints, as discussed next.

4.2 Formalization of the Precedence Constrained Spatio-temporal Scheduling Problem

Given a real-time application, the proposed work endeavours to appropriately map and execute the application on a run-time dynamically reconfigurable FPGA platform. As discussed in the system model, the application is represented as a task graph with an overall deadline within which the execution of all task nodes in the graph must be completed. The execution of the task graph must be conducted by choosing *one* among a set of multiple allowable versions for each task node and associating appropriate *load-start* and *execution-start* times corresponding to the selected version of each task node. Here, *load-start* time for a task refers to the instant at which loading of the task's bitstream (associated with the selected task version) into the configuration memory of the FPGA through its ICAP port, commences. Similarly, *execution-start* time of a task refers to the instant corresponding to the commencement of execution of the task on the FPGA. As discussed above, each task version has a distinct reward value which is awarded subject to successful completion. The objective of the overall formulation is to maximize aggregate rewards through judicious selection of task versions so that execution of all nodes in the task graph may be completed within the given deadline while satisfying all dependency and resource related constraints, as discussed below. To summarize, an optimal solution methodology corresponding to the formulation must correctly determine:

- when to reconfigure or load a task ?
- which version of a task to load ?
- where to place a task ?
- when to start the execution of a task ?

4.2.1 Calculation of ASAP time for loading and execution

The ***As Soon As Possible (ASAP)*** time allocation policy for a task graph statically determines the earliest possible commencement time for loading as well as execution, corresponding to each task node over all possible choice of task versions, assuming unlimited available resources (in terms of both FPGA floor area and number of loading channels).

The pseudo-code for the allocation policy is presented in algorithm 1. The algorithm starts by topologically sorting all tasks in the graph into an ordered list τ. Each element

of τ is then assigned its ASAP load time Tl_i^s and ASAP start time Te_i^s in sequence, in order to ensure that a task node T_i is only considered for Tl_i^s and Te_i^s assignment when all its predecessors have already been assigned their respective ASAP load and start times.

Line-4 of the algorithm assigns system initiation time "0" as the earliest load time (Tl_i^s) for all tasks. This is because, there do not exist any explicit dependency constraint corresponding to the loading and placement of tasks on the FPGA. The earliest start time (Te_i^s) for each node T_i is assigned in line 5. Te_i^s is obtained as the maximum over the earliest completion times of all its predecessors and its earliest load completion time.

Algorithm 1: *ASAP Load time and Start time*

Input:
i. The task graph $G(T, E)$
ii. k_i: # version of each task T_i
iii. $Trec_i^k$: Loading time for the k^{th} version of T_i
iv. $Trun_i^k$: Execution time for the k^{th} version of T_i
Output:
i. Tl_i^s: ASAP load time for T_i:
ii. Te_i^s: ASAP start time of each task T_i

1 Compute topological ordering of G and store in ordered list τ;
2 **while** τ *is not empty* **do**
3 Select T_i from the first element of τ;
4 Tl_i^s=0;
5 $Te_i^s = \underset{j:(T_j,T_i)\in E}{max} \left\{ max\ (Te_j^s + \underset{1\le k\le k_j}{min}(Trun_j^k)), (Tl_i^s + \underset{1\le k\le k_i}{min}\ (Trec_i^k)) \right\}$;

4.2.2 Calculation of ALAP time for loading and execution

The ***As Late As Possible (ALAP)*** time allocation policy for a task graph statically determines the latest possible commencement time for loading as well as execution, corresponding to each task node over all possible choice of task versions, assuming unlimited available resources (in terms of both FPGA floor area and number of loading channels).

The pseudo-code for the allocation policy is presented in algorithm 2. The algorithm starts by doing a reverse topological sort of all tasks in the graph into an ordered list $\hat{\tau}$. Each element of $\hat{\tau}$ is then assigned its ALAP load time Tl_i^l and ALAP start time Te_i^l in sequence, in order to ensure that a task node T_i is only considered for Tl_i^l and Te_i^l assignment when all its successor nodes have already being assigned there respective ALAP load and start times.

If a task T_i has no successor, then its ALAP start time is given by the latest time at which it must be started so that the task is able to complete at the deadline instant using the version with the least execution time. Lines 4 and 5 of algorithm 2 depicts this calculation. Step 7 of the algorithm calculates ALAP start times for the remaining task nodes. Te_i^l is given by the latest time at which T_i must start in order to complete on or before the earliest ALAP time among its successors, using the fastest version for T_i. Finally, the ALAP load time for each task is obtained by subtracting its minimum loading time from its ALAP start time.

Algorithm 2: *ALAP Load time and Start time*

Input:
i.The task graph $G(T, E)$
ii. k_i: # version of each task T_i
iii. $Trec_i^k$: Loading time for the k^{th} version of T_i
iv. $Trun_i^k$: Execution time for the k^{th} version of T_i
V. D_{Dag}: The deadline of the task graph.
Output:
i. Tl_i^l: ALAP load time for T_i:
ii. Te_i^l: ALAP start time of each task T_i

1 Compute reverse topological ordering of G and store in ordered list $\hat{\tau}$;
2 **while** $\hat{\tau}$ *is not empty* **do**
3 Extract T_i, the first element of list $\hat{\tau}$;
4 **if** T_i *has no successor* **then**
5 $Te_i^l = D_{dag} - \min_{1 \le k \le k_i} (Trun_i^k)$;
6 **else**
7 $Te_i^l = \min_{j:(T_i,T_j) \in E} (Te_j^l) - \min_{1 \le k \le k_i} (Trun_i^k)$;
8 Assign Tl_i^l=Te_i^l-$\min_{1 \le k \le k_i} (Trec_i^k)$;

4.3 An ILP Formulation

In this section, we present an *Integer Linear Programming (ILP)* solution to the DAG scheduling problem. For this purpose, we define two sets of binary decision variables: i. $\mathcal{Z} = \{Z_{ikl} : i = 1, 2, ..., |T|; k = 1, 2, ..., k_i; l = Tl_i^s, ..., Tl_i^l\}$, ii. $\mathcal{X} = \{X_{ikl} : i = 1, 2, ..., |T|; k = 1, 2, ..., k_i; l = Te_i^s, ..., Te_i^l\}$. For both sets of variables, indices i, k and l respectively denote task id, corresponding version id and time step. $Z_{ikl} = 1$, if the k^{th} version of T_i (T_i^k) starts loading at the l^{th} time step. $Z_{ikl} = 0$, otherwise. On the

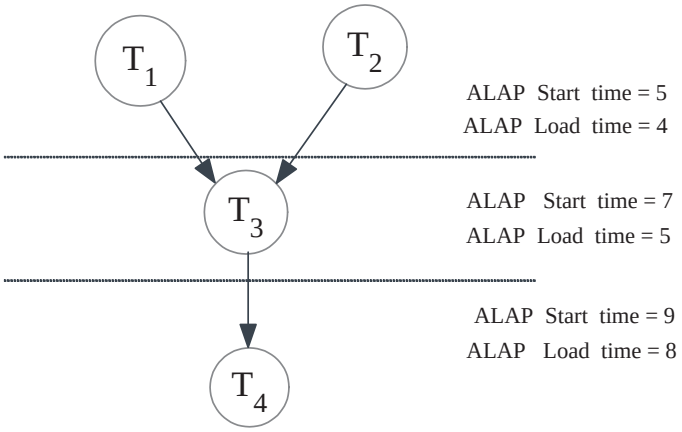


Fig. 4.1: ALAP start and load time allocation

other hand, $X_{ikl} = 1$, if the k^{th} version of T_i (T_i^k) starts execution at the l^{th} time step. $X_{ikl} = 0$, otherwise. We now present the required constraints on the decision variables to model this scheduling and placement problem. Additionally, we also define a set $\mathcal{Y}$ of integer decision variables: where $\mathcal{Y} = \{Y_i : i = 1, 2, ..., |T|\}$. Each decision variable $Y_i \in \mathcal{Y}$ is a two tuple Y_i=$\langle x_i, y_i \rangle$, where x_i, y_i denotes the left bottom coordinates of task T_i on the FPGA floor and thus, defines its exact placement. We now sequentially present the different constraints of the proposed ILP before presenting its overall objective function.

1. ***Reconfiguration Constraints:*** While loading (reconfiguring) a task at time step l in the FPGA, it must follow the following constraints:

   - ***Unique Load Time Constraint:*** During a reconfiguration, each task must start loading through the ICAP on the FPGA floor at a unique time step. That is,

   $$\forall i : 1 \leq i \leq |T| \mid \sum_{k=1}^{k_i} \sum_{l=Tl_i^s}^{Tl_i^l} Z_{ikl} = 1 \tag{4.1}$$

   The above constraint enforces that for each task T_i exactly one version will be selected for loading at a unique time step within $[Tl_i^s, Tl_i^l]$.

   - ***Single Load Channel (ICAP) Constraint:*** Only one task can be loaded through the single available ICAP port at a given time. Time taken to load

the k^{th} version of T_i is $Trec_i^k$. That is,

$$\forall l : 1 \leq l \leq D_{dag}| \quad \sum_{i=1}^{|T|} \sum_{k=1}^{k_i} \sum_{m=\alpha_i}^{l} Z_{ikm} \leq 1 \tag{4.2}$$

where $\alpha_i = max(0,\ l - Trec_i^k + 1)$. Equation 2 ensures that at any time l, the single ICAP port can be busy due to the ongoing load of atmost one task. A selected version (index k) of a task (index i) is in the process of being loaded at time l if it started loading atmost $l - Trec_i^k + 1$ time steps earlier.

2. ***Execution Constraints:*** The following constraints must be satisfied with respect to the start of execution of the tasks.

   - ***Unique Start Time Constraint:*** Each task must start execution on the FPGA floor at a unique time step. That is,

$$\forall i : 1 \leq i \leq |T| \mid \quad \sum_{k=1}^{k_i} \sum_{l=Te_i^s}^{Te_i^l} X_{ikl} = 1 \tag{4.3}$$

   - ***Load Completion Constraint:*** A task T_i can commence its execution only after its loading finishes. To enforce this, the following constraint must be satisfied for each task.

$$\forall i : 1 \leq i \leq |T| \mid \quad \sum_{k=1}^{k_i} \sum_{l=Te_i^s}^{Te_i^l} l \times X_{ikl} \geq \sum_{k=1}^{k_i} \sum_{l=Tl_i^s}^{Tl_i^l} l \times Z_{ikl} + \sum_{k=1}^{k_i} \sum_{l=Tl_i^s}^{Tl_i^l} Trec_i^k \times Z_{ikl} \tag{4.4}$$

     While LHS of the above inequality represents the time step corresponding to the start of execution of a task T_i, the RHS denotes its load completion time as a combination of load start time and load duration.

   - ***Dependency Constraint:*** Corresponding to each directed edge $(T_i, T_j \in E)$ in the DAG, the execution of task T_j must commence only after the completion of its predecessor, T_i. This dependency constraint between task T_i and T_j is

symbolically represented as follows:

$$\forall (T_i, T_j) \in E \mid \sum_{k=1}^{k_j} \sum_{l=Te_j^s}^{Te_j^l} l \times X_{jkl} \geq \sum_{k=1}^{k_i} \sum_{l=Te_i^s}^{Te_i^l} l \times X_{ikl} + \sum_{k=1}^{k_i} \sum_{l=Te_i^s}^{Te_i^l} Trun_i^k \times X_{ikl} \quad (4.5)$$

3. ***Placement Constraints:*** For a given temporal schedule of the tasks along with their selected versions, the placement constraints attempt to ensure that the tasks having overlapping life times on the FPGA floor do not spatially overlap with each other at any instant over the schedule length. Additionally, these constraints also guarantee that the tasks do not overlap with the FPGA boundaries.

   - ***Non-overlap constraints:*** Given a pair of tasks T_i^k and T_j^k, which must simultaneously co-execute on the FPGA floor and which should not overlap, we have four possible ways to position the two tasks so as to avoid overlap. Let $w_{i,k}$ and $h_{i,k}$ be the width and height of the selected version T_i^k of task T_i. Given a pair of tasks T_i^k, $T_j^{k'}$ whose execution life time overlap on the FPGA floor, T_i^k can either be relatively positioned to the right or left or below or above $T_j^{k'}$. At any given time, only one of the four possibilities mentioned above can be true. These relative position constraints may be transformed into inequalities given below.

$\forall l : 1 \leq l \leq D_{dag}, \forall (i,j) : 1 \leq i, j \leq |T|,$

$\forall k : 1 \leq k \leq k_i, \forall k' : 1 \leq k' \leq k_j, |$

$$Z_{ikl} \times (x_i + w_{i,k}) \leq Z_{jk'l} \times x_j, \ (T_i^k \text{ is to the left of } T_j^{k'}), \text{ or} \quad (4.6a)$$

$$Z_{ikl} \times (x_i - w_{j,k'}) \geq Z_{jk'l} \times x_j, \ (T_i^k \text{ is to the right of } T_j^{k'}), \text{ or} \quad (4.6b)$$

$$Z_{ikl} \times (y_i + h_{i,k}) \leq Z_{jk'l} \times y_j, \ (\ T_i^k \text{ is below of } T_j^{k'}), \text{ or} \quad (4.6c)$$

$$Z_{ikl} \times (y_i - h_{j,k'}) \geq Z_{jk'l} \times y_j, \ (T_i^k \text{ is above of } T_j^{k'}) \quad (4.6d)$$

As per equation 4.6a, at the time step l, if both the tasks (T_i^k, T_j^k) attempt to get placed on the FPGA floor then it can only be possible if the inequality

(as stated in eqaution 4.6a) satisfies. Given the two temporally overlapping tasks T_i^k, T_j^k, this equation enforces the necessary constraint to allow T_i^k to be feasibly placed to the left of T_j^k, avoiding any possibility of spatial overlap.

To satisfy one of these equations, we use all-pair binary variables a_{ij} and b_{ij} for tasks T_i^k and $T_j^{k'}$ as shown in Table 4.1.

Table 4.1: Different values of the variable pair (a_{ij}, b_{ij})

a_{ij}	b_{ij}	Remarks
0	0	T_i^k is to the left of $T_j^{k'}$
1	0	T_i^k is to the right of $T_j^{k'}$
0	1	T_i^k is below of $T_j^{k'}$
1	1	T_i^k is above of $T_j^{k'}$

W and H are the width and height of the FPGA, respectively. For each pair of tasks, we can rewrite these four equations (4.6a -4.6d) so that only one of them becomes non-trivial based on the actual relative position.

$$\forall l : 1 \leq l \leq D_{dag}, \forall (i,j) : 1 \leq i, j \leq |T|$$
$$\forall k : 1 \leq k \leq k_i, \forall k' : 1 \leq k' \leq k_j, |$$
$$Z_{ikl} \times (x_i + w_{i,k}) \leq Z_{jk'l} \times (x_j + W(a_{ij} + b_{ij})) \tag{4.7a}$$
$$Z_{ikl} \times (x_i - w_{j,k'}) \geq Z_{jk'l} \times (x_j - W(1 - a_{ij} + b_{ij})) \tag{4.7b}$$
$$Z_{ikl} \times (y_i + h_{i,k}) \leq Z_{jk'l} \times (y_j + H(1 + a_{ij} - b_{ij})) \tag{4.7c}$$
$$Z_{ikl} \times (y_i - h_{j,k'}) \geq Z_{jk'l} \times (y_j - H(2 - a_{ij} - b_{ij})) \tag{4.7d}$$

- ***Chip boundary constraints:*** It has to be ensured that the tasks are located within the boundary of the FPGA having dimension $W \times H$, that is,

$$\forall l : 1 \leq l \leq D_{dag}, \forall i : 1 \leq i \leq |T|, \forall k : 1 \leq k \leq k_i|$$
$$Z_{ikl} \times (x_i + w_{i,k}) \leq W \tag{4.8a}$$
$$Z_{ikl} \times (y_i + h_{i,k}) \leq H \tag{4.8b}$$

- ***Additional constraints:*** The type and range of the variables are defined as

integer, that is,

$$x_i \geq 0, y_i \geq 0 \tag{4.9a}$$

$$a_{ij}, b_{ij} \in \{0, 1\} \tag{4.9b}$$

4. ***Deadline Constraint:***

   - In order to ensure that the application G meets its end-to-end absolute deadline D_G, the sink node $T_{|T|}$ must complete execution by D_G. That is,

$$\sum_{k=1}^{k_{|T|}} \sum_{l=Te^s_{|T|}}^{Te^l_{|T|}} l \times X_{|T|kl} + \sum_{k=1}^{k_{|T|}} Trun^k_{|T|} \times X_{|T|kl} \leq D_{dag} \tag{4.10}$$

5. ***Objective:*** The objective of the formulation is to choose that feasible solution which maximizes overall system level reward through appropriate choice of task versions. Hence, the objective can be written as follows:

$$Maximize \quad \sum_{i=1}^{|T|} \sum_{k=1}^{k_i} \sum_{l=Tl^s_i}^{Tl^l_i} Z_{ikl} \times REW^k_i \tag{4.11}$$

4.4 Experiments and Results

In this section, we evaluate the performance of our proposed ILP formulation.

4.4.1 Experimental Setup

Performance evaluation of the proposed ILP formulation has been carried out through a comprehensive set of simulation-based experiments. The principal metric based on which evaluations have been carried out are as follows;

(i) *Normalized Achieved Reward (NAR in %)*:

$$NAR = \frac{\sum_{i=1}^{n} REW_i}{\sum_{i=1}^{n} REW^{k_i}_i} \times 100 \tag{4.12}$$

where $\sum_{i=1}^{n} REW_i$ denote the total reward obtained on successful execution completion of the DAG and $\sum_{i=1}^{n} REW^{k_i}_i$ is the maximum possible achievable reward, which is the

sum of rewards for the highest versions. As referenced in [65], for an application which consists of (η) number of tasks, width(w), height (h), execution time(e) and deadline (D_{app}), the load factor or utilization ratio of an application on the FPGA size of W and H is given by,

$$APP_{load} = \frac{\sum_{i=1}^{\eta} w_i \times h_i \times e_i}{W \times H \times D_{app}} \tag{4.13}$$

Let w_{avg}, h_{avg} and e_{avg} be the average width, height and execution time respectively for a given task sets, then equation 5.3 can be re-written as,

$$n = APP_{load} \times \left(\frac{D_{app} \times \left\lfloor \frac{W}{w_{avg}} \right\rfloor \times \left\lfloor \frac{H}{h_{avg}} \right\rfloor}{e_{avg}} \right) \tag{4.14}$$

where n is the number of tasks nodes obtained for a predfined APP_{load} value.

(ii) *Utilization (U) :* For an application which consists of n-number of tasks, width(w), height(h) and execution time(e), the utilization is given by equation (4.15). Where *makespan* is the DAGs maximum execution time completion and $W \times H \times makespan$ represents the total amount of resources available on the FPGA during the makespan.

$$U = \frac{\sum_{i=1}^{n} w_i \times h_i \times e_i}{W \times H \times makespan} \tag{4.15}$$

(iii) *Cummulative schedules execution time :* As stated in [65], this time is the run-time required by the scheduling algorithm (and the underlying placement) each time the scheduler is invoked. The $Sched\_exe\_time$ is expressed by equation (4.16), where n is the number of invocation of the scheduler and $Exec\_Time(i)$ the runtime required by the scheduling algorithm in its i^{th} call.

$$Sched\_exe\_time = \sum_{i=1}^{n} Exec\_Time(i) \tag{4.16}$$

Now various types of datasets have been generated by setting different values for the following parameters

1. APP_{load}: The APP_{load} is varied between 40% and 90%.

2. *Floor size of the FPGA*: All our experiments have been conducted assuming the FPGA floor size to be 52×128 (same as the actual dimension of Xilinx Virtex-4

(XC4VFX60)).

3. *DAGs graph*: The number of DAG edges used in our experiment varies from $(n+4)$ for an APP_{load} value of 40% to $(n+9)$ for an APP_{load} of 90%, where n is the task nodes obtained from equation 4.13. In generating the DAGs, except for the source task node, every node has at least one predecessor.

4. *Task versions*: We used a similar approach for task version generation as mentioned in [65]. Versions represent the different hardware implementations for a given task. Task versions are denoted as Normal version(NV), Half Normal Version (HNV), and Twice Normal Version (TNV). We denote the HNV as a half-sized NV obtained by dividing the width(resp. the height) by two but requires a twice longer execution time and a half reward value. The TNV has twice the width (resp. the height) of NV, but requires a half shorter execution time and a twice reward value. In addition, the reconfiguration time is directly proportional to the area used as was assumed in the introduction section. It is assumed that each task could have at least one version (i.e., the NV) and at most three versions (i.e., NV, HNV, and TNV). Therefore, task version assignment is taken from a uniformly distributed range of $[1, 4]$.

5. *Task size (Spatial Resource Demand)*: We have considered synthetic task sets of sizes varying from (16×16) and (32×32) for evaluating the performance of our heuristic algorithm as mentioned in [114]. Rectangular task dimensions have been obtained by separately generating widths (w_i) and heights (h_i) of the tasks from distributions having mean $\mu_{sz} = 8$ and standard deviation $\sigma_{sz} = 16$. For each task, the execution time and reconfiguration time ranges are taken from [114]. Accordingly, the execution time range is from 10 to 60 time units (ms), reconfiguration time from $500\mu s$ to $600\mu s$ and the reward incurred varies in the range of 10 to 100. It is to be noted that only the NV parameter values are generated.

6. *Number of Task*: We have considered the following number of task nodes for evaluating the ILP formulation; 10, 15, 20, and 25.

The data points shown in the charts are obtained as the average over 10 runs. The ILP solutions have been generated using the IBM CPLEX tool in OPL format. The simulation was performed on Intel(R) Core(TM) i5-1035G1 CPU @1.00GHZ 1.19GHz and 8GB installed memory(RAM).

4.4.2 Results

Table 4.2 shows the normalized reward obtained by the ILP formulation with varying APP_{load}. From the table, it can be observed that the normalized reward remains comparable with the changes of APP_{load}. This may be attributed to the fact that when the APP_{load} is increasing the number of tasks also increases, thus equation (4.13) remains more or less the same. These results, therefore, indicate that the achieved reward may be considered to be robust against variation of APP_{load}.

Figure 4.2 shows the utilization of the ILP formulation with a varying number of tasks. The utilization rate is 9%, 11%, and 6.5% when the number of tasks varies from 10 to 15, 15 to 20, and 20 to 25 respectively. The rate is attributed to the fact that as the number of tasks increases, the dependency constraint becomes more complex which resulted in an unsatisfied placement constraint.

Table 4.2: Normalized Acheived Result

$\mathbf{APP}_{load}$	**NAR (%)**
40	1.0
50	0.98
60	0.95
70	0.89
80	0.86
90	0.84

Figure 4.3 shows the performance of the overall execution time produced by our proposed ILP formulation. The figure illustrates that the execution time increases with the increase in the number of tasks. This attributes to the fact that the more the number of tasks increases the more steps that have to be checked for constraint satisfaction and thus, the final output can be obtained after a long interval time.

4.5 Summary

In this chapter, we have proposed an offline scheduling strategy for real-time precedence-constrained task graphs consisting of multi-mode safety-critical tasks where each mode is characterized by distinct spatial and temporal resource demands, reward obtained by the system on successful execution, and possibly accuracy of results produced. We have

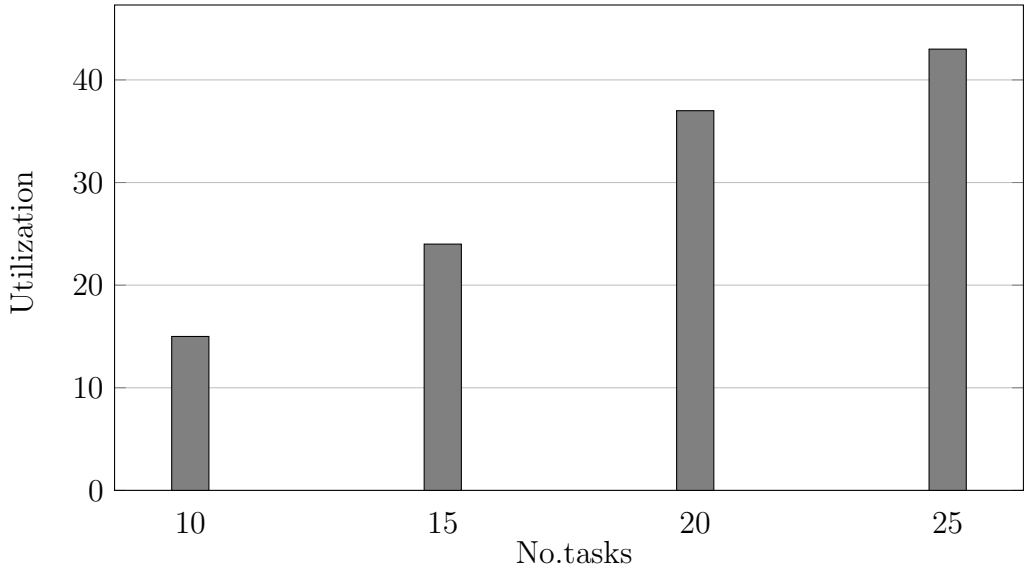


Fig. 4.2: Utilzation vs. No.tasks

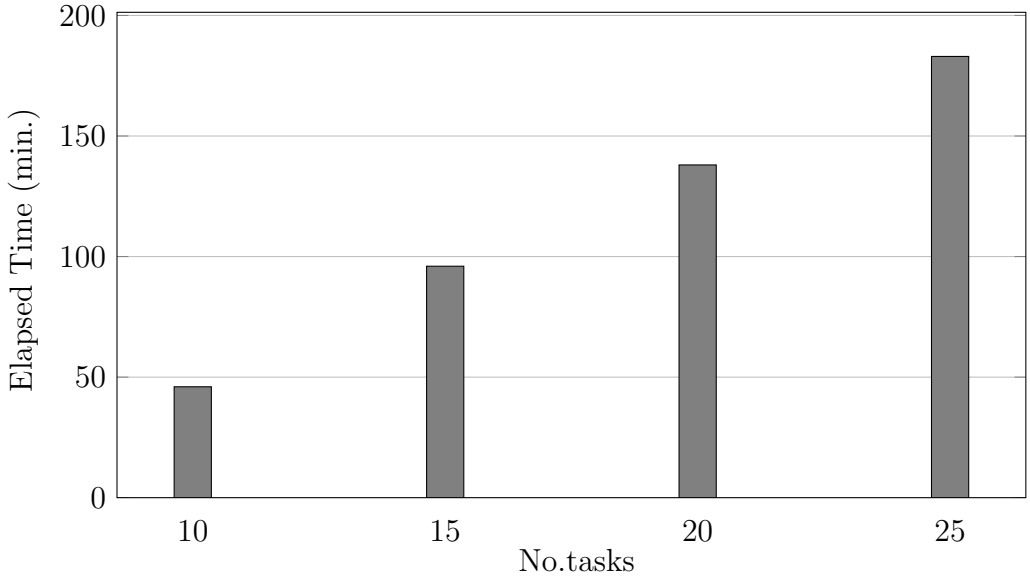


Fig. 4.3: Elapsed time vs. No. tasks

formulated the ILP for the problem and solved it through the CPLEX solver.

Chapter 5

Heuristic Approach to Real-Time Scheduling of Task Graphs on Partially Reconfigurable FPGAs

In the previous chapter, we proposed an ILP-based solution strategy for scheduling persistent real-time applications represented as a precedence-constrained task graph on partially reconfigurable FPGAs. Although ILPs provide an optimal solution, their highly exponential computational complexities make them prohibitively expensive when problem sizes become even moderately big in terms of the no. of tasks, their temporal (execution time) and spatial (dimensions on the FPGA floor) requirements, average branching factor, reconfiguration overheads, etc. In this chapter, we develop an offline heuristic algorithm for scheduling persistent real-time applications represented as a precedence-constrained task graph on partially reconfigurable FPGAs. The main contributions of this chapter are as follows. First, we formalize a multi-variant task scheduling problem for partially reconfigurable systems. Then, we propose our scheduling algorithm to maximize the accuracy by executing the highest task's version by managing reconfiguration overhead, dependency constraint, and resource constraints. Along with the scheduling strategy, we also proposed a novel algorithm to tackle the task placement problem for FPGAs. An evaluation using both synthetic hardware tasks revealed the efficacy of the proposed placement strategy.

5.1 System Model & Problem formulation

This work assumes a runtime partially reconfigurable FPGA platform with support for relaxed task placement based on the generic 2D flexible area model. Each task T_i is assumed to be rectangular in shape having dimension $w_i \times h_i$. T_i may be placed anywhere on the FPGA floor of dimension $W \times H$, such that it does not overlap with other tasks placed on the floor or with the FPGA boundaries. The FPGA is accompanied by a separate General Purpose Processor (GPP) and memory. Task bitstream images are stored and maintained in a repository residing in memory. All hardware reconfigurations are performed according to a schedule generated offline under the supervision of the GPP, by loading bitstreams from the repository into the configuration memory of the FPGA, through its ICAP port.

We model a real-time application as a precedence constrained Directed Acyclic Graph (DAG) $G = (T, E)$, where T is a set of hardware tasks ($T = \{T_i \mid 1 \leq i \leq |T|\}$) and E is a set of directed edges ($E = \{\langle T_i, T_j \rangle \mid 1 \leq i, j \leq |T|; i \neq j\}$) representing precedence relations between distinct pairs of tasks. An edge $\langle T_i, T_j \rangle$ refers to the fact that task T_j can begin execution only after the completion of T_i. It is further assumed that a hardware task T_i may have k_i different versions / implementations; that is, $T_i = \{T_i^1, T_i^2, \ldots, T_i^{k_i}\}$. Although, all versions of a task produce the same output, their execution times, area requirements and accuracy of results may vary. Different versions of a task essentially mean different hardware circuit implementations corresponding to the same functionality [28].

Task versions with varying degrees of *area Vs. execution time trade-offs* may be obtained by controlling the degree of parallelism for a set of implementations. Some known techniques to control the degree of parallelism are loop unrolling with different unrolling factors and realizing hardware with different pipeline stages. Hence, without loss of generality, this work assumes that, higher the area ar_i^j ($= w_i^j \times h_i^j$) consumed by the j^{th} version of the i^{th} task T_i^j, lower becomes its required execution time $Trun_i^j$ ($ar_i^j > ar_i^{j'} \implies Trun_i^j < Trun_i^{j'}$). The loading time $Trec_i^j$ corresponding to T_i^j is a function of its bitstream size, which in turn is proportional to the area ar_i^j consumed by T_i^j. A reward REW_i^j is assigned to T_i^j on successful completion. REW_i^j is a function of the accuracy of results produced by the j^{th} version of T_i, its execution requirement $Trun_i^j$, area demand ar_i^j as well as the relative importance of T_i with respect to other tasks.

Each task node is associated with a *load-start* time and an *execution-start* time. Here, *load-start* time for a task refers to the instant at which loading of the task's bitstream

(associated with the selected task version) into the configuration memory of the FPGA through its ICAP port, commences. Similarly, *execution-start* time of a task refers to the instant corresponding to the commencement of execution of the task on the FPGA. The application represented by the real-time task graph G is associated with an overall deadline D_{dag} within which a distinct chosen version of each task node in G must complete execution.

Problem formulation: The objective of the work is to generate a spatio temporal schedule which maximizes the aggregate rewards through the judicious selection of task versions for a given runtime partial FPGA platform. The generated schedule must ensure that execution of all nodes in the task graph is completed within the given deadline while satisfying all dependency and resource related constraints.To summarize, a solution methodology corresponding to the problem must correctly determine:

- when to reconfigure or load a task
- which version of a task to load
- where to place a task
- when to start the execution of a task

Table 5.1: Parameters Values for Example Task Sets

Task	$Trun_i$	$Trec_i$	w_i	h_i	REW_i
T_1^1	2	2	12	16	20
T_1^2	3	1	12	8	10
T_2^1	2	2	12	12	10
T_2^2	4	1	10	8	5
T_3^1	2	2	12	12	20
T_3^2	2	2	8	12	10
T_4^1	3	1	8	12	10
T_4^2	4	1	10	8	5

$Trun_i$: Execution time for task T_i; $Trec_i$: Loading time for task T_i ; w_i: Task Width; h_i: Height of T_i; REW_i: Reward obtained for task T_i

***Example*:** Let us consider the real-time task graph shown in figure 5.1. The task graph consist of four task, T_1,..., T_4, each task T_i having two version T_i^1, T_i^2. A task version (T_i^j) is associated with a distinct runtime ($T_{run_i}^j$), reconfiguration time ($T_{rec_i}^j$), dimension/size (ar_i^j)) and reward (REW_i^j). The floor size of the FPGA is assumed to be 12×24. The end to end deadline of the task graph is $D_G = 12$.

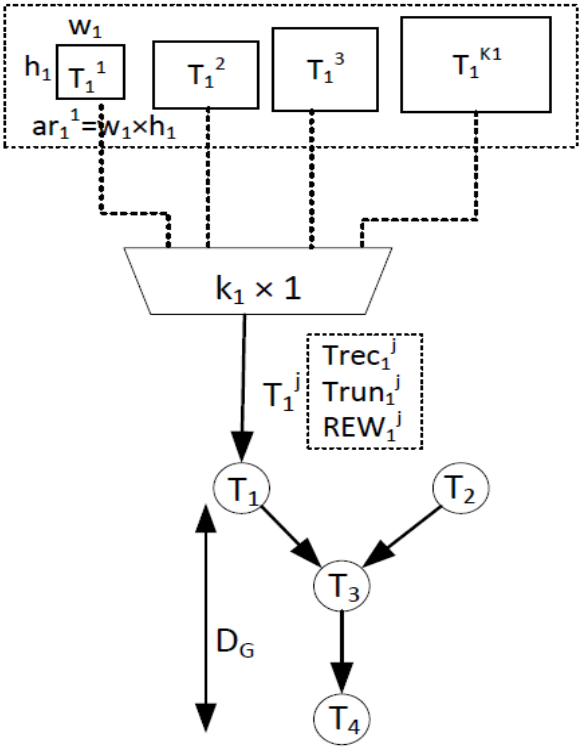


Fig. 5.1: The Task Graph

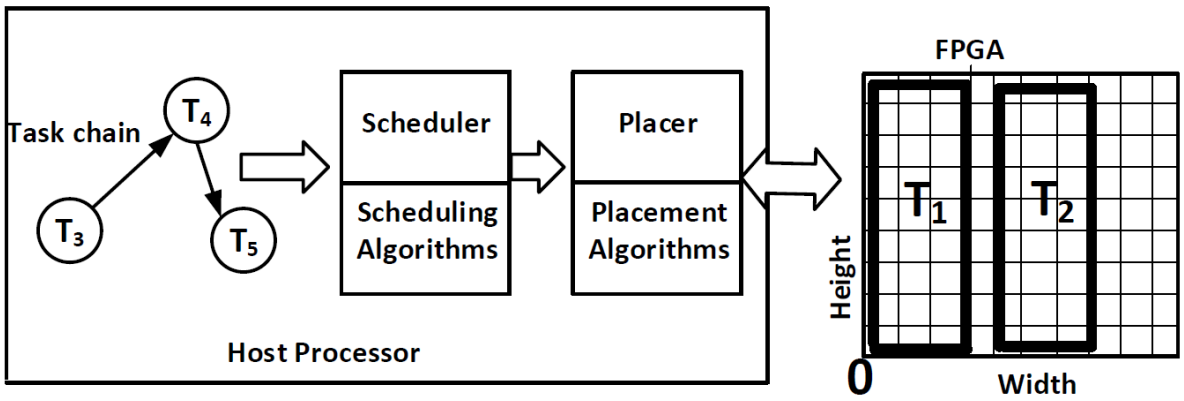


Fig. 5.2: Functional Model

5.2 Scheduling and Placement Heuristic

The overall functional representation of the proposed system model is presented in figure 5.2. The figure illustrates the Spatio-temporal scheduler for task graphs. The proposed scheduling strategy is composed of two important phases: *Scheduling* and *Placement*. Now, we discuss each phase in detail.

Algorithm 3: *Variation Aware Dag Scheduling*

Input:
i.The task graph $G(T, E)$
ii. ζ_i: No. versions of each task T_i
iii. $Trec_i^j$: Loading time of the j^{th} version of T_i
V. $Trun_i^j$: Execution time of j^{th} version of T_i
V. D_{Dag}: The deadline of the task graph.
Vi. REW_i^j: Reward for executing the j^{th} versison of T_i

Output:
i. Task Schedule /* Task versions (k_i), Load (Tl_i) and Execution start times (Te_i) */
ii. Achived system-level reward : REW_{sys}

/* Let $K = \{k_i : i = 1, 2, ..., |T|\}$, $Tl = \{Tl_i : i = 1, 2, ..., |T|\}$, $Te = \{Te_i : i = 1, 2, ..., |T|\}$, $\mathcal{Y} = \{Y_i : i = 1, 2, ..., |T|\}$ denote the sets representing the currently selected versions, load start times execution start times and left bottom coordinate of task T_i in the given task set T. */
/* Let τ denote the ordered set of tasks obtained by topologically sorting T */
$\forall T_i \in T$, Set $K_i = \zeta_i$ (highest version)
$\forall T_i \in T$, Compute the Penalty Factor $PF(T_i, k_i)$, using equation 5.1
Create a min-heap of tasks in T with the $PF(T_i, k_i)$ values as the key;
Calculate System level reward as: $REW_{sys} = \sum_i REW_i^{k_i}$;
while *MVDS* (τ, K, Tl, Te, $\mathcal{Y}$) $\neq$ *TRUE* **do**
    Extract the task T_j at the root of the min-heap;
    $k_j = k_j - 1$; /* Decrease the current version of T_j by one; */
    Update system level Reward as: REW_{sys}=$REW_{sys} - REW_j^{k_j+1} + REW_j^{k_j}$;
    Compute the $PF(T_j, k_j)$ and reheapify;
Return REW_{sys}

5.2.1 Scheduling

The algorithm "variation aware dag scheduling (vads)" is a heuristic scheduling strategy that attempts to maximize overall system-level reward by choosing an appropriate version of each task for execution on the FPGA floor. The algorithm starts by attempting to generate a feasible schedule by calling the "Multiple Variant Dag Scheduling (MVDS)" function with the highest version of all tasks. This feasible schedule must satisfy all the constraints mentioned in the problem formulation. However, if a feasible solution cannot be generated (this happens when function *MVDS()* returns FALSE) VADS chooses an individual task for which a metric known as *"Penalty Factor (PF)"* has the lowest value. The version of the selected task is decreased by one and then MVDS() is called again in an attempt to generate a feasible schedule with the currently chosen task versions. This process continues either until MVDS() generates a feasible schedule, or fails to do so even

Algorithm 4: *MVDS (K, Tl, Te, $\mathcal{Y}$, τ)*

Input:
i. $Trec_i^{k_i}$: Load time requirement for the k_i^{th} version of T_i
ii. $Trun_i^{k_i}$: Execution time requirement for T_i's k_i^{th} version
iii. D_{Dag}: Deadline of the task graph.
Output: TRUE / FALSE: Feasible or infeasible schedule

1 /* Input parameters K, Tl, Te and $\mathcal{Y}$ denote the sets representing the currently selected versions, load start times, execution start times and left bottom placement coordinates of each task T_i in the given task set T */
2 /* Input parameter τ represents the ordered set of tasks obtained by topologically sorting T */
3 $\hat{\tau} = \tau$ /* Copy tasks in ordered set τ into $\hat{\tau}$ */
4 Set ICAP-load = FALSE; /* ICAP-load : A flag which is set to FALSE if the ICAP is available for loading; TRUE, otherwise */
5 /* Let CT denote the set of tasks currently placed on the FPGA floor; */
6 /* Let RT denote the subset of CT, comprising of tasks currently executing on the FPGA floor; */
7 $CT = RT = \phi$;

with all tasks being in their lowest versions.

The penalty Factor (PF) is associated with the current version of each task. For the K_i^{th} version of a task say T_i, the penalty factor $PF(T_i, k_i)$ is defined as the ratio of the reduction in reward achieved by T_i with respect to the change in spatial resource consumed by it, when T_i's version decreases from k_i to $k_i - 1$. Penalty Factor $PF(T_i, k_i)$ is represented as :

$$PF(T_i, k_i) = \frac{REW_i^{k_i} - REW_i^{k_i-1}}{ar_i^{k_i} - ar_i^{k_i-1}} \tag{5.1}$$

In the attempt to generate a feasible schedule, the MVDS () is iteratively called by VADS as shown in the while loop (line 7-11, of algorithm 1). At any iteration, the tasks are maintained as min heap with a PF values of tasks as key. The tasks with the minimum PF value which is at the root of the heap is extracted and its level is decreased by one. If the task has still not reached its lowest level, its PF value is recalculated and then the task is reinserted back into the heap.

This value will be stored in a min-heap and a task (say, T_i) from the root of the heap will be selected. The version of T_i will be decreased by one and the corresponding system level reward will be re-calculated and the heap will be reformed based on the new version of a task. This step will continue until the feasible schedule (both spatially and temporally) is obtained.

```
8  for t = 0; t ≤ D_dag AND τ̂ ≠ NULL; t++ do
9      /* Function placer (T_j, CT, 𝒴) determines the coordinates Y_i of T_j, if placeable
       and returns TRUE; returns FALSE, if T_j is not placeable within the already
       placed tasks in CT */
10     /* Let T_j denote the first task currently in τ̂ and ICAP-Q denote the ordered
       set of placeable tasks which are yet to be loaded through ICAP */
11     while (placer (T_j, CT, 𝒴) == TRUE) do
12         Enque T_j into ICAP-Q
13         τ̂ = τ̂ \ {T_j} /* Delete T_j from τ̂ */
14     if (ICAP-load == FALSE AND ICAP-Q ≠ NULL) then
15         ICAP-load = TRUE /* Set ICAP port to busy; */ Load (T_j); /* Allocate
           ICAP for loading T_j */
16         Tl_j = t /* Set current time t as the load start time */
17         ILR= Trec_j^{k_i}; /* start loading T_j; ILR: an integer variable which holds the
           remaining time required to load the current task through ICAP */
18     else
19         ILR = ILR-1; /* Decrement remaining time */
20         if (ILR == 0) then CT = CT ∪ {T_j}; /* Add T_j to the set of currently
           placed task */
21             ICAP-load = FALSE;
22     if T_j ∈ CT AND all predecessors of T_j has finished execution then
23         RT= RT ∪ {T_j}; /* Add T_j to the set of running tasks */
24         Te_j = t /* Set t as the execution start time */
25         ER_j=Trun_j^{k_i}; /* start executing T_j; ER_j : An integer variable which holds
           the remaing execution requirement of T_j */
26     ∀T_j ∈ RT , ER_j = ER_j - 1
27     if ER_j ==0 /* T_j has completed execution */ then
28         RT = RT \ T_j; CT = CT \ T_j; /* Delete T_j from RT and CT */
29         FT = FT ∪ T_j /* Add T_j to set FT of finished tasks */
30 if |FT| ≠ |T| then
31     Return FALSE
32 else
33     Return TRUE;
```

Description of Function MVDS()

The actual scheduling and placement of tasks at each time step is carried out in function *MVDS()*. This Algorithm works as follows. Initially, the ordered set of tasks (τ) is stored in a set $\hat{\tau}$. There is only a single loading channel (PCAP/ICAP port) through which a task is downloaded into the FPGA and hence, the status of the ICAP is represented by a flag *ICAP-load* to mitigate any conflict. Initially, the ICAP port is free so *ICAP-load* is set as FALSE . At the same time, there is no task present in the FPGA thus, both RT and CT are set as NULL. The main part of the algorithm is presented in lines [8-33] detailing what happens at each time step t. It is repeated until the deadline of the DAG (D_G) is reached and the $\hat{\tau}$ is empty, i.e. all the tasks have been scheduled. In line [11-13], the *placer* employs a placement heuristic (described in next section) to find out whether task T_j (the first task in $\hat{\tau}$) will be placeable or not along with the already placed tasks on FPGA (represented as CT). If the T_j is found to be placeable then it is inserted into an ordered list which stores all the tasks that are ready to be loaded through ICAP (termed as *ICAP-Q*) and correspondingly, T_j will be removed from the $\hat{\tau}$.

Once the task is loaded into the *ICAP-Q*, the current status of the ICAP port will be checked. If the ICAP is found to be free (*ICAP-load* == *FALSE*) then the first task from $\hat{\tau}$ will be loaded and the ICAP port will be set as busy till the loading of the corresponding task get finished. The variable ILR will indicate the duration for which the ICAP will remain busy (line [17]). As the loading of task initiates, the ILR will decrease. ILR becomes zero when a task finishes its loading and thus, included in the list CT. Obviously as a consequence, the ICAP status will be set into free (line [19-21]).

Line ([22-29]) describes the scenario of the tasks execution on FPGA floor. A task (say, T_j) already present on the FPGA (member of set CT) will be ready for execution when all of its predecessors finish their execution requirements. Consequently T_j will be added in the set RT. All tasks belong to RT will continue their execution until their execution requirements get finished. The variable ER_j denotes the remaining execution requirement of T_j and thus, ER_j become zero when a task finishes its execution. After a task finishes its execution, it will be added to an another set FT and will be deleted from CT, RT.

At the end of the algorithm, it will be checked whether the number of finished task ($|FT|$) is equal to the number of tasks given in the input set T. Any mismatch will infer

a incomplete schedule. Otherwise, it will denote a successful schedule and *MVDS()* will return TRUE (line [30-33]).

Example: Example: Let us consider the real-time task graph shown in figure 5.1. The parameters associated with its task nodes are depicted in Table 5.1. The floor size of the FPGA is assumed to be 12×24. The end-to-end deadline of the task graph is $D_G = 12$. The proposed heuristic (*Variation Aware Dag Scheduling*) initially selects the highest versions for all tasks and attempts to generate a feasible schedule by calling MVDS(). However, MVDS() returns FALSE as the generated schedule (shown in Figure 5.3) violates the deadline. The reason behind this failure is that the highest versions of T_1 and T_2 cannot be simultaneously placed in the floor of the FPGA.

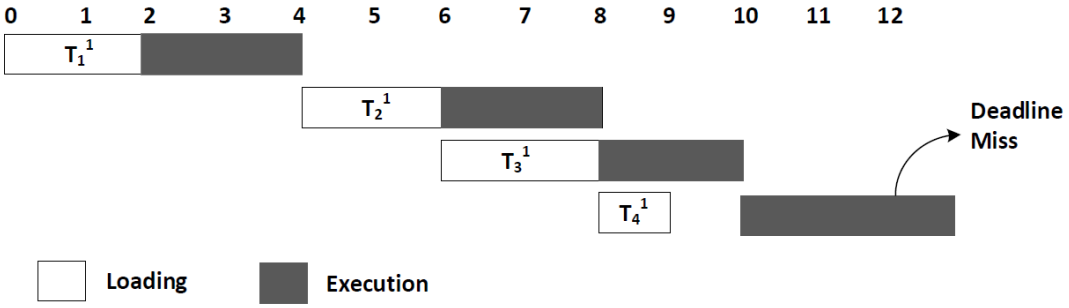


Fig. 5.3: Infeasible task schedule with highest versions

Now, VADS calculates the*Penalty Factor (PF)* for all tasks ($\{PF_2 = 0.07; PF_1 = 0.10; PF_4 = 0.31; PF_3 = 0.41\}$). As T_2 has the lowest PF value, its version is decremented by 1. The versions of all other tasks remain the same. MVDS() is then called again in an attempt to again generate a feasible schedule.

This time, MVDS() returns TRUE as a feasible schedule can be generated, as shown in figure 5.4.

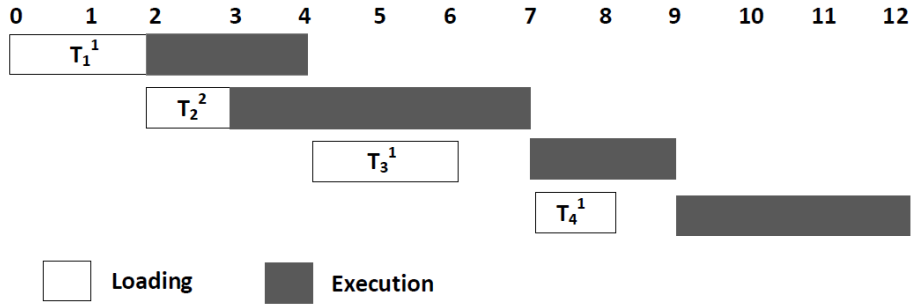


Fig. 5.4: Feasible task schedule

Table II shows the selected version, load start time, execution start time, placement coordinates and obtained reward values for each task node of the task graph. From Table II, the following points can be noted.

- A unique version of each task is loaded and executed.
- At any time instant, at most one task occupies the ICAP port
- No task start execution before its loading completes
- All inter-task precedence constraints are always satisfied
- The tasks never spatially overlap each other or with the FPGA boundary
- The sink task node is able to complete execution before the deadline

Table 5.2: Obtained values for scheduled Task Sets

Tasks	Selected Version	Load start time	Execute start time	Positions of Left bottom corner	Reward obtained
T_1	1	0	2	(0,0)	20
T_2	2	2	3	(0,16)	5
T_3	1	4	7	(0,0)	20
T_4	1	7	9	(0,12)	10

$Trun_i$: Execution time for task T_i; $Trec_i$: Loading time for task T_i ; w_i: Task Width; h_i: Height of T_i; REW_i: Reward obtained for task T_i

Algorithm 5: placer (T_j, PB)

Input:
i. T_j : j^{th}task placement request
ii. PB: Set of placed tasks (blocks) that combines CT and $\mathcal{Y}$ from *MVDS()*
Output: TRUE / FALSE: Feasible or infeasible placement

```
/*Let MERs be the set of maximal empty rectangles and LOC holds the corner with the
maximum overlap counter*/
NOR( PB );
COS( Graph);
POS( Graph);
LOC = MOP( MERs, T_j, PB );
if LOC == NULL then
    return FALSE;
else
    return TRUE; /*place the task at the first element of Max_Value*/
```

5.2.2 Placement

For a specific version of each task as selected by VADS (), MVDS () generates a deadline meeting temporal schedule if possible. However, this temporal schedule can be deemed to be feasible only if it is spatially schedulable on the FPGA floor. The placement strategy attempts to generate such a feasible spatial schedule. Specifically, for a subset of tasks (say, τ) having overlapping times in the temporal schedule, the placement strategy attempts a sub-region of size $w_i \times h_i$ for each task $T_i \in \tau$ such that no sub-region overlaps with the device boundaries or with other sub-regions. The spatial schedule consists of placed tasks along with vacant regions. A vacant region within a set of already placed tasks whose area is not large enough to allow the feasible placement of any other task is considered to be wasted due to fragmentation. One of the principal goals of any placement strategy is to minimize the total unutilized area lost due to fragmentation.

The MVDS () algorithm invokes the top placer module (i.e., placer ()) in line ([11-13]). The placer () module return TRUE for the feasible placement and FALSE otherwise. The module is composed of four main heuristic algorithms: (i)The Non_Overlap_Rectangle()- scan the FPGA area, identify the empty rectangles and store them as a forest graph where nodes are rectangles and edges represent the neighbor rectangles. (ii)Complete_Overlap_Side()- build the initial set of maximal empty rectangles (MERs). The algorithm collects all the complete/full overlap line segments between neighboring rectangles and merges the rectangles starting from the longest overlap line segment. (iii) Partial_Overlap_Side () - obtain the final set of maximal empty rectangles (MERs). The algorithm merges all rectangles that share same partial overlap line segments with their neighbors. (iv) Maximal_Overlap_Placement - determines the best placement location for the requesting task. The algorithm computes the maximum overlap side counter for the requesting task against the set of maximal empty rectangles (MERs) obtained previously. The task is placed in the empty rectangle whose overlap side counter is the maximum.

Non Overlap Rectangle (NOR)

The algorithm starts scanning the FPGA area by drawing the sweeping lines. Horizontal and vertical sweeping lines (shown as the red dot lines in fig.5.5) are drawn along the four boundaries of the placed tasks (say T_1 and T_2 of fig.5.5) beginning from the left end of the FPGA boundary to the right, and from the bottom to top. The intersection points

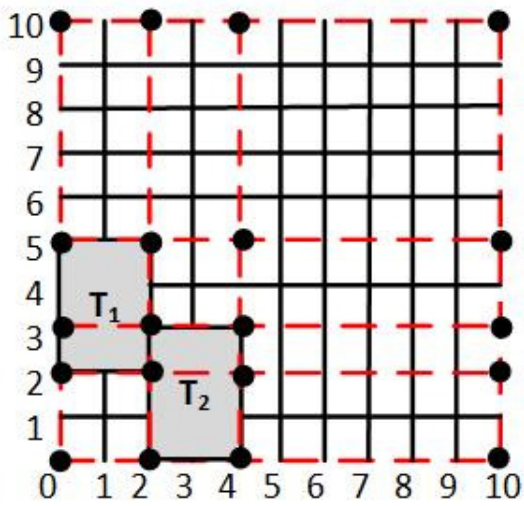


Fig. 5.5: horizontal and vertical sweeping lines

of these sweeping lines (black dotted marker in Fig.5.5) form the vertices of candidate rectangles. Even though the candidate rectangles contain both the occupied and the empty rectangular regions, NOR algorithm extract the empty ones. It is to be noted that rectangles are represented with their corresponding bottom left corner (BLC) and top right corner (TRC).

The listing in NOR (PB) illustrates the complete algorithm. The algorithm starts by drawing the horizontal and vertical sweeping lines along the edges of placed tasks and FPGA boundaries. These horizontal and vertical lines are sorted along the y-axis and x-axis respectively. Then, only the empty rectangles are extracted and stored in a graph. In the graph, nodes represent empty rectangles, edges refer to the neighbor rectangles, and weight is the length of the overlap line segment. The final output of the NOR algorithm is the adjacency list representation of the empty rectangles.

***Example(continued)*:** Let us consider the same real-time task graph depicted in table 5.1 and shown in fig. 5.1. The floor size of the FPGA is assumed to be 12×24. The end-to-end deadline of the task graph is $D_G = 12$. Initially, the FPGA was empty (i.e., PB = NULL) and the scheduler sends T_1^1, T_2^1, T_3^1 and T_4^1 sequentially for placement.

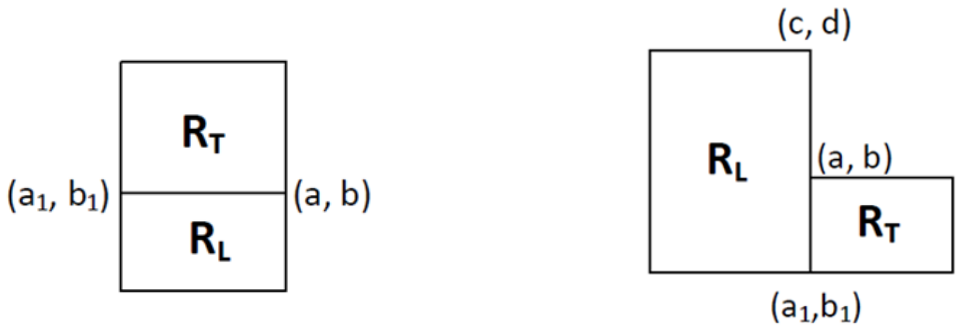
Since there is no placed task on the FPGA (i.e., PB = NULL), the function *DrawSweepingLines()* returns the horizontal and vertical sweeping lines along the boundaries of the FPGA. Sorting these lines respectively on y-axis and x-axis resulted in HL= {(0,0)(10,0), (0,10)(10,10) } and VL= {(0,0)(0,10), (10,0)(10,10) }. Rectangle formation is carried out

Algorithm 6: NOR (PB)

Input: PB: Set of placed tasks (blocks)
Output: Weighted forest graph of non-overlapping empty rectangles

```
1  /*function DrawSweepingLines(PB)- Draws a horizontal and vertical sweeping lines
   along the four line segments of the placed tasks and edges of the FPGA, from left to
   right and bottom to top with reference to the FPGA boundaries.*/
2  DrawSweepingLines(PB);
3  /*Let HL represent an array of sorted horizontal lines and m is the corresponding
   cardinality (i.e. cardinality(HL)) */
4  /*Let VL represent array of sorted vertical lines and n is the corresponding cardinality
   (i.e. cardinality(VL)) */
5  /*function FormRectangle() - return the left bottom and right top coordinates of the
   corresponding rectangle*/
6  for i = 0 to m − 2 do
7      for j = 0 to n − 2 do
8          /* Let R be the newley formed rectangle*/
9          R = FormRectangle(HL[i],VL[j]);
10         /*Region_Overlap() return TRUE if the considered rectangle has a shared region
           against any element of set PB, FALSE otherwise*/
11         for each p ∈ PB do
12             if Region_Overlap(R, P) == TRUE then
13                 Discard R(= NULL);
14         if (R ≠ NULL) then
15             CreateNode (graph, R)/* create node R and add it to the graph*/
16             /*function GetNeighbourRectangle() return a set of all possible neighbour
               rectangles of R (i.e.,NB), NULL if no single neighbour found*/
17             /*function length_overlap() compute the length of overlap line segments of
               two neighbour rectangles*/
18             /*function AddEdge(R, N, wt)-add an edge between R and its neighbour N*/
19             NB = GetNeighbourRectangle(HL[i], VL[j]);
20             for each N ∈ NB do
21                 for each p ∈ PB do
22                     if Region_Overlap(N, P)==TRUE then
23                         Discard N(= NULL);
24                 if N ≠ NULL) then
25                     wt= length_overlap(R, N);
26                     CreateNode(graph, N)
27                     AddEdge(R, N, wt);
28 return(Graph);
```

by the function *FormRectangle()* which resulted in R= (0,0)(12,24). Next, R is checked for emptiness using the function Region_Overlap (R, PB). Since PB = NULL, rectangle R is kept. In line (15), the function *CreateNode()* creates the rectangular node for R. The *GetNeighbourRectangle()* function return NULL because rectangle R has no neighbour. The final output of the algorithm is a single node, R.



(a) Complete Overlap Side

(b) Partial Overlap Side

Fig. 5.6: Ovelap Sides

The Overlap Sides and Rectangle Merging

Overlap sides are classified into two: the complete overlap side and the partial overlap side. The complete/fully overlap side is a line segment, where its endpoints are the corners of the neighboring rectangles. Fig.5.6a illustrates the fully overlapping sides for empty rectangles R_T and R_L. The endpoints (a_1, b_1) and (a, b) are the corners of both rectangles R_T and R_L, and therefore the line segment $(a_1, b_1)(a, b)$ is a complete overlap side. Whereas, the partial overlap side as shown in fig.5.6b is a side that is partially shared between two neighboring rectangles. $(a_1, b_1)(a, b)$ is the partial overlap line segment between rectangle R_T and R_L. Rectangle merging is the process of converting two neighboring rectangles into one. The BLC of the lower or left rectangle (i.e., R_L from fig 5.6) and the TRC of the top or right rectangle (i.e., R_T from fig 5.6). The maximal empty rectangle (MER) is constructed in this way. The MER is an empty rectangle that cannot be covered fully by any other empty rectangle.

Algorithm 7: COS (Graph)

Input: Adjacency list representation of NORs
Output: A subgraph with all complete overlap sides removed

```
1  /*Let List_Rect and NOR be sorted arrays of rectangles in descending order of the
   complete overlap sides*/
2  /* Initially, List_Rect = all nodes of forest graph, and NOR = NULL */
3  while (List_Rect ≠ NULL) do
4      Add List_Rect[0] in NOR;
5      while (NOR ≠ NULL) do
6          R_T = NOR[0]; /* get a rectangle with the longest complete overlap side*/
7          for every unvisited neighbour of R_T do
8              Add unvisited neighbour into NOR;
9          /*Let NB be the set of neighbour rectangles for R_T.*/
10         /*function EqualOverlapEdge(R_T, NB)- return a neighbour rectangle that share
           the same and equal longest complete overlap side with R_T*/
11         R_L= EqualOverlapEdge(R_T, NB);
12         /*let MR represents the merged rectangles*/
13         /* function AdjustGraph(R_T, R_L, MR)- Update graph structure for R_T, R_L and
           MR, i.e., delete edges, add edges and weights as demanded */
14         /*RemoveNodes(R_T, R_L)- remove the vertices R_T and R_L from the Graph*/
15         if R_L ≠ NULL then
16             AdjustGraph(R_T, R_L, MR);
17             RemoveNodes(R_T, R_L);
18             Add MR into List_Rect and NOR;
19             Discard R_T and R_L from List_Rect and NOR;
20         else
21             Discard R_T from List_Rect and NOR;
22 return (Graph);
```

Complete Overlap Side (COS)

The COS algorithm identifies and collects the complete overlap line segments. Based on the length of these line segments, we obtain sorted rectangles. Rectangle merging commences from rectangles that have the longest overlap side. The algorithm passes through two while loops; (i) while (List_Rect$\neq NULL$) makes sure that the nodes of a forest graph are visited. (ii) while (NOR$\neq NULL$) checks whether all nodes of the currently considered connected subgraph of a forest are visited or not. In line (6) the algorithm gets a rectangle with the longest complete overlap side (say R_T = NOR[0]). NOR stores

the rectangles in descending order of their complete overlap side length. In lines ([7 - 8]) the COS algorithm stores the unvisited neighbors of R_T into NOR. If R_T has multiple neighbors, the function *EqualOverlapEdge()* return a neighbor (say R_L) that has the longest complete overlap side. Once R_L is obtained, statements in lines ([16-19] are executed which are summarized as follows; merge R_T and R_L (i.e., MR), adjust graph structure, and update the different data structures accordingly. The process terminates when List_Rect = NOR = NULL.

Example continue: The NOR algorithm constructs a weighted forest graph. The COS algorithm proceeds on merging for the available complete overlap sides. Initially, List_Rect = {R} and NOR = {$\emptyset$}. In line (4), NOR = List_Rect[0] = {R}. In line (6) we have R_T = NOR[0] = R. currently, the only available rectangle is R_T = {R}, therefore, statements in lines ([7-8]) are not executed. Consequently, the function *EqualOverlapEdge()* return NULL (i.e., R_L = NULL). Now, the algorithm discards R from List_Rect and NOR at the line (21). Thus, List_Rect = NOR = NULL, and the algorithm terminates.

Partial_Overlap_side(POS)

POS algorithm computes the final set of MERs. Similar to the COS algorithm, POS also collects all the available partial overlap sides and creates sorted rectangles based on their length. As the listing POS () algorithm illustrates, line ([1-3]) removes all partial overlap sides that have edge weight below a predefined THRESHOLD. This enables the algorithm to give more importance to the longest partial overlap segments that might have brought a larger area benefit after falling apart and re-merging. Statements in lines ([5-11]) are the same as the COS algorithm discussed above. The function *MaxEdgeWt()* in line (12) determines the longest partial overlap line segment shared between a rectangle (say R_T) and its possible neighbors. The value returned by this function is assigned to the rectangle R_L. When $R_L \neq NULL$, the function *RegionFormation()* create a new region by drawing horizontal and vertical guillotine cut lines through R_T and/or R_L starting from their overlapped line segment endpoints to the opposite side as shown in fig. 5.7 and 5.8. As indicated in fig. 5.7b and 5.8b the new region NR (i.e., $\{NR_1, NR_2\}$) and the additional remaining fragmented regions (RM_1 and/or RM_2) are returned by the function *RegionFormation()*.

From the predefined THRESHOLD value and the area merging criterion, the POS

(a) Before (b) After

Fig. 5.7: New region formation through a horizontal guillotine cut lines

(a) Before (b) After

Fig. 5.8: New region formation through a vertical guillotine cut lines

algorithm determines whether to proceed with merging the rectangles (i.e., merge NR_1 and NR_2) or keep the original ones (i.e., R_T and R_L). The merging criterion states that the newly created region must be greater than its constituent original rectangles by at least 120%. This criterion is set for two reasons; i) fragmenting an existing region and merging them anew is a costlier operation in terms of computation demand. ii) Most of the remaining regions after falling apart and re-merging would be smaller in size which might not be used by the subsequent placement requests. It is to be noted that for our experimental part, a THRESHOLD value of 20 and the partial merging criterion of 120% were considered. If the test, in line (18) is successful, there are several operations to be performed. The function *AddNeighbour()* searches the neighbors of RM_1 and RM_2 and adds edges accordingly. The graph structure is adjusted for the old neighbor rectangles of R_T and R_L and the newly created rectangle NR with *AdjustGraph()* function. *AddEdge()* constructs edges between NR and the remaining regions. And finally, the variables are updated with the newly formed region(NR), the remaining fragments(RM_1 and RM_2), and discard the rectangles R_T and R_L.

Example(continued): The POS algorithm proceed from where COS stopped. State-

Algorithm 8: POS (Graph)

Input: A weighted forest subgraph
Output: MERs: set of Maximal Empty Rectangles

```
1  for every edge in the graph do
2      if edge weight < THRESHOLD then
3          DeleteEdge(Graph, edge)/*remove edge*/
4  /*Let PAR and PNOR be sorted arrays of rectangles in descending order of partial
   overlap sides*/
5  /*Initially, PAR = nodes of forest graph, and PNOR = NULL */
6  while (PAR ≠ NULL) do
7      PNOR = PAR[0] and MERs = PAR[0];
8      while (PNOR ≠ NULL) do
9          R_T = PNOR[0]; /* get a rectangle with the longest partial overlap side*/
10         for every unvisited Neighbour of R_T do
11             Add the unvisited Neighbour into PNOR and MERs;
12         R_L = MaxEdgeWt(R_T, NB) /*return the maximum edge weight between R_T and
           its neighbours (i.e,. NB)*/
13         /* function RegionFormation(R_T, R_L) - Create a new regions by drawing a
           horizontal/vertical guillotine cut lines through R_L and/or R_T*/
14         /*Let Rg be a set that holds the new formed region (i.e., NR) and the remaining
           regions(i.e., RM_1 and RM_2) from the original rectangles*/
15         /*function AddNeighbours()- searches the nebours of the newely created
           remaining regions(i.e., RM_1 and RM_2) and add edges if it found one*/
16         if R_L ≠ NULL then
17             Rg = RegionFormation(R_T, R_L);
18             if Area(NR)> 120%×Area(R_T) && Area(NR)> 120%×Area(R_L) then
19                 AddNeighbours(R_T, R_L, RM_1);
20                 AddNeighbours(R_T, R_L, RM_2);
21                 AdjustGraph(R_T, R_L, NR);
22                 RemoveNodes(R_L, R_T);
23                 wt_1= length_overlap(NR, RM_1);
24                 AddEdge(NR, RM_1, wt_1);
25                 wt_2= length_overlap(NR, RM_2);
26                 AddEdge(NR, RM_2, wt_2);
27                 Add NR, RM_1 and RM_2 in to PAR, PNOR and MERs;
28                 Discard R_T and R_L from PAR, PNOR and MERs;
29         Discard R_T from PAR, PNOR;
30 return MERs;
```

ments in lines ([1-3]) skipped because the graph is edgeless. In line (7) we have PAR = MERs = PNOR[0] = {R}. In line (9) the POS algorithm gets a rectangle with the longest partial overlap side. That is, $R_T = PNOR[0] = R$). Since R has no neighbour, the subsequent statements in lines ([10-11]) are skipped. line (12) return NULL (i.e., $R_L = NULL$), as a result statements in lines ([19-28]) are not executed. R is discarded from PAR and PNOR in line (29). Finally, PAR = PNOR = NULL and the algorithm terminates.

Algorithm 9: MOP (MERs, T_j, PB)

Input:
i. MERs: set of Maximum Empty Rectangles.
ii.T_j: Placement Requesting task ($= w_j \times h_j$).
iii.PB: a set of placed tasks with BLC and TRC.
Output: Max_Value: a variable that hold three information (corner, Maximal overlap counter and Orientation) .

1 /*Let Max_Rect contains a Copy of MERs*/
2 /* let HPB be set of bottom and top line segments of all placed tasks including the FPGA boundaries*/
3 /* let VPB be set of left and right line segments of all placed tasks including the FPGA boundaries*/

Maximal_Overlap_Placer(MOP)

The MOP algorithm determines a corner location with the highest number of overlap side counters. The overlap side counter at a corner of an empty rectangle is the sum of unit length overlap line segments between the requesting task's line segments and the boundaries of a placed and/or a FPGA. That is, we place the requesting task at a corner of a given MER and count the number of unit lengths shared between the currently placed requesting task and the previously placed tasks and/or FPGA boundaries. At each corner of the MERs, we consider two orientations; the horizontal orientation (HO = $w_j \times h_j$) and the vertical orientation (VO = $h_j \times w_j$).

The algorithm creates a set called Max_Set that stores three pieces of information (corner, overlap_counter, orientation) for each maximal empty rectangle. Where, corner refers to a vertex of a MER with the maximum overlap counter(i.e., out of the four corners), the overlap_counter is the number of unit length overlap line segments and orientation refers to either HO or VO. The *find_max()* function extracts the maximum overlap counter for

```
4  for each rectangle of Max_Rect do
5      /*Let sz1 and sz2 be task's sizes and Rw and Rh be currently considered
       empty rectangle's width and height*/
6      for each corner of a Rectangle do
7          /*let variable Morient hold the values returned by the function larger()
           initialized to NULL*/
8          for each Orientation do
9              if sz1 > Rw OR sz2 > Rh then
10                 not-placeable;
11                 continue;
12             /*Let Hline represent a set of horizontal unit length line segments that
               starts from a corner point and stepping towards left or right till it covers
               the size1 displacement/
13             for each hl1 ∈ HPB do
14                 for each hl2 ∈ Hline do
15                     if line_overlap(hl1, hl2)==TRUE then
16                         increment overlap_cnt;
17             /*Let Vline be a set of vertical unit length line segments that starts
               from the corner point and stepping upward or downward till it cover
               size2 displacement */
18             for each vl1 ∈ VPB do
19                 for each vl2 ∈ Vline do
20                     if line_overlap(vl1, vl2)==TRUE then
21                         increment overlap_cnt;
22             /*function larger() return the larger overlap counter with its respective
               Orientation*/
23             Mcorner = larger(overlap_cnt, Orientation );
24         /*Let Corner_best (CB) be a set that store a corner along with the best
           Mcorner obtained*/
25         CB = {(corner, Morient)};
26         Discard the occurence of (corner, NULL) from set CB
27     /*function find_max(CB) return values corresponding to max. overlap counter
       of set CB and assign */
28     /*Let Max_CB be a set that store the maximum overlap counter of given
       orientation for each corner*/
29     Max_CB = find_max(CB);
30     /*let Max_Set be a set that holds the best of each considered rectangle*/
31     Update max_Set with Max_CB;
32 Max_Value = find_max(Max_CB);
33 return Max_Value;
```

each corner location and store it in a variable Max_CB. Max_Set is a set that contains the maximum overlap counter of each rectangle (after comparison among the Max_CB). Finally, the maximal overlap counter is chosen and saved in a variable Max_Value which is returned by the algorithm.

Table 5.3: overap counter

Max_REC	corner	overlap counter	
		HO	VO
R	(0, 0)	44	0
	(12, 0)	44	0
	(0, 24)	44	0
	(12, 24)	44	0

***Example(continued)*:** MERs ={R}; $T = T_1^1 = 12X16$; line segments for $PB = NULL$ and line segments for FPGA={(0,0)(11,0), (0,0)(23,0), (11,0)(11,23), (0,23)(11,23)}. Decoupling the line segments in to horizontal(i.e., HPB) and vertical(VPB), we obtain; HPB = (0,0)(12,0), (0,24)(12,24) and VPB = (0,0)(24,0), (12,0)(12,24). Max_Rect = MERs = {R}, where R = (0,0)(12,24),with corresponding corners (0,0), (12,0),(0,24) and (12,24). Placing BLC of the requesting task at each corner of R, we obtain the results as shown in the table 5.3. As the table shows the maximum overlap counter obtained by the requesting task is 44 in horizontal orientation (HO). whenever an equal number of overlap counters is obtained, the MOP algorithm usually prefers to place tasks starting from the leftmost corner. Therefore, ((0,0), 24, HO) is stored in Max_Value for rectangle R. Additionally, the table illustrates that the overlap counter in VO is zero. The reason is that size of the task is greater than the width of the FPGA, and thus no feasible placement is generated.

T_2^1 was the second task in the sequence sent by the scheduler after the successful placement of T_1^1. The placer has to re-run the four algorithms to decide whether T_2^1 can be placeable or not. Accordingly, the placer returns NULL because T_2^1 and T_1^1 cannot be placed simultaneously in the available FPGA area.

5.3 Experiments and Results

In this section, we evaluate the performance of our proposed heuristic algorithms presented above.

5.3.1 Experimental Setup

Performance evaluation of the proposed heuristic algorithms has been carried out through comprehensive sets of simulation-based experiments by considering randomly generated DAGs. The principal metrics based on which evaluations have been carried out are as follows;

(i) *Normalized Achieved Reward (NAR in %)*:

$$NAR = \frac{\sum_{i=1}^{n} REW_i}{\sum_{i=1}^{n} REW_i^{k_i}} \times 100 \tag{5.2}$$

where $\sum_{i=1}^{n} REW_i$ denote the total reward obtained on successful execution completion of the DAG and $\sum_{i=1}^{n} REW_i^{k_i}$ is the maximum possible achievable reward, which is the sum of rewards for the highest versions. As referenced in [65], for an application which consists of (η) number of tasks, width(w), heght(h), execution time(e) and deadline (D_{app}), the load factor or utilization ratio of an application on the FPGA size of W and H is given by,

$$APP_{load} = \frac{\sum_{i=1}^{\eta} w_i \times h_i \times e_i}{W \times H \times D_{app}} \tag{5.3}$$

Let w_{avg}, h_{avg} and e_{avg} be the average width, height and execution time respectively for a given task sets, then equation 5.3 can be re-written as,

$$n = APP_{load} \times \left(\frac{D_{app} \times \left\lfloor \frac{W}{w_{avg}} \right\rfloor \times \left\lfloor \frac{H}{h_{avg}} \right\rfloor}{e_{avg}} \right) \tag{5.4}$$

where n is the number of tasks nodes obtained for a predfined APP_{load} value.

(ii)*Schedule length ratio (SLR) :* The schedule length ratio is given by equation (5.5).

$$SLR = \frac{makespan}{D_{dag}} \tag{5.5}$$

where, *makespan* is the DAGs maximum execution time completion and D_{dag} is the deadline of the whole task graph.

(iii) *Utilization (U) :* The utilization is given by equation (5.6). $W \times H \times makespan$

represents the total amount of resources available on the FPGA during the makespan.

$$U = \frac{\sum_{i=1}^{n} w_i \times h_i \times e_i}{W \times H \times makespan} \tag{5.6}$$

(iv) *Cummulative schedules execution time :* As stated in [65], this time is the runtime required by the scheduling algorithm (and the underlying placement) each time the scheduler is invoked. The $Sched\_exe\_time$ is expressed by equation (4.16), where n is the number of invocation of the scheduler and $Exec\_Time(i)$ the runtime required by the scheduling algorithm in its i^{th} call.

$$Sched\_exe\_time = \sum_{i=1}^{n} Exec\_Time(i) \tag{5.7}$$

Now various types of datasets have been generated by setting different values for the following parameters.

1. APP_{load}: The APP_{load} is varied between 40% and 90%.

2. *Floor size of the FPGA*: All our experiments have been conducted assuming the FPGA floor size to be 52×128 (same as the actual dimension of Xilinx Virtex-4 (XC4VFX60)).

3. *DAGs graph*: The number of DAG edges used in our experiment varies from $(n+4)$ for an APP_{load} value of 40% to $(n+9)$ for an APP_{load} of 90%, where n is the task nodes obtained from equation 5.4. In generating the DAGs, except for the source task node, every node has at least one predecessor.

4. *Task versions*: We used a similar approach for task version generation as mentioned in [65]. Versions represent the different hardware implementations for a given task.

   Task versions are denoted as Normal version(NV), Half Normal Version (HNV), and Twice Normal Version (TNV). We denote the HNV as a half-sized NV obtained by dividing the width(resp. the height) by two but requires a twice longer execution time and a half reward value. The TNV has twice the width (resp. the height) of NV, but requires a half shorter execution time and a twice reward value. In addition, the reconfiguration time is directly proportional to the area used as was assumed in the introduction section. It is assumed that each task could have at least one version (i.e., the NV) and at most three versions (i.e., NV, HNV, and TNV).

Therefore, task version assignment is taken from a uniformly distributed range of $[1, 4]$.

5. *Task size (Spatial Resource Demand)*: We have considered synthetic task sets of sizes varying from (16×16) and (32×32) for evaluating the performance of our heuristic algorithm as mentioned in [114]. Rectangular task dimensions have been obtained by separately generating widths (w_i) and heights (h_i) of the tasks from distributions having mean $\mu_{sz} = 8$ and standard deviation $\sigma_{sz} = 16$. For each task, the execution time and reconfiguration time ranges are taken from [114]. Accordingly, the execution time range is from 10 to 60-time units (ms), the reconfiguration time from $500\mu s$ to $600\mu s$, and the reward incurred varies in the range of 10 to 100. It is to be noted that only the NV parameter value is generated.

6. *Number of Task*: To evaluate the utilization of our proposed heuristics, we have considered the following number of task nodes; 10, 15, 20, and 25.

7. *Reconfiguration time*: We have considered varying the average reconfiguration time to be a predefined factor of the average execution time. Initially, the reconfiguration time was assumed to be negligible (i.e., download as many as possible number of tasks at each run). Next, we allow task downloading operations to be done at a predefined time factor of the execution time. Thus, the following reconfiguration time factors were considered in our experiments; $(ng, e/3, e/2,$ and $e)$, where e stands for the average execution time and ng denotes the negligibility of the reconfiguration time. It is to be noted that though the reconfiguration time for e was assumed to be in the same time units (i.e. ms) as that of execution, a small-time unit was initialized for it compared to the execution time.

The data points shown in the charts are obtained as the average over 10 runs on different DAG instances with a chosen set of parameters. In the formulas we have considered above, the reconfiguration time is included as a part of the execution time.
We implemented the algorithms in C language and the data set generation through Matlab. The simulation was performed on Intel(R) Core(TM) i5-1035G1 CPU @1.00GHZ 1.19GHz and 8GB installed memory(RAM).

5.3.2 Results

Figure 5.9 shows the normalized reward obtained by the proposed heuristics with varying APP_{load}. From the figure, it can be observed that the normalized reward remains comparable with the changes of APP_{load}. This may be attributed to the fact that when the APP_{load} is increasing the number of tasks also increases, thus equation (5.2) remains more or less the same. These results, therefore, indicate that the achieved reward may be considered to be robust against variation of APP_{load}.

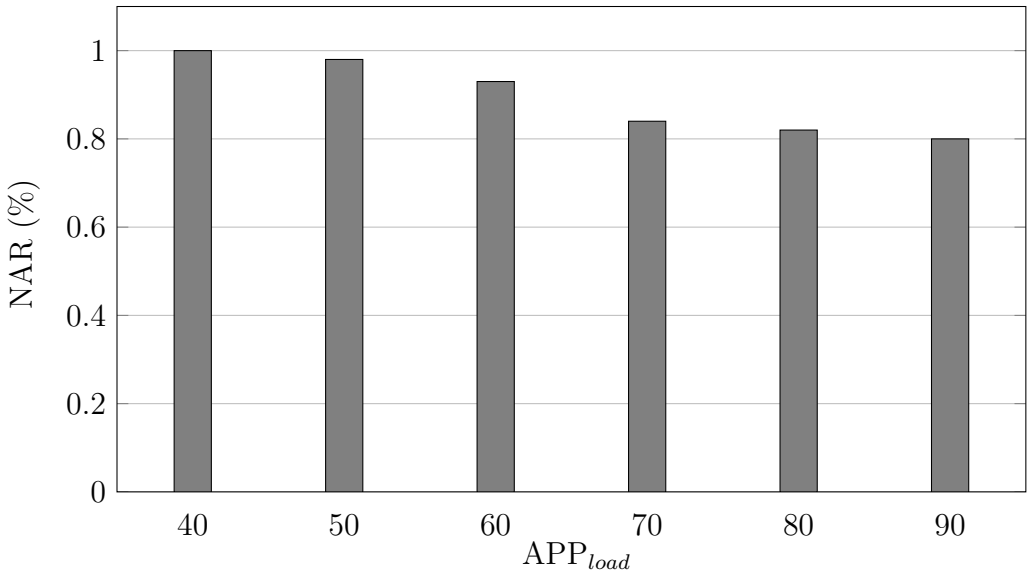


Fig. 5.9: NAR vs. APP$_{load}$

Figure 5.10 shows the schedule length ratio(SLR) obtained by the proposed heuristics with varying APP_{load}. The figure illustrates that the SLR increases with increasing APP_{load}. This is because, as the APP_{load} increases, the feasible schedules are obtained from the lowest task version combinations. As task versions decrease, the execution time increases, and as a result, the makespan time also increases. This result conforms to equation 5.10.

Figure 5.11 shows the utilization of our placement heuristic with the varying number of tasks. The figure illustrates that the utilization increases as the number of tasks increases. The rate of utilization is 7.3 %, 7.5%, and 4.1% when the number of tasks varies from 10 to 15, 15 to 20, and 20 to 25. The rate is comparable in the first two instances while it decreases afterward. This decrement is attributed to the fact that not only does the number of tasks increase, but also their dependencies become complex. This result in a

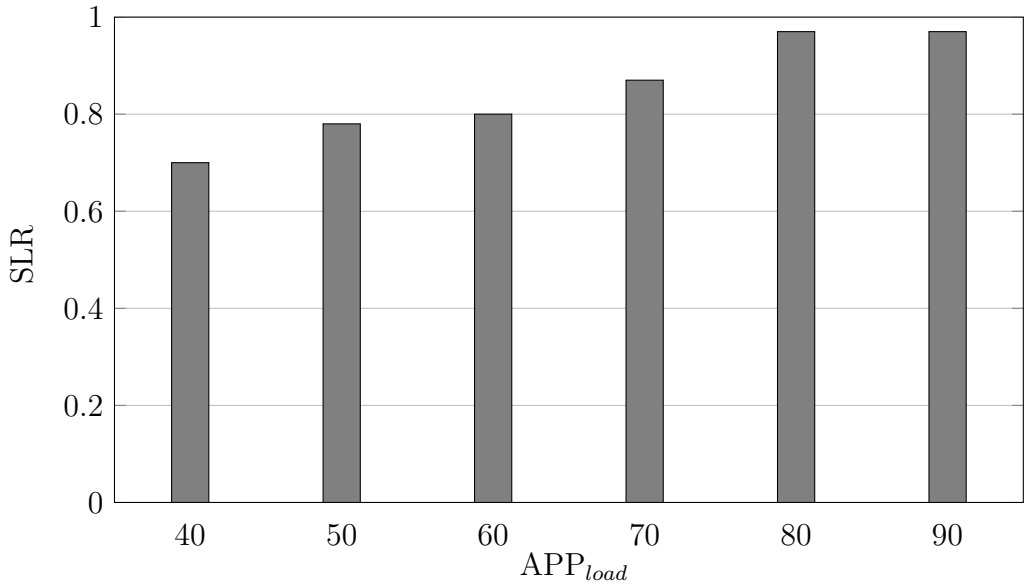


Fig. 5.10: SLR vs. APP$_{load}$

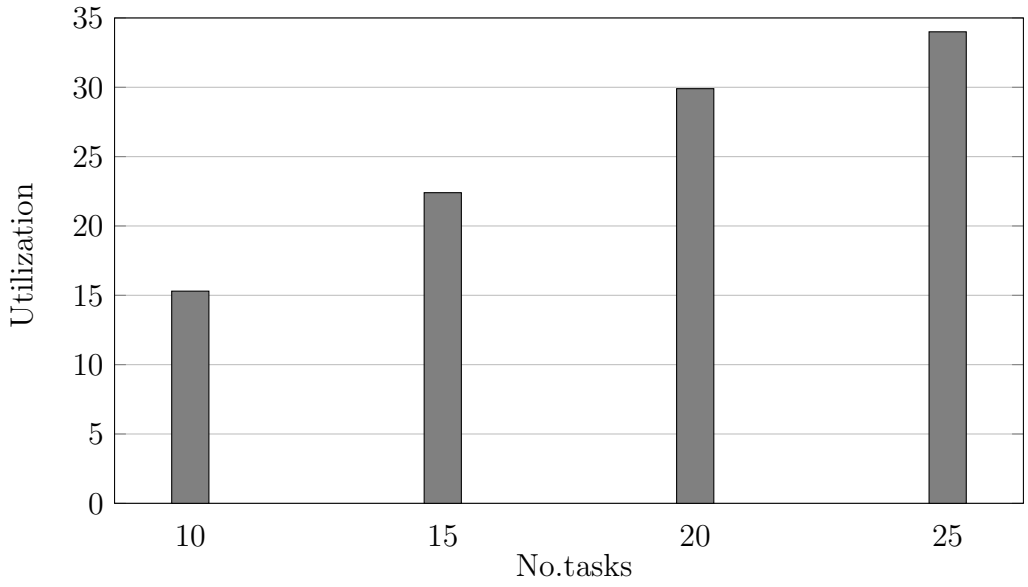


Fig. 5.11: Utilzation vs. No.tasks

high probability of an infeasible schedule.

Figure 5.12 shows the effect of the reconfiguration time on normalized achieved rewards(NAR). As the reconfiguration time factor varies from ng to e, the NAR continuously decreases. This is due to the fact that as the reconfiguration time increases, the possibility of generating a feasible schedule would be from the lowest task version combinations. For task number = 20 and reconfiguration factor(= e) the algorithm generates

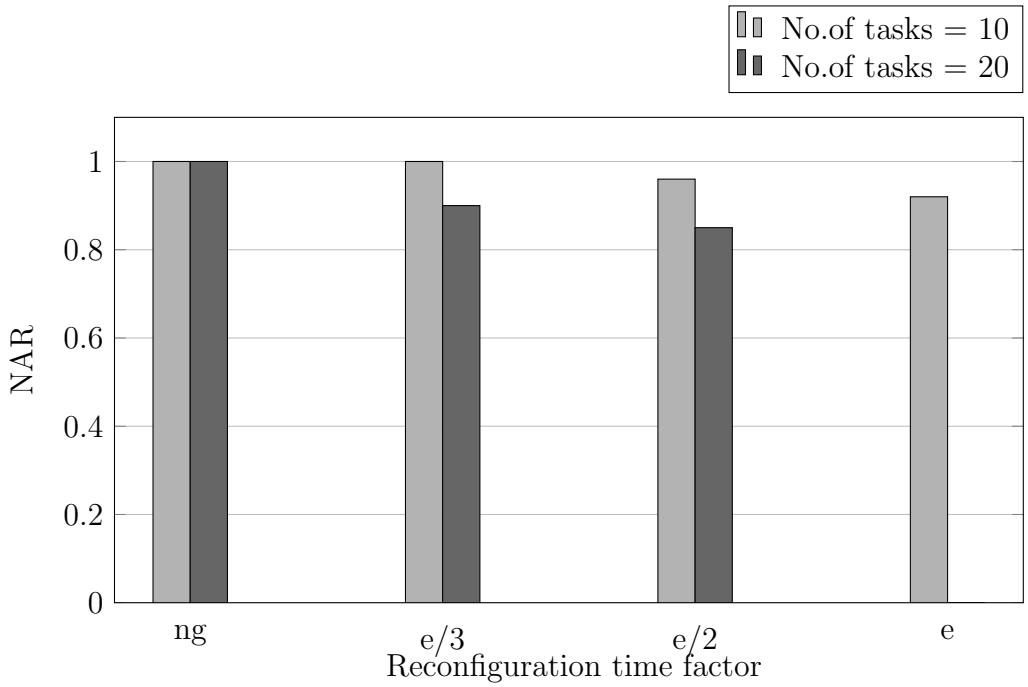


Fig. 5.12: NAR vs.Reconfiguration time factor

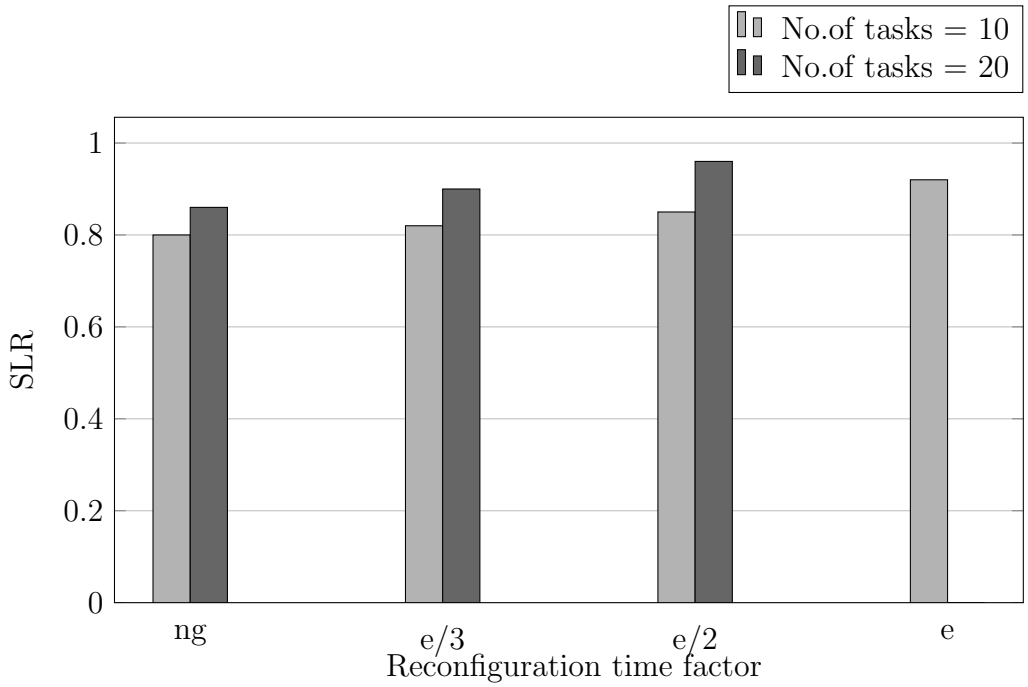


Fig. 5.13: SLR vs.Reconfiguration time factor

all infeasible schedules as demonstrated in figure 5.12.

Figure 5.13 shows the effect of reconfiguration time on schedule length ratio(SLR).

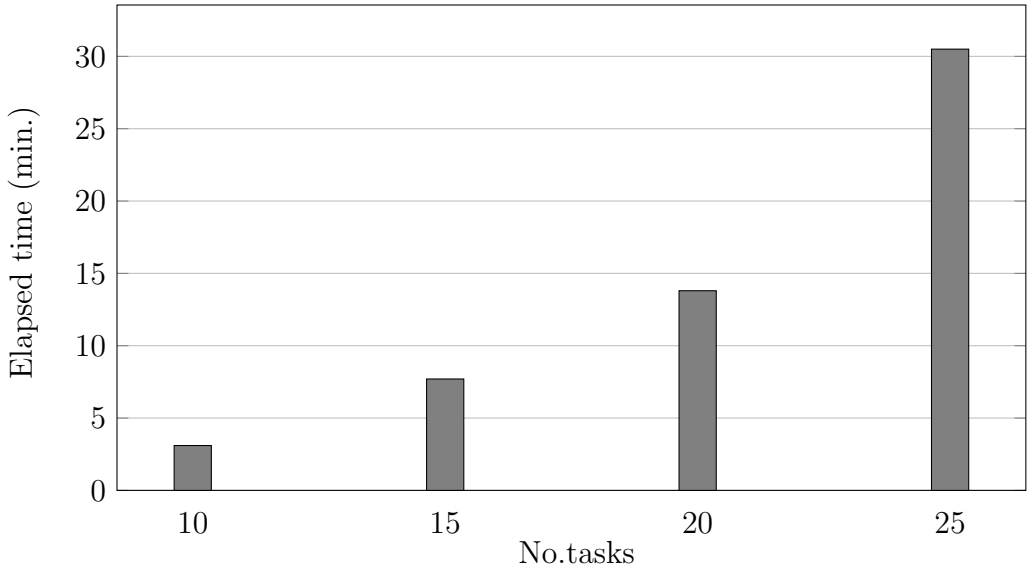


Fig. 5.14: Elapsed time vs. No.tasks

The SLR value increases as the reconfiguration time factor varies from ng to e. As can be seen from the figure, for task number = 20, the SLR value increases and finally becomes greater than one (i.e., $SLR > 1$) which is classified as an infeasible schedule. The SLR value for the infeasible schedule is omitted from the chart as shown in figure 5.13.

Figure 5.14 shows the performance of the overall execution time produced by our proposed heuristic algorithm. The figure illustrates that the cumulative execution time increases as the number of tasks increases.

5.4 Summary

In this chapter, we have presented an offline scheduling strategy for real-time precedence-constrained task graphs consisting of multi-mode safety-critical tasks where each mode is characterized by distinct spatial and temporal resource demands, reward obtained by the system on successful execution, and possibly accuracy of results produced. We have also considered the cases where the reconfiguration time varies in our heuristics algorithm. Simulation-based experimental results reveal that our heuristics algorithm can achieve an optimal solution with moderate computational overheads.

Chapter 6

Conclusions and Future Perspectives

6.1 Summarization

FPGA-based platforms are increasingly being looked upon as a lucrative and cheap alternative for executing many real-time safety-critical applications. Given an application, effectively organizing the executions of the application tasks while satisfying all timing constraints, is ultimately a scheduling problem. In this context, the schedule acts as a vital design component that determines an appropriate co-execution order for the application tasks such that the desired performance objectives may be achieved while satisfying all constraints. This thesis presented three static offline scheduler design approaches for reconfigurable systems: (i) a formal scheduler synthesis framework for the real-time tasks executing on an FPGA platform, using supervisory control of timed discrete event systems as the underlying formalism. (ii) an ILP-based solution strategy for scheduling persistent real-time applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs and (iii) a heuristic solution methodology for scheduling persistent real-time applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs.

The use of reconfigurable logic within the field of computing has increased during the last decades. The ability to change hardware during the design process enables developers to lower the time to market and reuse designs in several different products. Dynamic reconfiguration requires making decisions about the choice of new configurations and this may heavily depend on factors such as (i) the sequence of events during a particular run (ii) decisions on the relative execution order of a set of functionalities, over time (iii) predictive knowledge about the behavior of the functionalities (iv) Placement decisions corre-

sponding the functionalities, on the available reconfigurable real-estate (v) reconfiguration overheads, etc. Efficient scheduling decisions related to the selection of new configurations are therefore a very complex design issue because of the sheer exponential nature of the combinatorics of possible choices and are difficult to accomplish online. Hence, Offline formal approaches towards the design of reconfiguration controllers/schedulers are often a lucrative alternative that ensures designs that are correct by construction as well as optimal in terms of usage of resources. Scheduler synthesis based on the SCTDES framework starts with modeling the individual components and their associated constraints. Such model-based design of real-time hardware tasks with FPGAs as the processing platform is very complex and challenging. Chapter 3 of this thesis presents the scheduler synthesis scheme for a set of non-preemptive periodic real-time tasks along with the implementation details.

In certain applications, computational activities cannot be executed in an arbitrary order but have to respect some precedence relations depending on the design stage. Such precedence relations are usually described through a directed acyclic graph G, where tasks are represented by nodes and each task has a distinct execution time. Precedence relations are represented by the edges among nodes i.e output of a given node becomes the input of a node. In many complex applications, there exist such dependent tasks which require to be executed multiple times within a given period. For example, a task within a real-time control system takes input from the environment and then actuates after some processing. Such task graphs often require to be executed repeatedly within a given period. The execution of such dependent task graphs on FPGAs becomes a more complex problem as the hardware tasks (tasks bitstream) require to be loaded before the start of the execution. Modern FPGAs contain only a single loading channel known as ICAP/PCAP. Thus, no two tasks can simultaneously be loaded through that port. Hence, this resource constraint adds some extra complexities.

Having a complex task graph, it could be the case that we cannot simultaneously place all the task nodes within the finite area of a FPGA. However, through a proper scheduling and placement strategy, the task graph can be efficiently scheduled by maintaining the precedence constraints among nodes. Scheduling and partitioning of task graphs on reconfigurable hardware need to be carefully carried out to achieve the best possible performance. Tasks must be managed efficiently in time and space to exploit the advantages offered by the FPGA. A Spatio-temporal scheduling algorithm needs to find the earliest starting time for arriving tasks (when) and the region to accommodate (where) them. To

achieve the objective the scheduler must be able to handle all the following requests:

- what task to schedule at what time (temporal reconfiguration) ?
- Which version of the tasks to be loaded?
- When to load a particular version of task on FPGA through ICAP?
- where to place the task (spatial reconfiguration) ?
- when to start the execution of a task according with its precedence constraints (temporal scheduling) ?

In chapters 4 and 5, we presented two solution strategies for scheduling persistent real-time applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs. Below we summarize the chapters presented in this thesis.

Firstly, a background on real-time systems, supervisory control of timed discrete event systems, and fundamental technologies and principles concerning FPGAs were introduced in Chapter 2. This started with the different real-time scheduling algorithms for uniprocessor and multiprocessor systems. Furthermore, key concepts of supervisory control of timed discrete event systems and the steps required to synthesize a supervisor were explained. After introducing the FPGAs' fundamental technologies, the dynamic and partial reconfiguration, and real-time hardware tasks, the discussion of the Spatio-temporal Scheduling of hardware tasks proceeded in this chapter. Finally, the chapter concluded with a survey of formal and heuristic scheduling of real-time tasks on FPGAs.

In Chapter 3, our first contributory chapter, we have presented a formal scheduler synthesis framework for a set of non-preemptive periodic real-time tasks executed on the FPGA platform, using supervisory control of timed discrete event systems as the underlying formalism. Although in recent years, there have been a few significant works dealing with model-based real-time hardware task scheduling, this is possibly the first work that addresses the scheduler synthesis problem using SCTDES for FPGA platforms. Our framework development starts with the presentation of the system model and the assumptions considered. The typical task execution on reconfigurable FPGA platforms, the task execution, and the resource and timing constraint models were explained in detail. The composite task and resource constraint models and the description of supervisor synthesis with an example are also discussed. An experimental proof-of-concept validation

of our synthesized scheduler has been performed by implementing two independent real-time tasks on the Atlys Spartan-6 FPGA platform.

In Chapter 4, our second contributory chapter, we presented an offline ILP-based solution strategy for scheduling persistent real-time applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs. The chapter starts by describing the system model and formalization of the precedence-constrained Spatio-temporal scheduling problem. Next, the ASAP and ALAP time allocation policies are discussed. Then, sets of binary decision variables are declared for the ILP formulation. The different constraints of the proposed ILP and its overall objective function are explained. We perform the simulation-based experiment, and the result obtained summurized and discussed in this chapter.

In Chapter 5, our third contributory chapter, we presented a novel heuristics algorithm for the real-time scheduling of precedence-constrained task graphs on partially reconfigurable FPGAs. The proposed solution strategy commences by stating the problem definition and describing the system model and the considered assumptions. Then, the scheduling and placement heuristics are explained in detail. Two scheduling algorithms (i.e., VDAS and MVDS) have been discussed. Similarly, the four spatial algorithms (i.e., NOR, COS, POS, and MOP) are explained thoroughly. An example was also provided to elaborate on the working principles of the algorithms. We perform the simulation-based experiment, and the result obtained summurized and discussed in this chapter.

In summary, the work conducted as part of this thesis deals with; (i) the development of an offline formal scheduler synthesis mechanism for a set of independent non-preemptive periodic tasks. The practical viability of this proposal was tested and verified with an existing Atlys development board from Digilent. (ii) an ILP and a heuristic algorithm for solving applications represented as precedence-constrained task graphs on partially reconfigurable FPGAs are presented. Simulation-based validation has been done for both approaches.

6.2 Future Works

In this section, we present some future perspectives.

- Verification of the individual TDES models for correctness: Given a system and its specification model, the supervisor synthesized using the SCTDES framework is

provably *correct-by-construction* [115]. However, this correctness property is based on the assumption that the designs of individual systems and specification models are sound and complete. Adhoc mechanisms cannot be employed to make such strong guarantees on the correctness of the individual models. Handcrafted models developed by system engineers based on their domain knowledge and experience may often be prone to design flaws.

To avoid the issues associated with individual models, we may apply automated verification techniques to identify and correct the presence of possible errors in the developed models. For this purpose, formal approaches such as model checking seem to be an attractive alternative. In order to apply model-checking, the properties of interest, the correctness of which we desire to check, must first be identified. Given these properties, model checking through the following three steps: (1) The model $\mathcal{M}$ must be specified using the description language of a model checker; (2) The specification language must then be used to code the properties and this will produce a temporal logic formula $\emptyset$ for each specification; (3) Run the model checker with inputs $\mathcal{M}$ and $\emptyset$. The model checker outputs *YES* if $\mathcal{M}$ satisfies $\emptyset$ (represented by $\mathcal{M} = \emptyset$) and *NO* otherwise; in the latter case, a counter-example in the form of a trace of the system behavior exhibiting the violation of the property is produced. The automatic generation of such counter traces is an important tool in the design and debugging of safety-critical systems. When a model-checking mechanism is applied to the handcrafter system and specification models, we can ensure the soundness and completeness of these models against our chosen properties of interest.

- Dynamic partial reconfiguration as a technique was shown both in its practical usefulness and its limitations. The use of DPR technology for complex problems is limited by the current lack of advanced tools in industry or academia. In this regard, we would like to work on the development of tools that support flexible 2D area models with merging capability. We purview that such a tool will resolve many problems associated with the practical implementations of Spatio-temporal scheduling problems on FPGAs.

- Design of a seamless context switch/preemption mechanism is another open direction for future research in this area. Task contexts are stored in state-holding elements like Flip-flops and LUT-RAMs of CLBs and other Hard Blocks (BRAMs, MULs, etc.). Switching context in any specific region of the FPGA involves: (i)

capturing the contexts of tasks that were executing in the region prior to a switch, (ii) updating the contexts of these captured tasks and saving them in external memory, (iii) forming a new bitstream comprising of tasks that should execute in the region subsequent to the switch, (iv) restoring the new bitstream in the region to re-initiate execution after preemption. Literature [15, 16, 17, 18, 19, 20]has shown that significant improvements in both context capture / extraction and context updation times may be obtained through a selective bitstream read-back and context manipulation mechanism. These improvements have been able to effect drastic reductions in overall context switch overheads from more than 10ms to hundreds of microseconds to a few milliseconds. Such low overheads have now made preemptive scheduling more affordable in partially reconfigurable systems. As future work, we would like to design dynamic preemptive scheduling strategies for periodic real-time task sets to be executed on run-time partially reconfigurable platforms.

Disseminations out of this Work

Conference Papers

1. Cherinet Kejela, Rajesh Devaraj, Arnab Sarkar and Sangeet Saha "A Supervisory Control Approach for Scheduling Real-time Periodic Tasks on Dynamically Reconfigurable Platforms" Euromico Conference on Digital Systems Design 2022 (Accepted).

2. Cherinet Kejela, Sangeet Saha and Arnab Sarkar "Real-Time Scheduling of Precedence Constrained Task Graphs on Partially Reconfigurable FPGAs" (Under review).

References

[1] W. M. Wonham, "Supervisory control of discrete-event systems ece 1636f/1637s 2009-10," 2010.

[2] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer, 2008.

[3] Xilinx, "Virtex-4 family overview," *Tech. Doc. DS112 (v2. 0)*, pp. 1–8, 2010.

[4] T. Hayashi, A. Kojima, T. Miyazaki, N. Oda, K. Wakita, and T. Furusawa, "Application of fpga to nuclear power plant i&c systems," in *Progress of Nuclear Safety for Symbiosis and Sustainability*. Springer, 2014, pp. 41–47.

[5] J. Jin, S. Lee, B. Jeon, T. T. Nguyen, and J. W. Jeon, "Real-time multiple object centroid tracking for gesture recognition based on fpga," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*. ACM, 2013, p. 80.

[6] S. Bhasin, S. Guilley, A. Heuser, and J.-L. Danger, "From cryptography to hardware: analyzing and protecting embedded xilinx bram for cryptographic applications," *Journal of Cryptographic Engineering*, vol. 3, no. 4, pp. 213–225, 2013.

[7] D. Theodoropoulos, G. Kuzmanov, and G. Gaydadjiev, "A 3d-audio reconfigurable processor," in *Proceedings of the 18th annual ACM/SIGDA international symposium on FPGAs*, 2010, pp. 107–110.

[8] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "A communication aware online task scheduling algorithm for fpga-based partially reconfigurable systems," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 65–68.

[9] M. Huang, V. K. Narayana, and T. El-Ghazawi, "Efficient mapping of hardware tasks on reconfigurable computers using libraries of architecture variants," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on.* IEEE, 2009, pp. 247–250.

[10] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, and J. Becker, "Operating system for runtime reconfigurable multiprocessor systems," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[11] Z. Sun, H. Zhang, and Z. Zhang, "Resource-aware task scheduling and placement in multi-fpga system," *IEEE Access*, vol. 7, pp. 163 851–163 863, 2019.

[12] Z. Guettatfi, M. Platzner, O. Kermia, and A. Khouas, "An approach for mapping periodic real-time tasks to reconfigurable hardware," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 2019, pp. 99–106.

[13] R. Devaraj, A. Sarkar, and S. Biswas, "Supervisory control approach and its symbolic computation for power-aware rt scheduling," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 787–799, 2018.

[14] D. Chen, J. Cong, and P. Pan, "Fpga design automation: A survey," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp. 139–169, 2006.

[15] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Field Programmable Logic and Applications, 2005. International Conf. on.* IEEE, 2005, pp. 223–228.

[16] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on.* IEEE, 2010, pp. 352–355.

[17] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada, "Comparison of preemption schemes for partially reconfigurable fpgas," *IEEE ESL*, vol. 4, no. 2, pp. 45–48, 2012.

[18] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable fpgas," *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1505–1508, 2016.

[19] K. D. Pham, E. L. Horta, and D. Koch, "Bitman: A tool and api for fpga bitstream manipulations," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 894–897, 2017.

[20] M. Eckert, D. Meyer, and B. Klauer, "Context save and restore of partial reconfiguration regions for xilinx fpgas," *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 5–12, 2019.

[21] V. Kizheppatt and S. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys*, vol. 51, pp. 1–39, 07 2018.

[22] A. Brandin and W. Wonham, M, "Supervisory control of timed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 39, pp. 329 – 342, 1994.

[23] S. Miremadi, B. Lennartson, and K. Akesson, "A bdd-based approach for modeling plant and supervisor by extended fnite automata," *IEEE Transactions on Control Systems Technology*, vol. 20, pp. 1421 – 1435, 2012.

[24] V. Kizheppatt and fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," vol. 51, no. 4, 2015.

[25] D. Koch, *Partial Reconfiguration on FPGAs Architectures, Tools and Applications*. Springer, 2013, vol. 153.

[26] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 37–44.

[27] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "Openpr: An open-source partial reconfigration toolkit for xilinx fpgas," in *International Parallel and distributed processing Symposium*. IEEE, 2011.

[28] T. Marconi, "Online scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays," *Computers & Electrical Engineering*, vol. 40, no. 4, pp. 1215–1237, 2014.

[29] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[30] P. Altenbernd, "Deadline-monotonic software scheduling for the co-synthesis of parallel hard real-time systems," in *Proceedings of the 1995 European conference on Design and Test*. IEEE Computer Society, 1995, p. 190.

[31] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms." 2004.

[32] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*. IEEE, 2006, pp. 322–334.

[33] D.-I. Oh and T. P. Bakker, "Utilization bounds for n-processor rate monotone scheduling with static processor assignment," *Real-Time Systems*, vol. 15, no. 2, pp. 183–192, 1998.

[34] M. Lopez, M, M. Garcia, J. Diaz, L, and D. Garcia, F, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Jun 2000, pp. 25–33.

[35] A. Srinivasan, P. Holman, and J. Anderson, "The case for fair multiprocessor scheduling," in *Proceedings of the 11th International Workshop on Parallel and Distributed Real-time Systems, Nice, France*, Apr 2003.

[36] M. Caccamo and G. Buttazzo, "Optimal scheduling for fault-tolerant and firm real-time systems," in *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*. IEEE, 1998, pp. 223–231.

[37] D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele, "Partial reconfiguration on fpgas in practice; tools and applications," in *ARCS Workshops (ARCS), 2012*, Feb 2012, pp. 1–12.

[38] W. Nie, K.-J. Lin, and S. D. Kim, "Capacity-based admission control for mixed periodic and aperiodic real time service processes," in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on.* IEEE, 2011, pp. 1–8.

[39] P. Tan, H. Jin, and M. Zhang, "A hybrid scheduling scheme for hard, soft and non-real-time tasks," in *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on.* IEEE, 2006, pp. 7–pp.

[40] T. Amari, H. Gharsellaoui, M. Khalgui, and S. B. Ahmed, "Aperiodic os tasks scheduling for hard-real-time reconfigurable uniprocessor systems," in *Proceedings of the International Conference on Embedded Systems and Applications (ESA).* The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, p. 1.

[41] J.-y. Yin and G.-c. Guo, "An algorithm for scheduling aperiodic real-time tasks on a static schedule," in *Inf. and Comput. Sc., 2009. Second International Conference on*, vol. 1. IEEE, 2009, pp. 70–74.

[42] H.-K. Tang, P. Ramanathan, and K. Compton, "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources," in *Parallel Processing (ICPP), 2011 International Conference on.* IEEE, 2011, pp. 753–762.

[43] J. Lee, S. Lee, and H. Kim, "Scheduling of hard aperiodic tasks," in *ACM SIGPLAN Notices*, vol. 30, no. 11. ACM, 1995, pp. 7–19.

[44] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *RTSS, 1995. Proceedings., 16th IEEE*, pp. 152–161.

[45] S. Schorr and G. Fohler, "Online admission of non-preemptive aperiodic tasks in offline schedules," *Proceedings WiP Session*, 2010.

[46] S. Kato, "Real-time scheduling of periodic and aperiodic tasks on multiprocessor systems," Ph.D. dissertation, Keio University, 2008.

[47] S. Sáez, J. Vila, and A. Crespo, "Firm aperiodic task scheduling in hard real-time multiprocessor systems," in *Real-Time Programming 2003 (WRTP 2003): A Proceedings Volume from the 26th IFAC/IFIP/IEEE Workshop, Łagów, Poland, 14-17 May 2003*. Elsevier Science Limited, 2003, p. 51.

[48] B. Andersson and C. Ekelin, "Exact admission-control for integrated aperiodic and periodic tasks," *Journal of Computer and System Sciences*, vol. 73, no. 2, pp. 225–241, 2007.

[49] S.-J. Park and K.-H. Cho, "Supervisory control for fault-tolerant scheduling of real-time multiprocessor systems with aperiodic tasks," *International Journal of Control*, vol. 82, no. 2, pp. 217–227, 2009.

[50] M. W. Wood, "Application, implementation and integration of discrete-event systems control theory," 2005.

[51] R. P. J and W. M. Wonham, "The control of discrete event systems," in *Proceedings of the IEEE*. IEEE, 1989.

[52] S. Jovanovic, C. Tanougast, and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for fpga-based reconfigurable systems," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. IEEE, 2007, pp. 358–364.

[53] K. Danne and M. Platzner, "Periodic real-time scheduling for fpga computers," in *Intelligent Solutions in Embedded Systems, 2005. Third International Workshop on*, May 2005, pp. 117–127.

[54] Y. Chen and P. Hsiung, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc," *Real-Time Systems*, vol. 23, no. 1-2, pp. 489–496, 2006.

[55] J. Tabero, J. Septién, H. Mecha, and D. Mozos, "A low fragmentation heuristic for task placement in 2d rtr hw management," in *FPL*. Springer, 2004, pp. 241–250.

[56] C. Steiger, H. Walder, M. Platzner, and L. Thiele, "Online scheduling and placement of real-time tasks to partially reconfigurable devices," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE, 2003, pp. 224–225.

[57] P.-A. Hsiung, M. D. Santambrogio, and C.-H. Huang, *Reconfigurable System Design and Verification*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[58] C.-C. Chiang, "Hardware/software real-time relocatable task scheduling and placement in dynamically partial reconfigurable systems," Ph.D. dissertation, National Chung Cheng University, Taiwan, 2007.

[59] L. Chen, T. Marconi, and T. Mitra, "Online scheduling for multi-core shared reconfigurable fabric," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 582–585.

[60] Y.-H. Chen and P.-A. Hsiung, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc," in *Embedded and Ubiquitous Computing–EUC 2005*. Springer, 2005, pp. 489–498.

[61] M. Hubner, C. Schuck, and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 8–pp.

[62] M. Sanchez-Elez and S. Roman, "Reconfiguration strategies for online hardware multitasking in embedded systems," *arXiv preprint arXiv:1301.3281*, 2013.

[63] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The erlangen slot machine: A dynamically reconfigurable fpga-based," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 15–31, 2007.

[64] M. Natale, D and E. Bini, "Optimizing the fpga implementation of hrt systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007.

[65] G. Wassi, M. E. A. Benkhelifa, G. Lawday, F. Verdier, and S. Garcia, "Multi-shape tasks scheduling for online multitasking on fpgas," *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2014.

[66] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1, pp. 177–185, 1974.

[67] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE design & Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[68] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on fpgas: Free space partitioning and 2d-hashing," in *IPDPS '03 Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE, 2003, pp. 239–243.

[69] M. Esmaeildoust, M. Fazlali, A. Zakerolhosseini, and M. Karimi, "Fragmentation aware placement algorithm for a reconfigurable system," in *Electrical Engineering, 2008. ICEE 2008. Second International Conference on*. IEEE, 2008, pp. 1–5.

[70] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online fpga placement," in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 960–965.

[71] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for online placement on reconfigurable devices," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 134.

[72] S. J. Olakkenghil and K. Baskaran, "An fpga task placement algorithm using reflected binary gray space filling curve," *International Journal of Reconfigurable Computing*, vol. 2014, p. 5, 2014.

[73] M. Koester, M. Porrmann, and H. Kalte, "Task placement for heterogeneous reconfigurable architectures," in *Field-Prog. Tech., 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. 43–50.

[74] A. Eiche, D. Chillet, S. Pillement, and O. Sentieys, "Task placement for dynamic and partial reconfigurable architecture," in *(DASIP), 2010*. IEEE, 2010, pp. 228–234.

[75] Q.-H. Khuat, D. Chillet, and M. Hubner, "Considering reconfiguration overhead in scheduling of dependent tasks on 2d reconfigurable fpga," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*. IEEE, 2014, pp. 1–8.

[76] Y. Lu, "Realistic online resource management for partially reconfigurable systems," 2011.

[77] N. Guan, Q. Deng, Z. Gu, W. Xu, and G. Yu, "Schedulability analysis of preemptive and nonpreemptive edf on partial runtime-reconfigurable fpgas," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 4, p. 56, 2008.

[78] Z. Guettatfi, O. Kermia, and A. Khouas, "Over effective hard real-time hardware tasks scheduling and allocation," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–2.

[79] F. Dittmann and M. Gotz, "Applying single processor algorithms to schedule tasks on reconfigurable devices respecting reconfiguration times," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 4–pp.

[80] K. Danne and M. Platzner, "An edf schedulability test for periodic tasks on reconfigurable hardware devices," in *ACM SIGPLAN Notices*, vol. 41, no. 7. ACM, 2006, pp. 93–102.

[81] J. W. S. Liu, *Real-Time Systems*, 1st ed. Prentice Hall, 2000.

[82] S. M. Lauzac, "On multiprocessor scheduling of preemptive periodic real-time tasks with error recovery," Ph.D. dissertation, University of Pittsburgh, 2000.

[83] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[84] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. IEEE, 2000, pp. 35–43.

[85] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø, "Using dynamic partial reconfiguration of fpgas in real-time systems," *Microprocessors and Microsystems*, vol. 61, pp. 198–206, 2018.

[86] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 1–12.

[87] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, "A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017, pp. 96–101.

[88] B. Seyoum, M. Pagani, A. Biondi, and G. Buttazzo, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 481–490.

[89] D. Casini, P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration," *Journal of Systems Architecture*, vol. 124, p. 102416, 2022.

[90] J. Goossens, X. Poczekajlo, A. Paolillo, and P. Rodriguez, "Acceptor: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 209–219.

[91] G. Valente, T. Di Mascio, L. Pomante, and G. D'Andrea, "Dynamic partial reconfiguration profitability for real-time systems," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 102–105, 2020.

[92] S. Saha, A. Sarkar, and A. Chakrabarti, "Spatio-temporal scheduling of preemptive real-time tasks on partially reconfigurable systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 4, p. 71, 2017.

[93] J. Vidal, F. De Lamotte, G. Gogniat, J. P. Diguet, and P. Soulard, "Uml design for dynamically reconfigurable multiprocessor embedded systems," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. European Design and Automation Association, 2010, pp. 1195–1200.

[94] I. Quadri, H. Yu, A. Gamatie, E. Rutten, S. Meftali, and J. Dekeyser, "Targeting reconfigurable fpga based socs using the uml marte profile: From high abstraction levels to code generation," *Int. J. Embedded Syst*, vol. 4, pp. 204 – 224, 2010.

[95] G. Labiak, M. Wegrzyn, and A. Munoz, "State based design controllers for fpga partial reconfiguration," *International society for optics and photonics*, vol. 9662, p. 96623Q, 2015.

[96] J. Aylward, C. H. Crawford, K. Inoue, S. Lekuch, K. Muller, M. Nutter, H. Penner, K. Schleupen, and J. Xenidis, "Reconfigurable systems and flexible programming for hardware design, verification and software enablement for system-on-a-chip architectures," in *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2011, pp. 351–356.

[97] L. Gong and O. Diessel, "Modeling dynamically reconfigurable systems for simulation-based functional verification," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 9–16.

[98] S. Singh and C. J. Lillieroth, "Formal verification of reconfigurable cores," in *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00375)*. IEEE, 1999, pp. 25–32.

[99] X. An, E. Rutten, J.-P. Diguet, and A. Gamatié, "Model-based design of correct controllers for dynamically reconfigurable architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 3, p. 51, 2016.

[100] R. Wiśniewski, "Dynamic partial reconfiguration of concurrent control systems specified by petri nets and implemented in xilinx fpga devices," *IEEE Access*, vol. 6, pp. 32 376–32 391, 2018.

[101] R. Wiśniewski, G. Bazydło, L. Gomes, and A. Costa, "Dynamic partial reconfiguration of concurrent control systems implemented in fpga devices," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1734–1741, 2017.

[102] S. Guillet, F. d. Lamotte, N. l. Griguer, É. Rutten, G. Gogniat, and J.-P. Diguet, "Extending uml/marte to support discrete controller synthesis, application to reconfigurable systems-on-chip modeling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, p. 27, 2014.

[103] O. Diessel and H. Elgindy, "On dynamic task scheduling for fpga-based systems," *International Journal of Foundations of Computer Science*, vol. 12, no. 05, pp. 645–669, 2001.

[104] G. Wassi-Leupi, "Online scheduling for real-time multitasking on reconfigurable hardware devices," 2012.

[105] C. Ekelin, "Clairvoyant non-preemptive edf scheduling," in *ECRTS*. IEEE, 2006, pp. 7–pp.

[106] G. Wassi, M. E. A. Benkhelifa, G. Lawday, F. Verdier, and S. Garcia, "Multi-shape tasks scheduling for online multitasking on fpgas," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. IEEE, 2014, pp. 1–7.

[107] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "Online task scheduling for the fpga-based partially reconfigurable systems," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2009, pp. 216–230.

[108] A. El Farag, H. M. El-Boghdadi, S. Shaheen *et al.*, "On the acceptance tests of aperiodic real-time tasks for fpgas," in *IPDPS 2009*.

[109] T. F. Abdelzaher, V. Sharma, and C. Lu, "A utilization bound for aperiodic tasks and priority driven scheduling," *Computers, IEEE Transactions on*, vol. 53, no. 3, pp. 334–350, 2004.

[110] A. Al-Wattar, S. Areibi, and G. Grewal, "An efficient evolutionary task scheduling/binding framework for reconfigurable systems," *International Journal of Reconfigurable Computing*, vol. 2016, 2016.

[111] W. M. Wonham, *Supervisory control of discrete-event systems*. Springer, 2015.

[112] A. Vahidi *et al.*, "Efficient supervisory synthesis of large systems," *Control Engineering Practice*, vol. 14, no. 10, pp. 1157–1167, 2006.

[113] R. Brandt, V. Garg, R. Kumar, F. Lin, S. Marcus, and W. Wonham, "Formulas for calculating supremal controllable and normal sublanguages," *Systems Control Letters*, vol. 15, no. 2, p. 95, 1990.

[114] S. Saha, A. Sarkar, and A. Chakrabarti, "Spatio-temporal scheduling of preemptive real-time tasks on partially reconfigurable systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, pp. 1 – 26, 2017.

[115] B. A. Brandin and W. M. Wonham, "Supervisory control of timed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 39, no. 2, pp. 329–342, 1994.