Online Scheduling for Real-Time Multitasking on Reconfigurable Hardware Devices

A Thesis submitted for the degree of Doctor of Philosophy

By

Guy Wassi-Leupi

Department of New Media & Technologies,
Faculty of Design, Media & Management,
Buckinghamshire New University
Brunel University, London

July, 2011

Abstract

Nowadays the ever increasing algorithmic complexity of embedded applications requires the designers to turn towards heterogeneous and highly integrated systems denoted as SoC (System-on-a-Chip). These architectures may embed CPU-based processors, dedicated datapaths as well as reconfigurable units. However, embedded SoCs are submitted to stringent requirements in terms of speed, size, cost, power consumption, throughput, etc. Therefore, new computing paradigms are required to fulfil the constraints of the applications and the requirements of the architecture.

Reconfigurable Computing is a promising paradigm that provides probably the best trade-off between these requirements and constraints. Dynamically reconfigurable architectures are their key enabling technology. They enable the hardware to adapt to the application at runtime. However, these architectures raise new challenges in SoC design. For example, on one hand, designing a system that takes advantage of dynamic reconfiguration is still very time consuming because of the lack of design methodologies and tools. On the other hand, scheduling hardware tasks differs from classical software tasks scheduling on microprocessor or multiprocessors systems, as it bears a further complicated placement problem.

This thesis deals with the problem of scheduling online real-time hardware tasks on Dynamically Reconfigurable Hardware Devices (DRHWs). The problem is addressed from two angles :

(i) Investigating novel algorithms for online real-time scheduling/placement on DRHWs.

(ii) Scheduling/Placement algorithms library for RTOS-driven Design Space Exploration (DSE).

Regarding the first point, the thesis proposes two main runtime-aware scheduling and placement techniques and assesses their suitability for online real-time scenarios. The first technique discusses the impact of synthesizing, at design time, several shapes and/or sizes per hardware task (denoted as *multi-shape task*), in order to ease the online scheduling process. The second technique combines a *looking-ahead scheduling* approach with a *slots-based* reconfigurable areas management that relies on a 1D placement. The results show that in both techniques, the scheduling and placement quality is improved without significantly increasing the algorithm time complexity.

Regarding the second point, in the process of designing SoCs embedding reconfigurable parts, new design paradigms tend to explore and validate as early as possible, at system level, the architectural design space. Therefore, the RTOS (Real-Time Operating System) services that manage the reconfigurable parts of the SoC can be refined. In such a context, gathering numerous hardware tasks scheduling and placement algorithms of various complexity *vs* performance trade-offs in a kind of library is required. In this thesis, proposed algorithms in addition to some existing ones are purposely implemented in C++ language, in order to insure the compatibility with any C++/SystemC based SoC design methodology.

Key-words: FPGA, Reconfigurable SoC, DES, Modelling, Scheduling, Placement, RTOS.

Acknowledgments

This work was carried out in two institutions : at Faculty of Design, Media & Management, Buckinghamshire New University, High Wycombe (UK), and, in the Computer Architecture team at ETIS/CNRS lab, ENSEA, University of Cergy (France).

I would like to take this opportunity to thank Professor Geoff Lawday for giving me the opportunity to undertake this thesis dissertation. I would also like to express my gratitude to him for being always understanding and encouraging even when I felt stressed.

I am sincerely thankful to my supervisors Dr Kevin Maher from Buckinghamshire New University and Dr Amine Benkhelifa from ETIS lab, University of Cergy, for their valuable supervision and support over the years.

I owe a great debt to Professor François Verdier from University of Nice Sophia-Antipolis who advised, encouraged, and inspired my research. I have been blessed with two experts in reconfigurable computing, Amine and François. They have shown generosity and availability towards my research project.

I am grateful to Professor John Boylan and Buckinghamshire New University for supporting me with the funding necessary to carry out my research. I also gratefully thank the ETIS lab for providing me with extra funding along with a suitable research environment.

I would like to thank Professor Chris Hudson, Professor Inbar Fijalkow, Professor Bertrand Granado, Dr Peter Wilkinson, Laura Bray, Howard Bush, William Lishman, Dr Anne Evans, staff at Buckinghamshire New University and staff at ETIS lab, University of Cergy, for their support and advice. I will not forget my fellow research students and my former colleagues from Buckinghamshire New University and the ETIS lab. We have been sharing very fruitful discussions.

Thanks to Paul & Lucy Tanyi, Antoine & Ariane Kamina, Bernard Mankem, Julius Ebokolle, Premkumar Elangovan, Knowledge Mpofu, Indrachapa Bandara and many others that are not personally named here. They all made my life in UK enjoyable and sociable. Thanks to Randolph Boyd for its availability.

My family has been a constant source of understanding, encouragement and love. Thanks to my brothers and sisters, Michel, Alex, Theresine and Anne-Marie for their constant support and love. They always believed in me.

I dedicate this work to my beloved parents Jeanne-d'Arc Beumani and Emmanuel Leupi
to my wife Lydie-Flore, to my little princesses Jane-Veronica, Lise-Nahomie
and to Françoise-Soul-angel.
I wish Jeanne d'Arc, Françoise and Jeanne witnessed this achievement
I am quite sure, they are all smiling on me up there

Contents

List of Figures

List of Tables

Nomenclature

A_{fpga}	:	Area or total amount of resources on the reconfigurable array (FPGA)
a_i	:	Release time of a task or job T_i
Ap_{load}	:	Relative application load or the relative amount of resources required to complete an Ap_k on a given reconfigurable array of size $A_{fpga} = W \cdot H$
Ap_n	:	Application consisting of n jobs (or set of n tasks or jobs)
ar	:	Aspect ratio of a hardware task
c_i	:	Execution or processing time of task T_i
d_i	:	Absolute deadline of task T_i
D_i	:	Relative deadline of task T_i
e_i	:	Execution or processing time of task T_i
ft_{av} , rt_{av}	:	Average flow time or response time of a job sequence or tasks set
ft_i , rt_i	:	Response time or flow time of a scheduled task T_i
ft_{tot} , rt_{tot}	:	Total flow time or response time of a job sequence or tasks set
f_i	:	Finishing time of a task or job
h_i	:	Height of hardware task T_i
Hp	:	Hyper-period of a periodic tasks set
J_i	:	Job number i, with $i = 1...n$
l_i	:	Laxity of a task T_i
m	:	Number of processors or resources available for jobs processing
M_i	:	Processor number i in a multiprocessor system which consists of m processors $M_1, M_2, ..., M_m$
mk	:	Scheduling makespan or length of the scheduling
n	:	Number of jobs to be processed
n_a	:	Number of jobs accepted by the scheduling algorithm, among the k jobs in the application Γ_k, where $n_a \leq k$, and $k = n_a + n_j$
N_{acc}	:	Number of accepted tasks or jobs
n_j	:	Number of jobs rejected by the scheduling algorithm, among the k jobs in the application Γ_k, where $n_j \leq k$
N_{rej}	:	Number of rejected tasks or jobs
P_i	:	Period of a periodic task T_i

Rd_i	:	Rejection delay of a task T_i
Res_{Ap}	:	Total amount of resources required to complete all the k jobs $J_{i=1...k}$ of an application Ap_k
Res_{Ti}	:	Total amount of resources required to complete an instance of a hardware task T_i
$Rj_{\Gamma_k(\%)}$, $Rj\_(\%)$	:	Tasks rejection ratio
rt_{av} , ft_{av}	:	Average flow time or response time of a job sequence or tasks set
rt_i , ft_i	:	Response time or flow time of a scheduled task T_i
rt_{tot} , ft_{tot}	:	Total flow time or response time of a job sequence or tasks set
res_Γ	:	Computation load of an application Γ_k relative to a given reconfigurable array of size $A_{fpga} = W \cdot H$.
Res_Γ	:	Computation load or total amount of resources required to complete application Γ_k.
s_i	:	Starting time of a task or job
t_d	:	Absolute deadline of a set of tasks or an application
t_D	:	Relative deadline of a set of tasks or an application
T_i	:	Task number i, with $i = 1...n$
t_{rej}	:	Rejection time of a task or job
$U_{fpga(\%)}$	:	Utilization ratio of an FPGA of size $A_{fpga} = W \cdot H$ on which an application Ap_k has been scheduled
UR_{qm}	:	Differential quality metric of a scheduling on a reconfigurable hardware device
$U^{(t)}{}_\Gamma$	:	Time utilization factor of a set of tasks Γ
$U^{(t)}{}_{Ti}$	:	Time utilization factor of task T_i
w_i	:	Width of hardware task T_i
wr	:	Width ratio between a task and its fitting slot or cluster
wt_i	:	Waiting time of a scheduled task T_i
W , H	:	Width and Height of the reconfigurable hardware device (e.g. FPGA)
x_i	:	X coordinate of hardware task T_i
y_i	:	Y coordinate of hardware task T_i

α	:	Machine(s) or processor(s) environment
β	:	Application to process on the machine along with its constraints (e.g. time, precedence, etc...)
γ	:	The objective function of a scheduling
Γ , Γ_n	:	Set of tasks or jobs , set of n tasks or jobs
$\wp_i$	:	Priority of a task T_i
$\forall_i$	:	For all i
$\exists$	:	There exists
$\bigcap$, $\cap$	:	Intersection
$\bigcup$, $\cup$	:	Union
$\in$	:	Belongs to

Acronyms

ADAS Advanced Driver Assistance Systems

ASIC Application Specific Integrated Circuit

ASIP Application-Specific Instruction set Processors

ASSP Application-Specific Standard Parts

BF Best Fit

BSF Biggest Size First

CAD Computer Aided Design

CDMA Code Division Multiple Access

CLB Configurable Logic Block

CMOS Complementary Metal Oxyd Semiconductor

CPLD Complex Programmable Logic Device

DDR Double data rate

DRHW Dynamically Reconfigurable Hardware Device(s)

DPRHW / DPRHWs Dynamically and Partially Reconfigurable Hardware Device(s)

DPRHW-OS Operating System for Partially and Dynamically Reconfigurable Hardware

DRA Dynamically Reconfigurable Architecture

DSE Design Space Exploration

DSP Digital Signal Processor/Processing

EAAF Earliest Available Area First algorithm

EDA Electronic Design Automation

EDF Earliest Deadline First

EDGE Enhanced Data Rates for GSM Evolution

EEPROM Electrically-erasable programmable read-only memory

ETIS Equipes Traitement de l'Information et Systèmes

FF First Fit

FPGA Field Programmable Gate Array

GPP General Purpose Processor

GPRS General Packet Radio Service

GPS Global Positioning System

GSM Global System for Mobile Communications

HDL Hardware Description Language

HLL High Level Languages

HPC High Performance Computing

HSCSD High Speed Circuit Switched Data

HSTL High Speed Transceiver Logic

HW / SW Hardware/Software

IC Integrated Circuit

IDE Integrated Design Environment

IP Intellectual Property

ISP Instruction-set-processor

IT Information Technology

ITRS The International Technology Roadmap for Semiconductors

JTAG Joint Test Action Group

KAMER Keeping All Maximum Empty Rectangles

LAN / WLAN Local Area Network / Wireless LAN

LUT Look-Up Table

LVCMOS Low Voltage CMOS

LVTTL Low Voltage Transistor-Transistor-Logic

MAC Multiply-Accumulate

Mbps Millions of bits per second or megabits per second

MBps Megabytes per second.

MER Maximum Empty Rectangle

MIMO Multiple Input Multiple Output

MIPS Million Instructions Per Second

MMAC Millions of Multiply Accumulate

MPSoC Multi-Processors System-On-a-Chip

MSDL Merge Server Distribute Load

NF Next Fit

NRE Non-recurring engineering

OFDM Orthogonal Frequency Division Multiplexing

OS Operating System

OS4RHD Operating System for Reconfigurable Hardware Devices

PC Personal Computer

PDA Personal Digital Assistant

PE Processing Element

PLD Programmable Logic Device(s)

QAM Quadrature Amplitude Modulation

QoS Quality of Service

RC Reconfigurable Computing

RFT Right First Time

RISC Reduced Instruction Set Computer

RSOC Reconfigurable System-On-a-Chip

RT Real-Time

RTOS Real-Time Operating System

SDR Software Defined Radio

SERDES Serializer/deserializer

SFAF Smallest Fitting Area First algorithm

SIMD Single Instruction Multiple Data

SoC System-On-a-Chip

SoPC System-On-a-Programmable-Chip

SRAM Static Random Access Memory

SSF Smallest Size First

SSTL Series Stub Terminated Logic

TTM Time to market

UART Universal Asynchronous Receiver/Transceiver

UML Unified modeling language

UMTS Universal Mobile Telecommunications System

VLSI Very Large Scale Integration

W-CDMA Wideband Code Division Multiple Access

WCET Worst Case Execution Time

1D-VSSH 1D Variable Size Slots Horizon algorithm

1D-VSSS 1D Variable Size Slots Stuffing algorithm

3G 3rd Generation radio communications system

Chapter 1

Introduction

Dynamically reconfigurable hardware devices such as FPGAs[1] are becoming the enabling technology for true hardware multitasking in embedded systems and high performance computing. Such devices are increasingly used in heterogeneous System-On-a-Chip designs. This thesis copes with the problem of scheduling online real-time hardware tasks on reconfigurable hardware devices. As online real-time applications are targeted, scheduling algorithms are combined with appropriate placement strategies in order to provide low scheduling runtime overheads. Online real-time scheduling requires on-the-fly reconfiguration (partially or not) of the reconfigurable device in a reasonable amount of time, depending on the targeted application domain. This work targets dataflow oriented embedded applications as they are the most suitable for hardware implementation.

1.1 Precis of Embedded Systems and Research Rationale

The following summary is a plain English introduction to embedded systems, which is given to address a wide audience, and in particular a résumé of embedded reconfigurable systems that relate to the area of research set out in this PhD thesis. Embedded electronic systems cover an extensive range of topics whereby dedicated programmable electronic systems are the key mechanisms in aircraft fly-by-wire systems, satellite navigation, vehicle engine management and entertainment systems along with space vehicle control and communication, machine tool control, cameras, mobile telecommunications, robots and toys, etc. Today the number and variety of embedded system

[1] Field Programmable Gate Array

applications appears to be endless. However, although there are a multitude of embedded systems in production, the challenges facing designers are significant as recent technological advances are pushing the boundaries of engineering, such as high-speed digital design coupled with constraints in power consumption and the increasing complexity of information communication, together with the increased demands of new compliance standards for improved interoperability and reliability.

Nowadays, as technology advancement for the semiconductor industry is driven by Moore's Law[2], a complete system with numerous functionalities can be implemented on the same integrated circuit chip. Such a system is denoted as SoC[3] or MPSoC[4] when integrating more than one processor core (e.g. general purpose processors, domain specific processors, etc.), dedicated processing blocks like voice encoding/decoding, cryptography, etc.), memory blocks, inputs/outputs blocks, inter-blocks communication media along with other analog blocks (RF blocks, ADC/DAC blocks, antenna, MEMS sensors, etc.) as pictured in figure 1.5, on page 14.

One consequence of the complexity of designing modern embedded systems has been the introduction of new university courses in signal integrity engineering (as communication bandwidth is ever increasing) and embedded system architecture, especially the so-called SoC. Concurrent with the new undergraduate courses, there is a surge in novel post graduate research programmes associated with embedded systems engineering.

An important area of embedded systems development is telecommunications, where the next generation of cellular telephony and broadband radio are seen as significant areas of research. Future telecommunication and radio systems will require a step change in architectural design, software complexity and associated computing power to achieve the improvements envisaged for the next generation of telecommunication functionality. In the past, these challenges have been solved by adding dedicated digital signal processors (DSPs) known as ASICs[5] to conventional General Purpose Processors (GPP) or programmable DSP[6]. A custom hardware such as an ASIC is used to provide the processing performance required by a given application and consume less power while a GPP or a programmable DSP brings more flexibility to the system.

However, designed as a SoC or not, this architecture is not seen as sufficient to meet the ever increasing demands of embedded applications presented above.

[2] Gordon Moore, the co-founder of Intel, predicted in the 1960s that processor through-put would double every eighteen months

[3] System-on-a-Chip

[4] Multi-Processors SoC (System-on-a-Chip)

[5] Application Specific Integrated Circuits

[6] Digital Signal Processor

1.2 Raison D'être for using Reconfigurable Hardware Devices in Embedded SoCs

One solution to the step change requirements in embedded system architectural design is the evolution of SRAM[7]-based PLDs[8] such as FPGAs. Although FPGAs were first introduced in the mid 1980s by Xilinx, the recently introduced devices are highly developed. By providing some form of post-fabrication hardware programmability, these devices are the enabling technology for reconfigurable computing. Today, as shown in figure 1.1, Xilinx shares the market lead with Altera in programmable solutions. These companies are proposing architectural solutions that bridge the gap between general programmable processors and custom integrated circuits and that provide the computational power and flexibility required for future systems. Nevertheless the advances in reconfigurable hardware technology (e.g. SRAM-based FPGA and similar devices) in particular its dynamic runtime reconfigurability, have brought new possibilities in embedded systems design. Indeed, dynamically reconfigurable hardware devices provide more flexibility and silicon area reuse in addition to their intrinsic parallelism (spatial implementation) as illustrated in figure 2.20 on page 62. However such architectures raised new challenging design issues. The principal area of interest of this research can be summarized through two main points listed below:

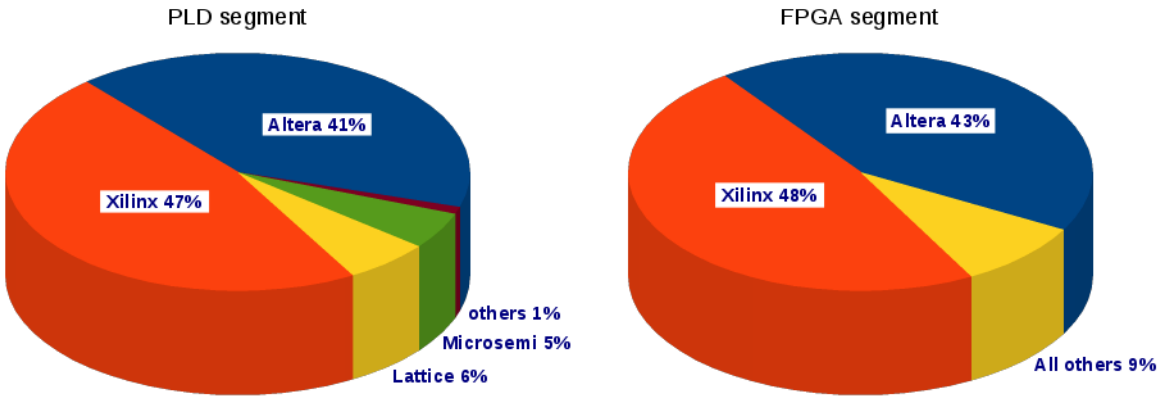


Figure 1.1: Programmable device market segment share in 2011 (Xilinx, Company reports).

[7] Static Random-Access Memory (SRAM) is a type of semiconductor memory which does not require to be preiodically refreshed, as SRAM memory uses bistable latching circuitry to store each bit. However the data are lost when the memory is not powered.

[8] Programmable Logic Devices

- The lack of embedded systems design methodologies leveraging dynamic reconfigurability of FPGA-like reconfigurable hardware devices, when they are incorporated in such systems in general, and especially when they are on the same silicon die (SoC).
- The need of an *Operating System - like* manager to cope with the scheduling and placement of hardware tasks in heterogeneous embedded systems (on a single intergrated circuit or not) featuring such a reconfigurable fabric.

There are three main but non-exhaustive reasons for developing reconfigurable architectures are :

1. The dynamicity of new embedded applications.
2. The market and manufacturing costs constraints.
3. Advances in reconfigurable hardware devices technology.

1.2.1 Dynamic and Online Embedded Applications

Embedded computer systems provide more possibility to interact with their environment (e.g. the user or another computer system in its neighborhood). Hence, embedded applications are becoming more dynamic and require more computation power to process all incoming or outgoing data and more often in a safety-critical context.
Hereinafter are few and non-exhaustive embedded dynamic applications:

Example of Software Defined Radio

Next generation mobile telecommunication terminals will require flexibility and high performance under low-power and size constraints. Indeed, they will be expected to be flexible in order to dynamically adapt to any wireless infrastructure, download and run applications offered by services providers, while providing reliable functionalities, such as Personal Digital Assistant (PDA), mobile phone, MP3 player, Global Positioning System (GPS), Audio/Video streaming, messaging, and other multimedia services available on future networks. To achieve this aim, mobile terminals will need architectures that are fast enough to run complex algorithms at high data rates, typically 10 Mbps to 100 Mbps (as pictured in figure 1.2, from Schüler and Tan, 2004), and flexible enough to accommodate various new standards and protocols. Software Defined Radio (SDR) concept is at the nearly ultimate stage of this evolution. A *software radio* implies an embedded computer system where the functionality of the mobile terminal should be defined in

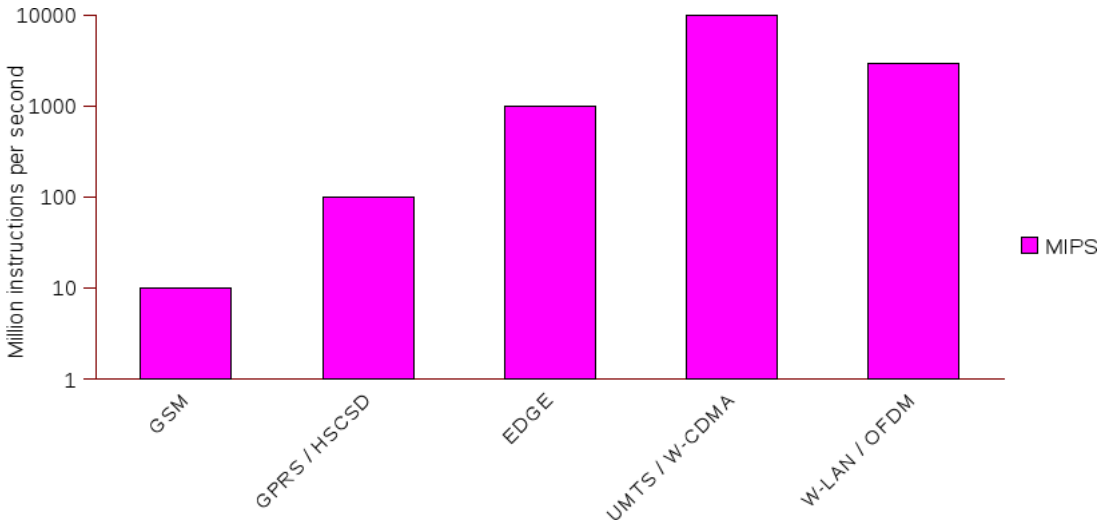


Figure 1.2: Processing requirements for wireless access protocols (Schüler and Tan, 2004)

software, so that it enables full programmability and adaptability on-the-fly. Consequently, such an 'all-in-one' device could be used everywhere in the world giving global mobility without adding new hardware, regardless of the number of global wireless communication standards. Nevertheless SDR requirements are stringent, where flexibility entails a powerful reconfigurable computational capability with multifunctionality. Moreover, successful SDR systems will require to operate under severe constraints (e.g. reduced power consumption and physical size, low cost, etc.). These systems are difficult to achieve with GPP processors, programmable DSP and ASICs; even though modern microprocessors and programmable digital signal processors are quite flexible and have benefited from fabrication advances and tool development over recent years. Processor technology has reached an unexpected plateau that contradicts Moore's law. Today manufacturers have found that increasing the processor clock speed and reducing the processor device size has led to undesirable heat problems and unacceptable power consumption. Modern processor manufacturers have chosen to provide multiple processors within a single integrated circuit to advance Moore's Law, but the software complexity of parallel processing has thwarted their progress. SDR designers have chosen to use dedicated hardware and coprocessors to provide high-performance, low power systems. But the latter solution lengthens the design process while not providing the flexibility needed in software defined radio applications. To overcome these deficiencies in flexibility, a dedicated component, such as an ASIC, could be used for each telecommunication standard or protocol. With this approach the flexibility is achieved by switching from one component to

another to meet the required standard or protocol. The main advantage is that high-performance is easily achieved, since each custom component is optimized for a particular standard. Except that with the proliferation of standards this solution is not cost effective because of the resulting size of the silicon die, where the cost of an integrated circuit is related exponentially to die size. Also such an embedded system would suffer from weight, power consumption and time to market disadvantages.

Example of electronic embedded in transportation systems

In transportation systems (e.g. aerospace, avionics, car, railway, etc.), functionalities assigned to the embedded computer are no longer limited to basic control and multimedia tasks. Indeed, the on-board computer increasingly consists of sophisticated systems performing complex real-time analysis on data collected by numerous sensors in order to assist pilots in critical flight situations. In any case, such systems are expected to be more and more intelligent, and require ever increasing computing capabilities allowing them to process in real-time the data collected, in order to take rapid decisions while insuring security and safety to the users. The on-board computer must drive these systems which react in real-time to dangerous and unexpected situations. Example of such systems are radar systems, auto-pilot systems, collision-detection systems, communication systems, etc.

Example of an Automatic Target Detection and Tracking System

Automatic target detection and tracking is an important area of research in video processing thanks to its great potential in military and civil applications. Example of such applications field are navigation, security, robotics, vehicular communications, etc. One of the challenging applications field is aerospace and defense where there is a need to detect, recognize and track flying targets (e.g. missiles, drones, fighters, etc.). Target detection and tracking is a good example of dynamic and computationally intensive application. It aims to detect, identify and track targets mostly in infrared image sequence with a given spatial and temporal resolution. The spatial resolution gives the size of each frame in the video (e.g. 640 x 512) and the temporal resolution gives the number of frames per second (e.g. 25 Hz for 25 fps). The dynamicity of the application relies on non predictable spatial and temporal factors such as:

- the date and the position of appearance of a target(s) on the visual field.
- the identity of the target(s) (type, shape, size and number of objects in a frame, etc.).

- its trajectory (direction and speed of each object, etc.).

The system obviously consists of two major parts :

1. a *static* part which detects and recognizes objects in a video frame (detection sub-system).
2. a more *dynamic* part which tracks objects in the video sequence (tracking sub-system).

Detection aims to identify all the objects and connected components in the image. It uses functions such as Sobel and/or Canny edge detection, blobs detection, surface filling-in, objects' surrounding rectangles calculation, thresholding, erosion and dilation, etc.
The *tracking* sub-system performs detected objects tracking using following steps :

- initializing the tracking algorithm using objects' surroundding rectangles along with blobs previously built.
- applying the tracking algorithms (e.g. CAMSHIFT) on detected blobs.
- updating the size of objects' surrounding rectangles and the list of connected components.

One key feature of such a system is that on a video frame, many objects tracking should be performed concurrently and the number of tracked objects may evolve over time. This makes the scenario even more dynamic. Therefore, performant and evolvable computation resources are needed. As the number of objects to pursue is unknown beforehand, one approach is to implement in software the detection (or static) part of the system, and to dynamically instantiate on demand tracking functions on dynamically reconfigurable hardware. Doing so, one tracking function block per target is implemented (as a thread) on the dynamically reconfigurable hardware, thus taking advantage of flexibility and performance of such hardware.

Example in car design (e.g. System detecting driver fatigue)

Cars are becoming more equipped with systems which improve global safety by providing more assistance to the driver. The driver safety is improved by systems capable of detecting its sleepiness (e.g. Bandara and Hudson, 2006). In addition pedestrians safety is improved by the Advanced Driver Assistance Systems (ADAS). The ADAS requires a high computation capability to achieve vision functionalities (stereovision, pattern recognition, complex scenes analysis) in a very short timeframe in order to detect pedestrians and thus reduce the number of accidents.

In all the above cited applications, cost and power consumption constraints have to be added to the aforementioned safety (reliability, hard real-time sensitivity, high computing capability, etc.)

and security constraints. In addition, image processing is involved in most of these applications. Thanks to its inherently parallelism, image processing has proven to be suitable for hardware implementation. Most of the time, output pixels could be concurrently computed . Reconfigurable architectures are good candidates for implementing such applications. Indeed, they suit to applications where huge amount of data (image, sound, etc.) are processed in parallel and periodically. In addition, they provide an upgradability or hardware programmability which increases the system lifetime and therefore decreases the non-recurring engineering costs.

1.2.2 Technology Advances, Market and Costs Constraints

1. ***Reducing non-recurring engineering (NRE) costs***

   Current and future research trends into embedded System-on-a-Chip (SoC) design are dictated by key factors such as market demands and technology advances. New research trends on reconfigurable hardware devices (especially FPGAs) result from these key factors, as these devices are currently seen as a solution to successfully design complex embedded systems; in particular the design of systems submitted to stringent constraints of the above-mentioned applications. The current market for wireless handsets is driven by factors such as price, flexibility, functionality and mobility; put simply, a handset has to be light weight, all inclusive and with extended battery life. Moreover, consumer electronic products such as wireless handsets have relatively short life cycles. The merciless competition between mobile service providers and handset design conspire to ever decrease prices and increase functionality. Consequently, the main challenge is to reduce production costs while providing more functionality and flexibility to consumers. While designing new products, shortening Time-To-Market (TTM) and increasing Right-First-Time (RFT) are important aspects that reduce non-recurring engineering (NRE) costs, as NRE costs of integrated circuit (IC) design are rocketting today. Many research projects in Electronic Design Automation (EDA) mainly aim to achieve those two aforementioned purposes (short TTM and high RFT). For example, FPGA-based platforms have been promoted for complex ASIC rapid prototyping. A rapid prototyping allows the designers to shorten the design process, to prevent design failures and to avoid costly redesign by validating their designs earlier and by prototyping their concepts within a reduced timeframe. Furthermore, designers are incorporating reconfigurable hardware in their designs, leading to cost effective products and enabling the release of new products without re-manufacturing the integrated circuit chip. FPGA-based

designs are even preferred to an ASIC based system for the implementation of low-volume applications thanks to their very low non-recurring engineering (NRE) cost.

2. ***Modern reconfigurable hardware technology impacting SoC design***

   Although a reconfigurable device like the FPGA typically has a higher power consumption than a comparable ASIC, the FPGA technology is evolving and current devices allow designers to consider FPGA as a computing resource for hardware acceleration. However, FPGAs are mainly used for rapid prototyping and glue logic purpose, nonetheless FPGAs are challenging programmable DSP processors by embedding hardwired DSP blocks, such as multipliers and distributed memories. Also, processor cores are currently embedded within FPGA structures (figure 2.16 on page 50). In so doing, an embedded hard-core or softcore processor core allows the designer to combine on a single FPGA all the benefits provided by a Von Neumann architecture with the parallelism provided by a spatial implementation.

   What is more important and a pivotal consideration in this research is reconfiguration capabilities of Dynamically Reconfigurable Hardware Devices like FPGAs, which allow them to bridge the gap between hardware and software platforms implementation (figure 2.20 on page 62). Even though programmable and dedicated DSPs remain the traditional implementation platforms for digital signal processing applications, many studies and demonstration platforms (e.g. Blyler, 2005; Petersen, 1995; Tessier and Burleson, 2001) have shown that using reconfigurable hardware devices, such as FPGAs, provides the best trade-off between flexibility, performance and power consumption, especially in digital signal processing applications. FPGA programmability brings out a flexibility lacking in ASICs, while FPGA spatial structure is more suitable for to intrinsic data parallelism found in DSP functions. Thus, all these factors point the way forward for the FPGA to provide a significant performance improvement over traditionally implemented embedded system platforms.

   Current advances in semiconductor technology allow FPGAs to integrate more than 10 million gates[9] on a single chip while running at a relatively low frequency (e.g. 500 MHz).

[9] Achronix Semiconductor Corp. announced (in fall 2010) strategic access to Intel Corporation's 22nm process technology, and plans to develop the most advanced FPGAs, the Achronix Speedster22i FPGA family. The device will provide more than 2.5 M LUTs in size, equivalent to an ASIC of over 20 million gates.

Altera Stratix 5 FPGAs as well as Xilinx Virtex-7 FPGAs both manufactured at 28 nm process technology, provide more than one million LUTs, which is more than double the size of logic and memory available in Stratix or Virtex-II FPGAs. One million LUTs are enough to instantiate on the FPGA tens of softcore

Figure 1.3 depicts such an increase in LUT density over the last decade. In consequence, unlike the microprocessor, Moore's Law will still drive the FPGA market for the near future. One example is the FPGA Stratix 5 from Altera [10], manufactured at 28 nm process technology. Moreover, today complete systems are integrated on a single chip. Whereby, this so called System-on-a-Chip (SoC) allows designers to integrate on single silicon dies one or more embedded processor cores executing software in addition to classical hardware, such as an ASIC, In/Out device, memory, Intellectual Property (IP) blocks and FPGA programmable hardware. Nonetheless, SoC design requires new design paradigms and represents an active area of embedded system research. SoC design approach reduces the total silicon size required for the SoC, and correspondingly its cost. An exciting prospect is that the inclusion of a reconfigurable FPGA within a SoC could provide solutions to some of the demanding requirements of future embedded system applications.

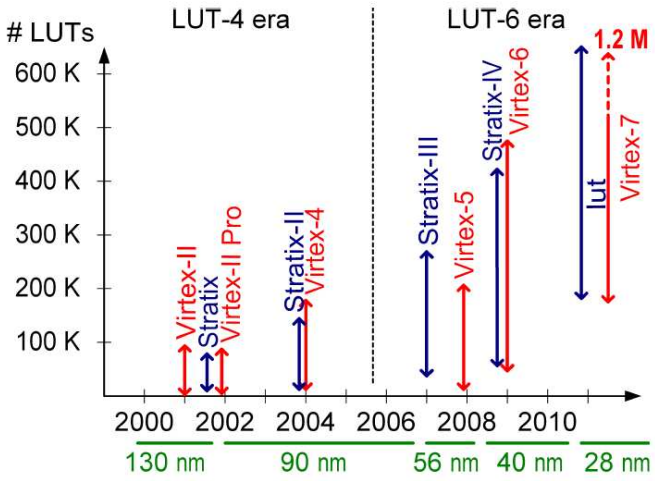


Figure 1.3: The increase in logic density in FPGA over one decade and over the corresponding process technology (Koch and Torresen, 2010).

Fortunately, latest trends in reconfigurable hardware devices technology show improvements in capacity (decreasing transistor size and consumption), performance (increasing clock frequency) and reconfigurability (partial on-the-fly, see chapter 2, section 2.5 from page 33).

CPUs along with the required peripherals.

[10] The Altera Stratix 5 FPGA was announced in January 2010 and planned to be available in 2011 (Altera, 2010a,b)

To summarize, using reconfigurable hardware in embedded systems design could reduce design costs by lowering non-recurring-engineering costs and by providing a computation platform combining flexibility and performance.

1.3 Related Research Issues

1.3.1 System-On-Chip Design Overview

As stated previously, systems to design are becoming increasingly complex in order to satisfy application demands. In addition, technology allows huge systems to be integrated on a single chip, making their design more difficult. Such a so-called System-on-a-Chip (SoC) is in growing need for new design paradigms. Because of the increasing complexity, new design methodologies emphasize system level design. Design stages could be divided in three main levels (or less) as shown in figure 1.4 and summarized below :

1. ***System Level*** *(figure 1.4)*

   System performance is evaluated early and various partitioning decisions are made. While engineering a new system, first specifications of the application are done at high level of abstraction according to system requirements. At that level, abstract models and templates of different implementation technologies from different vendors are first included in order to pre-define the architecture, to take partitioning decisions and to evaluate mapping needs for each target. System level simulation then allows the designer to evaluate in a reasonable amount of simulation time, the performance impacts using different architecture alternatives. The system architecture is refined by providing System Level simulation with cycle accurate models of the architectural blocks and requirements of the application.

   In the case of a SoC with a reconfigurable part, the reconfigurable hardware is seen as a computing resource among others like instruction set processors or dedicated processors. The reason for incorporating reconfigurable parts are expressed through systems requirements (e.g. a given mix of reconfigurability, upgradability and performance). So, the impacts of scheduling, placement and reconfiguration overhead on the global performance of the system is also evaluated. Academic and commercial research groups are proposing various system level co-design methodologies and tools for few application domains (e.g. wireless communication and multimedia systems).

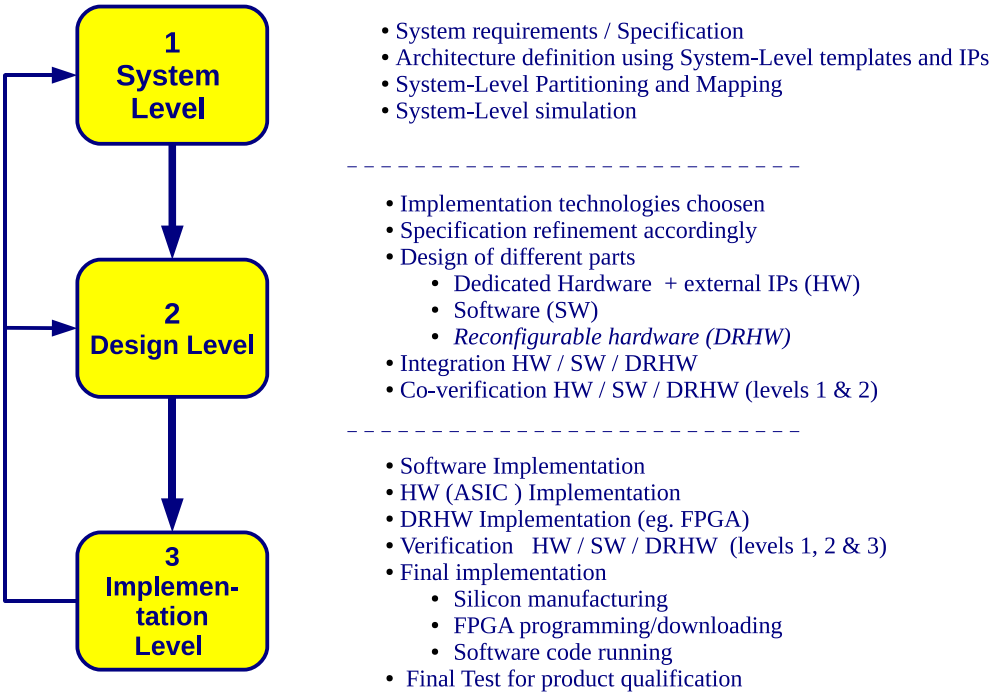


Figure 1.4: Main levels in the generic design flow of a SoC

2. ***Design Level*** *(figure 1.4)*

   Specification is refined and verification process customized to suit the chosen implementation technologies and related design tools. Different parts of the system (reconfigurable and fixed hardware, software) are individually designed and then integrated in a single model in order to be model-checked by the verification process previously customized. At this level, all details of the implementation platforms are needed (processors, memories, external IPs, FPGAs, etc.) and vendor design and simulation tools are used. For example if dynamically reconfigurable hardware (DRHW) is involved, the FPGA technology along with its simulation and emulation tools are chosen during specification refinement with respect to the reconfigurability needed (partial, dynamic, etc.). Vendor tools are used during integration and co-verification steps.

3. ***Implementation level*** *(figure 1.4)*

   where the whole design is implemented using commercial and technology-dependant tools. The fixed hardware part (ASIC) is also manufactured and the final text is done on the final product for its qualification.

1.3.2 Reconfigurable System-On-Chip Design

As stated above and illustrated in figure 1.4, designing an electronic system (on chip or not) always follows a number of steps ranging from system level specification to final implementation (including mask generation for ASIC design). Over the years, EDA[11] tools have evolved, providing sometime push-button processes to design electronic systems. However, as SoCs are turning more heterogeneous (figure 1.5) by integrating reconfigurable hardware devices in addition to sequential processors and dedicated hardwired components, designers are facing new paradigms of computing and programming. Consequently, new automatic mapping methods of algorithms on the platform are needed, in order to take into account the additional flexibility brought by reconfigurable hardware devices. Indeed, such a platform allows the system to modify its hardware functionalities at run-time (on-the-fly) following the changing needs of applications. As for any ASIC design, a post-manufacture failure would be of very costly. One has to guarantee the expected behaviour (RFT[12]) of the system before manufacturing. Unfortunately, dynamic hardware reconfigurability in addition to the traditional software flexibility leads to systems with less predictable behaviour, making the integration and validation of software parts (running on programmable processors) and hardware parts (running on dedicated and reconfigurable hardware) more challenging. Current tools do not enable an accurate assessment of dynamicity criteria brought both by the software and reconfigurable hardware. Hence, design methodologies along with tools still have a long way to go before fully exploiting the potential capabilities of reconfigurable hardware devices.

As illustrated in figure 1.5, a SoC containing one or many reconfigurable hardware fabrics is denoted as RSoC[13]. SoCs are turning to Multi-Processors SoC (MPSoC) integrating on a single chip many programmable and dedicated processors, customized blocks (voice encoder/decoder, cryptography, etc.), memory blocks, I/O blocks, various communication media (hierarchical buses, NoC[14], etc.), ADC[15] /DAC[16] blocks, RF[17] front ends, heterogeneous blocks like sensors and MEMS[18], and sometime reconfigurable hardware blocks like FPGAs (figure 1.5).

In order to refine aims and objectives of this research, here below are discussed few additional

[11] electronic design automation
[12] Right First Time
[13] Reconfigurable System-On-Chip
[14] Network on chip
[15] Analog to Digital Converter
[16] Digital to Analog Converter
[17] Radio Frequency
[18] Micro-Electro-Mechanical Systems

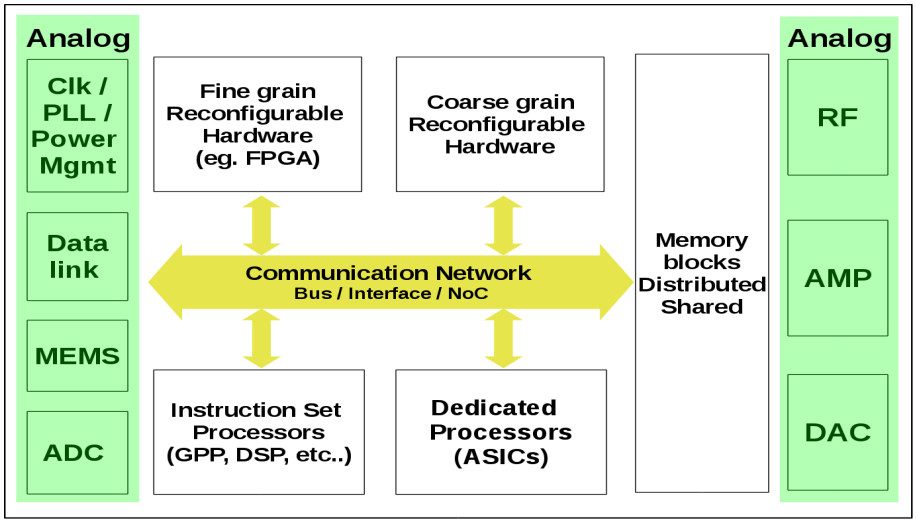


Figure 1.5: Generic architecture of a Reconfigurable System-On-Chip.

challenges brought by the increasing use of DRHW[19] in SoCs design. These issues are identified at different stages in the design process.

1. ***System specification and Hardware/Software Codesign***

   Obviously, using DRHW adds more complexity in traditional hardware software integration and verification. Emerging hardware/software Co-design methodologies are allowing designers to specify and refine their systems in unified environments (Ptolemy, Polis) and languages (SystemC (www.systemc.org), SystemVerilog (www.systemverilog.org), Celoxica (2000), ImpulseC (www.impulseaccelerated.com), etc.). As illustrated in figure 1.6, hardware/software co-design shortens the design time by enabling concurrent design of hardware and software parts of the system. Unlike the traditional design methodology, the trend today is to delay the partitioning stage of the design to allow the movement of a task scheduling from hardware (dedicated or reconfigurable) to software and, conversely, to be kept as flexible as possible during the design process. Previously, hardware/software partitioning was clearly defined as a spatial (or functional) partitioning. But the advent of DRHDs has provided the motivation for new research in hardware/software partitioning. Indeed, dynamically reconfigurable hardware has filled the gap between hardware and software, making the partitioning process both temporal and spatial (e.g. Chehida and Auguin, 2002;

[19] Dynamically Reconfigurable Hardware Devices (e.g. FPGAs)

Kaplan et al., 2003).

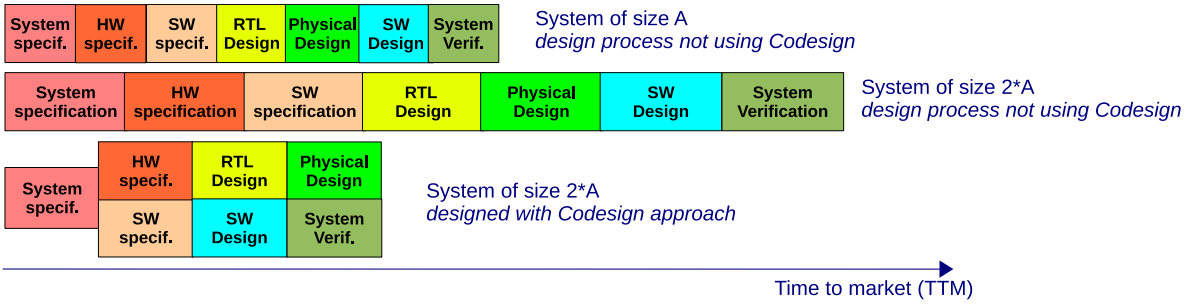


Figure 1.6: Hardware/Software Co-Design shortens the design process(Fujitsu, 2002)

As stated above, at a time when Moore's Law is getting less profitable for microprocessor improvement, it is getting meaningful to increase performance by adding more reconfigurable hardware along with distributed memories and by using pipelining. Hence, as most of research challenges in embedded SoC design which tend to leverage silicon technology advances, FPGA research also falls within the global problem known as 'the productivity gap'. It states that thanks to Moore's Law, integration technology is growing faster than the ability of the engineers and design tools to benefit from the doubling of through-put every eighteen months. As shown in figure 1.7, the gap between the number of logic gates that can integrate a single chip and the number of logic gates designers could integrate in their design using existing design methodology and tools increases over years at about the rate predicted by Gordon Moore.

2. ***C to hardware compilation***

   Unlike microprocessor implementation using high level C-like languages, mapping algorithms in fine grain reconfigurable hardware (e.g. FPGA) is still a complex and almost manual task. Indeed, this is done using low level Hardware Description Languages (HDLs). The designer must specify his design almost at bit level, reminding the early days of microprocessor programming. While taking the FPGA as an example, since it remains the finest grain reconfigurable hardware device, its design process is quite similar to the ASIC design. In addition, compiling an algorithm to target an FPGA is noticeably complicated as FPGA, unlike a microprocessor, does not have an instruction set. To overcome those problems and exploit the overriding FPGA advantages, numerous researchers (e.g. Athanas and Silverman, 1993; Gokhale and Stone, 1998; Gokhale et al., 2000; J. L. Tripp and Hutch-

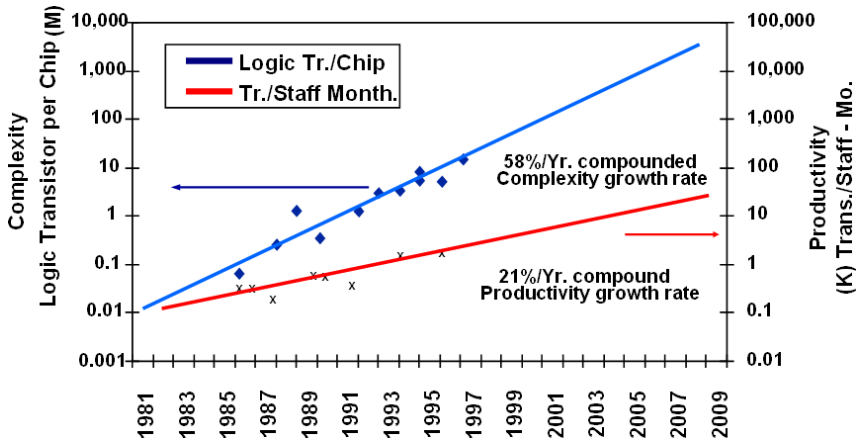


Figure 1.7: Productivity gap according to ITRS (The International Technology Roadmap for Semiconductors, www.itrs.net)

ings, 2002) and companies (e.g. MathWorks, www.mathworks.com)[20] have addressed High Level Languages (HLL) and their compilers to target reconfigurable hardware. Another example is the Furthermore, a solution consists of designing coarse-grained reconfigurable architectures which would be easier to target with High Level Languages as detailed in the next paragraph.

3. ***Need of coarse-grained reconfigurable architectures***

   To overcome some of the previously discussed limitations, new coarse-grain reconfigurable architectures with limited configurability compared with FPGAs were investigated. Coarse-grained architectures are customized to suit a specific class of applications. In such a so-called word level granularity architecture, the reconfiguration is done at functional level (instead of gate level like in FPGAs). As in the example in Schüler and Tan (2004), a reconfigurable processor consisting of an array of configurable and interconnected arithmetic logic units (ALUs) and enabling high level parallelism is a good example of this type of architecture. Although, such application oriented architectures are not as suited as fine-grained FPGAs to implement a multipart function at the bit level. However, a

[20] e.g. The *Simulink HDL Coder* from *MathWorks* generates bit-true and cycle-accurate, synthesizable Verilog and VHDL code from Simulink models and MATLAB code. The generated HDL code can be simulated and synthesized using industry-standard tools and then implemented on FPGAs and ASICs. The main advantage of this so-called Model-Based Design approach it to relies on the same model throughout the design, enabing FPGA-in-the-loop cosimulation and accelerating verification and time-to-market.

coarse-grained reconfigurable hardware saves a significant amount of circuit area, power and reconfiguration overhead compared to a traditional FPGA based system. As discussed in Gokhale and Graham (2006) and Ebeling et al. (1996), the trend is to develop high level languages and dedicated compilers for such architectures, with the hope that they will be easier to target than fine grain architectures such as FPGAs. Later in Chapter 2 these concepts and architectures will be briefly presented alongside their associated research areas.

1.3.3 Operating System for Reconfigurable System-On-a-Chip

As stated in previous sections, many of today's embedded applications, such as signal and image processing for mobile telecommunications, wireless multimedia, automotive electronics, avionics and robotics, demand an increasingly dynamic performance and variable functionality. To meet these needs, FPGA developers provide large-scale devices to allow embedded system designers to consider the use of an FPGA as a computing resource at the same level as a microprocessor or a programmable DSP processor. A consequence of FPGA runtime partial reconfiguration and embedded dedicated blocks is the need for a resources manager acting like an *Operating-System*. This resources manager being capable of efficiently managing both the microprocessor and the reconfigurable hardware part of the architecture (e.g. implemented on an FPGA) that run respectively software tasks and hardware tasks. In a traditional Operating System (OS), software tasks are sequentially created, run, pre-empted and/or deleted, allowing them to sequentially use a single microprocessor in a time-shared basis. In the case of the FPGA/ISP[21] architecture, the OS may be viewed as an additional abstraction layer which hides the details of the underlying microprocessor and reconfigurable hardware part from the software designer (e.g. Steiger et al., 2004). By extending this principle to reconfigurable hardware, an OS can provide a hardware abstraction layer which eases the sequential and spatial implementation of hardware tasks on reconfigurable blocks (e.g. Mignolet et al., 2003). This allows multitasking on a reconfigurable hardware platform. Therefore, in a reconfigurable system with both hardware and software tasks the OS concept is extended to an Operating System for Reconfigurable Systems. As stated in Nollet et al. (2003), the principal aim of an Operating System for a Reconfigurable SoC is the simplicity of designing reconfigurable hardware platforms and efficient management of the associated computing resources. What is exciting are the new research areas initiated by

[21] Instruction Set Processor

Operating System for RSoC and, moreover, reconfigurable computing system design. Resources allocation, HW/SW partitioning, tasks scheduling, placement, routing, pre-emption, migration (hardware/softare, hardware/hardware) are few of the challenges involving numerous groups (e.g. Compton et al., 2002; Walder and Platzner, 2002; Noguera and Badia, 2004) investigating new reconfigurable FPGA/ISP architectures along with their programming models and implementation.

1.4 Contribution of the Thesis

There are two major contributions in this thesis :

1.4.1 Algorithms for Online Real-Time Scheduling & Placement

As just stated above, designing an operating system for reconfigurable systems is a key challenge in RSoC design as it raises a number of issues. Among these issues, this thesis will focuse especially on algorithms which are suited to the scheduling and placement of online real-time hardware tasks on dynamically reconfigurable architectures. Hardware tasks scheduling and placement aim to efficiently and dynamically schedule and place modules on the dynamically reconfigurable hardware devices. Therefore, the algorithms will always seek the best trade-off between two, sometimes conflicting objectives: the algorithms runtime overhead and the scheduling/placement quality in terms of chip utilization ratio, chip fragmentation, tasks rejection ratio, makespan, etc. To enable multitasking and hardware virtualization on partially reconfigurable hardware devices, reconfiguration time overheads and area fragmentation are among the issues to overcome. The algorithms proposed in this thesis are mainly applied to partially reconfigurable hardware device. In the proposed methodology, scheduling algorithms runtime overheads are sometime evaluated on real platforms featuring an embedded processor, instead of desktop or laptop computers. For example, two algorithms (Cui and Deng, 2007; Handa and Vemuri, 2004c) are implemented on a Microblaze softcore processor instantiated on a Xilinx Spartan 3E FPGA. By doing so, as the methodology relies on more accurate experiments, the online real-time constraints are more likely to be accurately assessed. Different models of elements that are involved (reconfigurable fabric, scheduler, placer, application) are also proposed and discussed in Chapter 4. However, these models do not consider inter-tasks communication and communication between the CPU part and the reconfigurable hardware device, as communication is beyond the scope of this thesis.

1.4.2 Scheduling & Placement Algorithms for OS-driven Design Space Exploration

System level co-design methodologies are widely accepted as an approach to overcome the increasing complexity of SoC design. As an Operating System aims to manage all the resources of a given platform, architecture definition could be done from an OS perspective. An OS-driven methodology for RSoC design has been proposed in Miramond et al. (2009a). Initiated by the French national research council, the OveRSoC project (Miramond et al., 2009a) aims to develop a complete model of an RSoC platform in order to investigate services a real time Operating System (RTOS) for RSoC should provide. Such Operating System for RSoC should manage processing resources that are on the chip, including the dynamically reconfigurable hardware parts of the RSoC. Consequently, as dynamically reconfigurable hardware is involved, hardware tasks scheduling and placement, hardware context switching, hardware/software task migration are pivotal to this research. Our contribution to the OveRSoC design methodology intends to provide DRHDs models and a set of scheduling and placement algorithms. Indeed, at system level simulation, while performing architecture definition and mapping (figure 1.4, page 12) these models and algorithms help in exploring various mapping solutions and in tuning the system partitioning accordingly. As the methodologies such the OveRSoC rely on a SystemC-based simulation engine, the models description and most of the scheduling and placement algorithms implementation are done in C++ in this work.

1.5 Outline of the Thesis

This first chapter has presented a snapshot of embedded systems that use reconfigurable hardware devices like FPGAs. Put simply, this introduction has striven to explain how the use of state-to-the-art reconfigurable hardware devices can be used as processing resources in embedded systems to enhance their performance; however their use raises significant new issues, especially in embedded RSoC design. These issues range from hardware/software partitioning to hardware tasks scheduling and placement.

The main focus of this thesis is on the latter problem, with an emphasis on the online real-time context. However, there a two main aspects of RSoC design to be highlighted as the contribution of the thesis :

1. The first aspect acts at runtime, where appropriate algorithms for online real-time schedul-

ing and placement of hardware tasks on dynamically and partially reconfigurable hardware devices are required. Novel methods for improving the quality of the online real-time scheduling and placement of hardware tasks on reconfigurable hardware devices are proposed and assessed.

2. The second aspect acts at design or compilation time, and that consists of feeding a given system level design methodology with scheduling and placement algorithms for dynamically reconfigurable architectures. This thesis assumes that the Design Space Exploration (DSE) of the RSoC is OS-driven. Several scheduling and placement algorithms are implemented in addition to the newly proposed algorithms. The suitability of these algorithms for C++/SystemC-based RSoC design methodology (e.g. OVeRSoC methodology) is established.

The rest of the thesis consists of 6 chapters and they are organized as follows:

Chapter 2 gives a comparative presentation of different implementation architectures for signal processing in order to see which category fits the dynamically reconfigurable hardware devices. The chapter concludes with explanation on how the above mentioned second aspect of this research can be applied to a Platform-Based Design approach.

Chapter 3 starts with a theoretical background on scheduling problems. In order to point out similarities between microprocessors scheduling and reconfigurable hardware devices scheduling, the chapter first presents both problems. It then emphasizes specific challenges raised by the scheduling of hardware tasks on dynamically and partially reconfigurable arrays. The chapter ends with a literature review on placement strategies and the resulting reconfigurable hardware fragmentation.

Chapter 4 relies on the previous chapters and on preliminary experiments to draw the proposed methodology for scheduling and fitting online real-time hardware tasks on DPRHW[22]. Based on these accurate experiments, the chapter demonstrates the need for a trade-off between the scheduling algorithm and the underlying placement strategy. The chapter emphasizes the importance of providing more than one size and/or shape per hardware task. In addition, Chapter 4 presents different models and related metrics used to assess the performance of the aforementioned scheduling and placement algorithms. The end of the chapter discusses on how these models and metrics

[22] DPRHW : dynamically and partially reconfigurable hardware devices.

can be used at system level design stage with any SystemC-based SoC architecture exploration (e.g. OVeRSoC methodology for DSE [23]).

Chapter 5 is an in-depth study of the designed algorithms that deal with the online real-time scheduling of hardware tasks. The proposed *multi-shape scheduling* algorithms are backed up with novel reconfigurable areas management strategies which are suitable for *online looking-ahead scheduling* algorithms. Along the chapter, whenever possible, the suitability of the proposed scheduling algorithms for online real-time problems is emphasized.

In Chapter 6, performance of the scheduling algorithms and placement strategies proposed to schedule online real-time tasks on DPRHW are presented and discussed using simulation results. These latter are expressed through metrics such as the algorithms runtime overhead, the tasks rejection ratio, the reconfigurable hardware utilization ratio and the scheduling makespan.

Chapter 7 concludes the thesis. The review of the work presented in the thesis is summarized, highlighting the contribution to the body of knowledge. The results from all the experiments are summarized to establish the primary hypothesis of the thesis, regarding the online real-time scheduling of hardware tasks on DPRHWs. This chapter also suggests possible future directions of the research, and emphasizes the RSoC design challenge aspect.

[23] Design Space Exploration

Chapter 2

Dynamically Reconfigurable Architectures *vs* Implementation Alternatives

2.1 Introduction

The previous chapter introduced recent advances in embedded digital signal processing (DSP) applications and discussed various design challenges brought by these advances. The chapter also pointed out the exponential increase of bit rates that require current and future communications applications (up to 100 Mbits on 4G wireless channels). Through FPGAs, the also presented how the development of reconfigurable architectures is bringing new considerations in embedded SoC design.

This chapter will present the most known digital architectures used to implement Digital Signal Processing functions in embedded systems and their distinctive features. The chapter slightly discusses the assets and the drawbacks of each architecture, according to criteria such as time-to-market, performance, price, silicon area, development ease, power consumption and feature flexibility.

The processors platforms for DSP implementation can be classified in two main families:

1. Software or sequential processors
2. Hardware or parallel processors (sometime used as co-processors).

However, the new trend is to combine both sequential and parallel processors in an appropriate proportion in order to achieve a given goal. The chapter will also point out the suitability of reconfigurable computing systems for future embedded System-on-a-Chip applications. The chapter ends by an in depth presentation of reconfigurable hardware architectures, with an emphasis on FPGAs architecture.

2.1.1 The Switch from Analog to Digital Signal Processing

In its earlier days, Signal Processing was performed using analog circuits. These circuits were processing signals in their continuous form. They consist of passive (resistances, capacitances, diodes, inductances, etc...) and active (transistors, integrated circuits, etc...) components. But thanks to the advances in digital circuit design in general and microprocessor technology in particular, Signal Processing progressively moved to the digital domain. Today, Digital Signal Processing is used in a wide range of applications (wireless communications, medical imaging, avionics, automotive, etc...). Figure 2.1 illustrates the basic principle of a DSP system in which an analog signal used as input is first converted to a sequence of numbers, which are digitally processed to achieve a given purpose. The numeric results are then converted back to produce the desired analog output.

Figure 2.1: A simplified representation of a Digital Signal Processing System

Processing digital data brings out many advantages over analog signal processing. Particularly, digital data are easier to manipulate (multiplexing, filtering, compression, errors detection and correction, arithmetic operations, etc...). Indeed, programmable processors provide all the flexibility of software programming. However the frequency range in DSP processors is limited compared to analog circuits because of the Analog-to-Digital Converter (ADC) and the Digital-to-Analog Converter (DAC) frequency limitations. In addition, DSP solutions tend to be more complex and consume more power.

2.1.2 The most common DSP Functions

Nowadays, DSP applications are ubiquitous. They can be found in many consumer devices including mobile handsets (such as mobile phone, personal assistant, digital camera, GPS, etc.), TVs, DVDs players, games console, etc. Hence, the main applications of DSP are audio signal processing, digital image processing, speech processing and digital communications. The most common DSP functions found in those applications are listed below:

- Filtering
- Transform (e.g. Bilinear, Fast-Fourier, Discrete-cosine)
- Convolution
- Modulation and demodulation (e.g. MIMO, QAM, etc.)
- Multiplexing and demultiplexing
- Signal generation

2.1.3 Software vs Hardware Platforms

DSP algorithms could be implemented in embedded electronic systems using different platforms. Choosing a given platform depends on the design trade-offs such as performance, power and price to be achieved. The most common platform implementation of DSP functions are:

- General Purpose Processors (e.g. Pentium, PowerPC) and Microcontroller Units (e.g. ARM Cortex-M3). RISC architectures are more used.
- Programmable Digital Signal Processors (DSP) which are digital processing oriented microprocessors.
- Application-Specific Integrated Circuits (ASICs) which are dedicated components optimized to run specific DSP functions at the best performance and the lowest power consumption.
- Programmable hardware devices (e.g. FPGAs or PLDs) which are components capable of implementing any DSP algorithms in a parallel way and which provide a kind of hardware programmability.
- Other variants such as Application-Specific Instruction set Processors (ASIP) or Application-Specific Standard Parts (ASSP), coarser grain reconfigurable devices and RISC/GPP architectures. Those variants combine specific implementation platforms cited above in order to overcome some of their drawbacks.

Those implementation platforms could be globally divided in two main families, depending on how computations are performed: Software (or sequential) implementation and Hardware (or parallel) implementation.

2.2 Software Implementation Platforms

Software implementation platforms are based on *Von Neumann machine* in which a single instruction (or operation) is performed at a time, as shown in figure 2.2 (a). General Purpose Processors (GPP), Microcontrollers (MCU) and Digital Signal Processors (DSP) are such platforms. They are denoted as sequential architecture processors since in principle each operation is executed in sequence on a single Arithmetic and Logic Unit (ALU) circuit controlled by an instruction memory (figure 2.3). By changing the content of the instruction memory in an appropriate way (using the Instruction Set of the processor), almost any function or algorithm could be implemented as a sequence of basic operations (figure 2.2(a)). As shown in figure 2.2(b), each instruction is implemented following the traditional Von Neumann computer cycles of Instruction Fetch (IF) and Decoding (D), operands Read (R), instruction Execute (EX) and result Write back (W). Hence, five cycles (one at a time) are needed to achieve a single instruction, in contrast with parallel architectures. Consequently, sequential processors are totally flexible and capable of handling tasks from a very wide range of applications. Depending on the application, this full flexibility could be at the expense of power consumption and/or a low performance or quality of service. Hence, trade-offs between flexibility and performance of sequential processors had led to several architectures each optimized for a given goal or a given class of applications.

2.2.1 General Purpose Processors (GPPs)

A GPP is the most popular sequential processor. It is a single integrated circuit that mainly contains a Central Processing Unit (CPU). As shown in figure 2.3, a GPP relies on a *Von Neumann* architecture that consists of a CPU (hosting a Control Unit and an Arithmetic and Logic Unit), a single central memory which holds both instructions and data, and an Input/Output unit for external communication. The Control Unit (CU) extracts instructions from memory, decodes and executes them, calling on the Arithmetic and Logic Unit (ALU) if necessary in order to perform arithmetic and logical operations on read operands, and then writes back the results in data memory.

One limitation of Von Neumann architecture is known as the *Von Neumann bottleneck* where

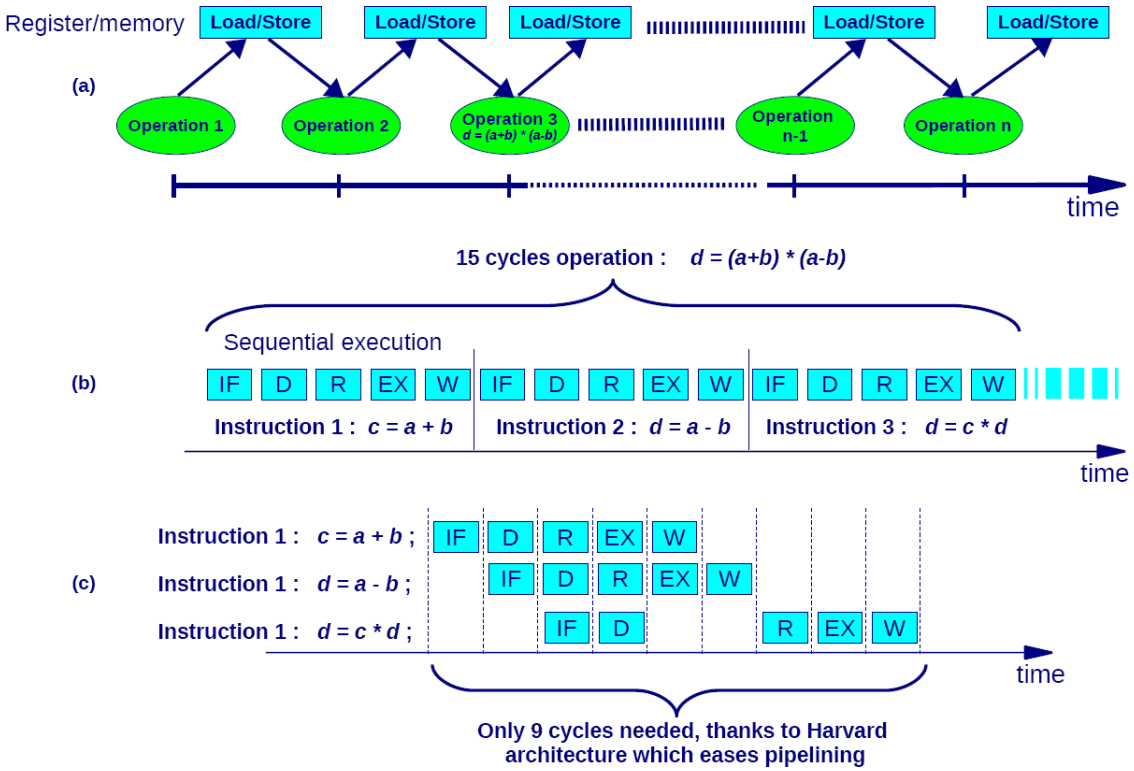


Figure 2.2: Sequential execution. (a) a single operation at a time (b) sequential execution (c) pipelined execution providing higher throughput.

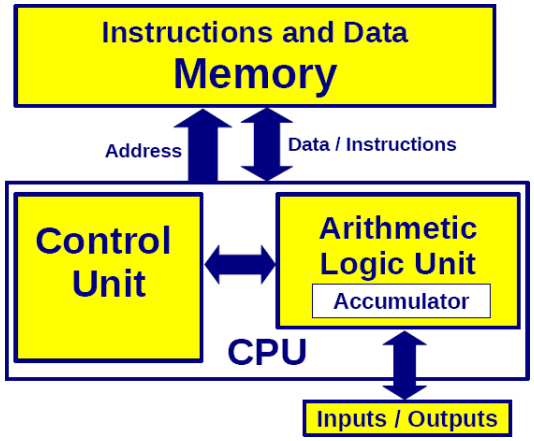


Figure 2.3: The Von Neumann architecture.

the throughput (data transfer rate) between the CPU and memory is limited compared to the amount of available memory. On one hand, the CPU processing speed and memory size have constantly increased following the Moore's law. Pentium and PowerPC processors are a few examples of such architectures that can easily be clocked at 2GHz or more. Hence, they are used in systems such as personal computers, workstations or any other application where power consumption is not a predominant concern. On the other hand, the Von Neumann architecture derives less benefit from the Moore's law as both data and instructions are accessed in the memory through the same port as illustrated in figure 2.3, thus limiting the transfer rate. This architecture has evolved over the years to reduce this bottleneck.

A way of reducing the *Von Neumann bottleneck* is to store data and program in two separated memories. It is the philosophy behind *Harvard architecture* that enables concurrent instruction and data access and rises the bandwidth between the CPU and the memory. Nowadays, Harvard architecture (or a modified Harvard architecture) is used in DSP processors and most of Microcontrollers. This architecture eases Instruction Level Parallelism (ILP or the so-called pipelining) as some of the five cycles (cited above and illustrated in figure 2.2(b)) could be performed concurrently if they do not belong to the same instruction. For instance, Instruction Fetch (IF) and operands Read (R) cycles need to access respectively instruction memory and data memory. As depicted in figure 2.2(c), if belonging to two different instructions, these two cycles could be run concurrently thanks to the fact that instruction memory and data memory have separated access ports. Even the execution (EX) cycle of a third instruction could be on the run at the same time. These overlappings in instruction cycles execution are limited by dependency between instructions, especially when a given instruction is fed by data resulting from another one. Pipelining adds more parallelism in the architecture. The resource (silicon die) utilization ratio and instructions execution throughput are increased accordingly.

To summarize, in a GPP, very high performance at high clock rates comes at the cost of power consumption.

Microcontroller units (MCUs) integrate on a single chip a processor core, memory, and programmable input/output peripherals. They are processors tailored for control purposes. Indeed, increasing the clock rate of CPU-based architectures to meet the rising requirements of embedded applications will never suffice. As previously stated, CPU-based architectures can easily run at high clock speeds and provide a flexibility that allow them can handle any signal processing function. But they are not suitable for embedded applications because of the resulting power

consumption. Thus, many vendors have designed lower speed MCUs (compared to GPPs) in order to target embedded power-aware applications. As they are control oriented, MCUs are capable of running embedded real-time operating systems. However, the new trend is to derive many sub-families of MCUs, where each sub-family offers specific extensions for a given application domain. For example, some MCUs provide a good system trade-off for some basic DSP applications by coupling some DSP functionalities with their CPU. Therefore their instruction set is enriched with dedicated DSP instructions. The ARM Cortex-A8 is an example of an MCU where the *ARM NEON* SIMD [1] accelerator is attached to the CPU. Even with these improvements, MCUs are far from meeting processing requirements of highly intensive DSP applications since they are not designed for this purpose. One example among many is their numeric accuracy which is usually very low and rarely reaches 32 bits. This is not sufficient in most of embedded DSP applications.

Multiprocessor architectures are another solution of increasing performance level by adding more parallelism in the architecture while limiting the clock rate. They enable more instruction level parallelism (ILP), but remain unsuitable for embedded devices because of the resulting power consumption.

2.2.2 Programmable Digital Signal Processors (DSPs)

Digital Signal Processors are programmable processors optimized for digital signal processing oriented applications. Thanks to their ever decreasing power supply, DSP processors are also power efficient. Thus, they are very well suited to mathematically intensive applications embedded in SoCs. Their Harvard architecture along with other architectural improvements attempt to achieve a kind of parallelism in order to offer a high MIPS (Million of Instructions Per Second) and MMACS signal processing performance that is competitive with FPGAs and ASICs.

In addition, DSP processors have been proven excellent for short time to market. Indeed, most of digital signal processing algorithms are written in C and C-like programming languages easier to implement on sequential processors. Similar to other software programmable processors, many tools (compilers, code generators) and libraries (DSP functions, IPs) have been developed over the years to ease DSP implementation and code reuse. Hence, a DSP processor is orders of magnitude easier to target than a hardware implementation platform, even if in some exceptional cases some portions of the code are written in assembling language to optimize the implementation.

[1] Single Instruction Multiple Data

Nowadays, unlike the above-mentioned general-purpose DSP processors, DSPs vendors are moving to a more market-specific approach in DSPs architectures. The new trend is to design DSPs that target a more specific market. In doing so, these DSPs architectures provide the required performance and propose various solutions for the specific market without sacrificing their programmability and their reusability. Examples of DSP processors are TI's TMS320C6000 (with its Multi-MAC VLIW architecture), TI's C55, TI's C64, ADI's Blackfin and CEVA-X families, etc.

Summarizing, sequential processors are flexible and easy to program. They provide the best silicon area efficiency since a fixed structure (e.g. a single ALU) is used sequentially to perform a set of micro-instruction under the control of a Control Unit. They are slower in terms of computation throughput and more power-hungry as increasing operation speed (clock rate) increases the power consumption accordingly. As architectures are continuously improved, the gap between different sequential processors presented above is narrowing , as most of the processors are based on Harvard architecture (see figure 2.20 on page 62).

In this thesis, an application (resp. a task) designed to run on a software platform is referred to as software application (resp. software task).

2.3 Hardware Implementation Platforms

In a hardware implementation, processing is undertaken in parallel. Each instruction is implemented as a custom logic circuit. Hence, several intructions mapped directly in hardware can be executed at the same time (in single cycle), in contrast with sequential processor. One example is illustrated in figure 2.4 where a whole Finite Impulse Response (FIR) filter is implemented as a single instruction which is performed in one clock cycle. Operations needed to implement the N^{th}-order filter (N multiplications and $N-1$ additions) occurred in parallel at each clock cycle; the only constraint being the signal propagation through the longest path between the input X_n and the output Y_n of the logic circuit. One output sample is delivered per clock cycle, in contrast with a software implementation. Indeed, in a Von Neumann architecture the above $2N-1$ arithmetic operations are sequentially done on the single Arithmetic and Logic Unit (ALU) as basic instructions which cannot overlap.

This example of a FIR filter (figure 2.4) seen as an instruction also shows that the size and the complexity of an instruction are arbitrary. Many instructions could be pipelined in order to implement a bigger instruction or operation. Doing so, instructions are performed in parallel at

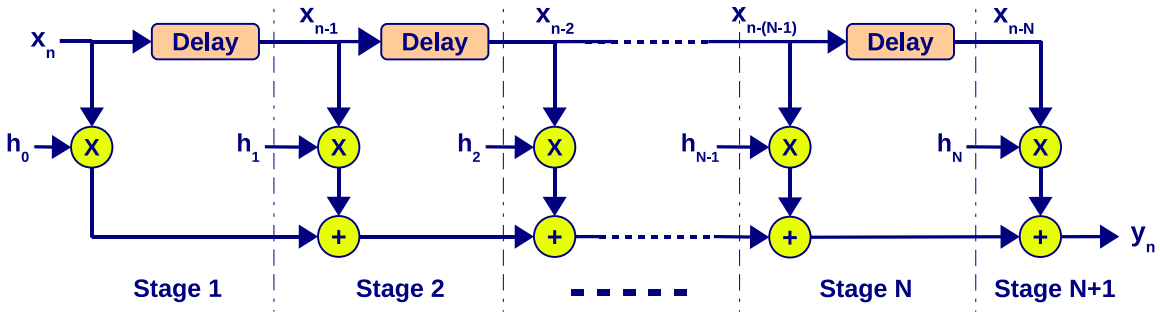


Figure 2.4: Dataflow representation of one instruction performing an N^{th}-order ($N+1$ taps) FIR filtering.

each stage of the pipeline. The depth of the pipeline is only limited by data dependencies within the implemented application and the available implementation resources on the circuit (e.g. number of gates in an ASIC or number of Configurable Logic Blocks in an FPGA).

Thanks to the fact that hardware implementation allows the designer to tailor their circuit for a specific application with maximum parallelism, it yields highest performance in terms of throughput, but lacks flexibility. Fortunately, today's FPGAs tends to overcome this lack of flexibility by enabling runtime reconfigurability.

In any hardware implementation platform, algorithms or functionalities are mapped using Hardware Description Languages (e.g. VHDL, Verilog, etc.) to describe hardware. Such languages are not suitable for algorithms transcription as it contradicts the traditional sequential way of human thinking. Even if HDLs are moving to higher levels of abstraction over the years, writing programs is still a low level and time consuming task for designers. HDLs along with development environments and designers expertise are far from being as mature as high level C-like sequential languages targetting sequential processors (e.g. automatic code generation, compilers, etc..).

In this thesis, an application (resp. a task) designed to be implemented on a hardware platform is referred to as hardware application (resp. hardware task).

The following section presents a couple of hardware implementation platforms.

2.3.1 ASIC Implementation

ASIC implementation is the perfect example of a parallel implementation. Today, ASIC remains the best candidate for computationally intensive applications (even in real-time DSP embedded

ones). Algorithms are directly mapped into silicon through hardware gates. These gates are tuned to achieve the highest performance and the lowest power consumption while occupying the smallest silicon area. For example, if both designed at 90nm technology, cell-based ASIC is faster, can be up to 40 times smaller and consumes about 10 times less power compared to SRAM-based FPGA (Bolsens, 2005; Kuon and Rose, 2006). However, ASIC suffers from its lack of flexibility, since each ASIC chip is designed and manufactured for a specific purpose, and could not be changed or upgraded anymore. For example, as pointed out in Chapter 1, using ASIC technology to achieve Software Defined Radio requirements means designing one ASIC chip for each standard (or function) in the system. In a system that contains many ASICs, one ASIC would be used at the same time, while others would remain unused or in an idle state. This approach leads to products with very large silicon area, high power consumption and consequently, non cost-effective. In addition, modifying a standard or adding new feature leads to a complete redesign of its ASIC.
Embedded applications are facing exponential growth of their requirements, leading to complex and denser ASICs and SoCs design, integration and validation. This growing complexity lengthens the design process and consequently delays the time-to-market (TTM). The verification process became the key point of the design since any failure leads to a costly complete redesign. Furthermore, the non-recurring engineering (NRE) cost of ASICs is also mainly increasing drastically as the costs of creating masks in today's deep submicron geometries are becoming unaffordable as shown in figure 2.5. Design tools are struggling to handle challenges posed by each new technology generation. Consequently, an ASIC product could become obsolete before getting into the market. This makes ASIC implementation cost-effective only for high-volume products. However, in contrast with Von Neumann-like architectures where the clock frequency is reaching a plateau, ASICs will still benefit from *Moore's law* for a while.

2.3.2 Fine and Coarse Grain Reconfigurable Arrays Implementation

Field Programmable Gate Arrays (FPGAs) are known as the finest grain reconfigurable hardware. They can implement fairly small pieces of logic (1-bit level). Hence, they are the main alternative to ASICs. If big enough, an FPGA could implement any digital system. Consequently, they have been used for ASIC prototyping. As previously stated in Chapter 1, TTM and RFT are the two factors which fostered FPGA development. The two main technologies are SRAM-based FPGAs and anti-fuse-based FPGAs. Depending on the technology used, an FPGA could be configurable only once or many times. For example, the main feature of SRAM-based FPGAs is to be reconfigurable

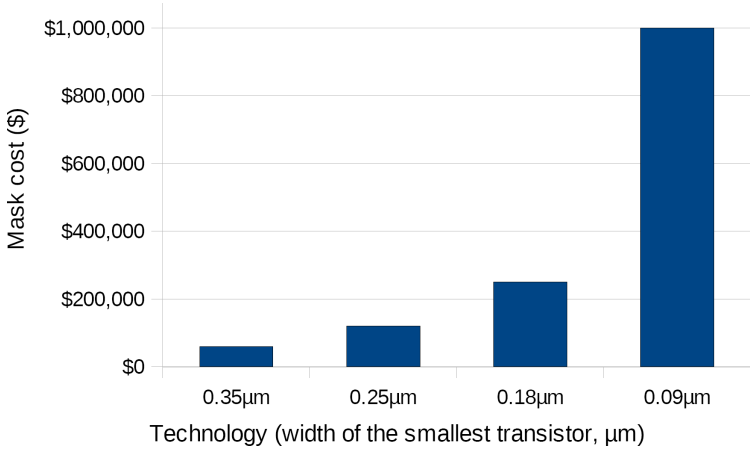


Figure 2.5: Mask cost exponentially grows with technology.

an unlimited number of times. They hold their configuration in a static memory. They ally flexibility and performance. Their programmability is not as good as sequential processors, but their performance is not far below that of ASIC. However, dynamically reconfigurable SRAM-based FPGAs are seen as an enabling technology for reconfigurable computing. The anti-fuse FPGAs will not be emphasized in this section, as they do not enable dynamic reconfiguration.

A trend in programmable logic devices is to develop coarse grained reconfigurable architectures. In these latter, the granularity is at word level (8, 16 and/or 32 bits level). This granularity provides a better performance (thanks to embedded hardwired blocks) and reduces the costs in terms of configuration logic, but at the cost of flexibility.
A more detailed study of fine grain (FPGA) and coarse grain reconfigurable arrays technology is presented later in this chapter in sections 2.5 and 2.6.

2.4 ASIP/ASSP Implementation

Even though agreeing that SoCs including one or many CPU subsystems and reconfigurable hardware in addition to specific hardware blocks are the future of embedded systems design, many studies (e.g. Hartenstein, 2001a,b; Rabaey, 2001; Jerraya, 2004) argue that a universally efficient implementation platform is an illusion. According to them, a general purpose computing technology platform could never meet different combinations of requirements of different applications. In addition, Gatherer et al. (2004) stated clearly that FPGAs and reconfigurable processors are only a stop-gap solution in the march towards an all-soft SDR, despite their numerous advan-

tages. Hence, even dynamically reconfigurable platforms have to be optimized for a given class of applications. Platform-Based design concept presented below aims to follow this trend. Similarly, one could say that configurable processors cited above are used to generate Application-Specific Instruction set Processors (ASIP). ASIPs or ASSPs (Application-Specific Standard Parts) are circuits optimized for a class of products or a particular application domain. ASIPs/ASSPs are becoming an alternative to ASIC design, thanks to their shorter time-to-market. Indeed, once they are designed, different versions or new generation of a product could be derived from the same design. ASIPs/ASSPs are increasingly designed as SoCs integrating multiple cores (processor cores and semi-programmable circuits). Some application oriented platforms like TI OMAP cited in the paragraph 2.7 below are referred to as ASSPs. One can say that ASIP/ASSPs are between ASICs and FPGA, since they are less flexible than FPGA, but provide a better performance.

2.5 Fine-grained Reconfigurable Hardware Devices

2.5.1 Introduction

As previously said, in a fine grain reconfigurable architecture, the granularity is at the bit-level. This means that even a one-bit logic function could be implemented on this architecture. This section presents fine grain reconfigurable architectures and mainly SRAM-based FPGAs as they are far the most used to implement dynamic and partial reconfiguration. However, this study is also valuable for other devices like CPLDs or other antifuse-based FPGA, as they rely on the same principle.

2.5.2 FPGA Architectures

A basic FPGA consists of three major elements as shown in figure 2.6:

(i). Combinational logic blocks

(ii). Programmable interconnects

(iii). Programmable Input/Output pins

Combinational logic blocks are configurable blocks used to implement custom logic functions. They are denoted as logic elements (LEs) or configurable combinational logic blocks (CLBs). These blocks are generally arranged as a two-dimensional structure and are surrounded by programmable interconnects. The circuit to be implemented is divided into small modules, each fitting in a

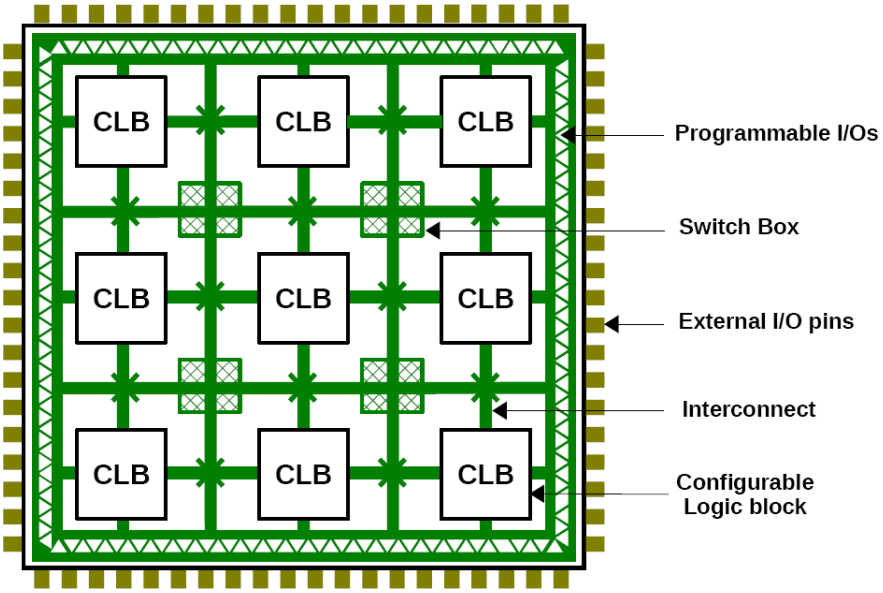


Figure 2.6: Simplified structure of an FPGA.

logic block. Several blocks are then interconnected using programmable interconnects in order to implement the whole circuit, if too big to fit in a single block. If the FPGA is reprogrammable, the implemented function could be changed by updating the content of the configuration memories. Input and Output pads are assigned using programmable Input/Output blocks (IOBs). IOBs could be also programmable as low-power or high-speed connections. If there are enough implementation resources on the chip, any function could be implemented by interconnecting basic blocks. There are several types of interconnect, depending on the distance between logic blocks to interconnect. Special interconnects are dedicated to clock signals.

2.5.3 FPGA Technology

The two main technologies are :

1. ***Antifuse-based FPGAs***

   In antifuse-based FPGAs, interconnections rely on antifuses which two terminals are separated by a dielectric. Hence, they are of high impedance at their normal state and can be switched to their low impedance state (fused state) by applying a high voltage which melts the dielectric and reduces the resistance. The nodes are then connected for good. For this reason, they are also known as *one-time programmable* FPGAs. The desired logic function

is obtain by connecting basic logic elements in the appropriate way. Antifuse technology, a cheaper technology than SRAM, can achieve higher speeds and occupies less space on the die. In addition, antifuse technology makes more reliable and secure FPGAs compared to SRAM-based, safeguarding against cloning, overbuilding, reverse engineering and radiation bombardment. Consequently, antifuse FPGA technology is the best option for satellite applications. However, as antifuse FPGAs are only *one-time-programmable*, they are not suitable for reconfigurable computing and, therefore, are no more detailed in this thesis.

2. ***Memory-based FPGAs***
   In memory-based technology, memory parts are either SRAM[2] or flash EEPROM[3]. SRAM or flash EEPROM are used to configure both interconnections and logic blocks. Configurations are held in the memory. Consequently, unlike flash EEPROM which are permanently programmed FPGAs, SRAM-based are volatile and the configuration is lost at power off. However, the SRAM approach is the most widely used in FPGA configuration.

2.5.4 FPGA Structures

No matter which technology is used, the overall structure of figure 2.6 remains the same.

Island style architecture

This type of architecture was chosen by Xilinx from the beginning while introducing FPGAs in 1985. The FPGA consists of a planar array of programmable logic blocks with vertical and horizontal programmable routing resources.

Sea-of-gates architecture

In this architecture, logic blocks are spread all over the IC chip and routing resources provide a logarithmic connectivity. Indeed, interconnections are mainly made through neighbor-to-neighbor routes which are faster, in addition to other general routing resources. Sea-of-gates topology was used by Xilinx (in its 6000 series) and by Actel (in its ProASIC family).

[2] Static Random Access Memory

[3] Electrically-Erasable Programmable Read-Only Memory

Hierarchical architecture

Hierarchical architecture is the philosophy of Altera. There are several plans in the FPGA, but these plans are not physical. They correspond to different logic levels. For example, one element of a logic level may contain elements of a lower logic level, leading to the notion of logic hierarchy. Each level uses the topology of island style architecture with dedicated routing for each level. This hierarchical approach both in logic and interconnects provide smaller and more predictable routing delays, and therefore higher operation frequency.

2.5.5 SRAM-based FPGA

This section focuses on SRAM-based FPGAs technology, as they can be reprogrammed an unlimited number of times and even during system operation, enabling dynamic (on-the-fly) reconfiguration. These two features are the main advantages of SRAM technology. In addition, unlike other technologies, SRAM technology uses standard CMOS[4] cells and the whole FPGA can be fabricated with standard VLSI[5] processes. However, SRAM-based technology occupies more chip area compared to other technology, making them relatively expensive. Furthermore, SRAM configuration memory is power consuming even when its contain remains unchanged. As the SRAM contents are lost at power-off, there is a need for an external non volatile storage memory to keep configuration data in order to reconfigure the FPGA at power-on.

LUT-based Logic Elements

In SRAM-based FPGAs, each logic element or configurable logic block (LE or CLB in figure 2.8) is built around a LUT[6]. A LUT can implement any $n - inputs$ combinatorial function (or sequential by adding a flip-flop latch at the output). For example in figure 2.7, a LUT implements a function by storing its truth table in SRAM memory. Hence, an $n - inputs$ function requires a 2^n location SRAM (resp. 3 and 2^3 in the example of figure 2.7). Values of function S in the truth table are pre-stored in the SRAM in a way that each value is at the address corresponding to its combination of inputs (e.g figure 2.7 where values 0 and 1 are stored respectively at addresses 101 and 110). The SRAM is then connected to a decoder which uses the input combination to access the corresponding location and route the correct result of the function to the output.

[4] Complementary Metal Oxyd Semiconductor
[5] Very Large Scale Integration
[6] Look-Up Table

One advantage of LUT implementation over static logic gates is that no matter which function is implemented by the LUT, the delay through the logic element is the same.

Thanks to this fine grain granularity, a full parallelism (spatial implementation) is achieved as much as in ASICs. Hence, one can recreate a complete ASIC design in an FPGA by programming and interconnecting basic blocks. This feature made FPGAs suitable for complex ASIC prototyping.

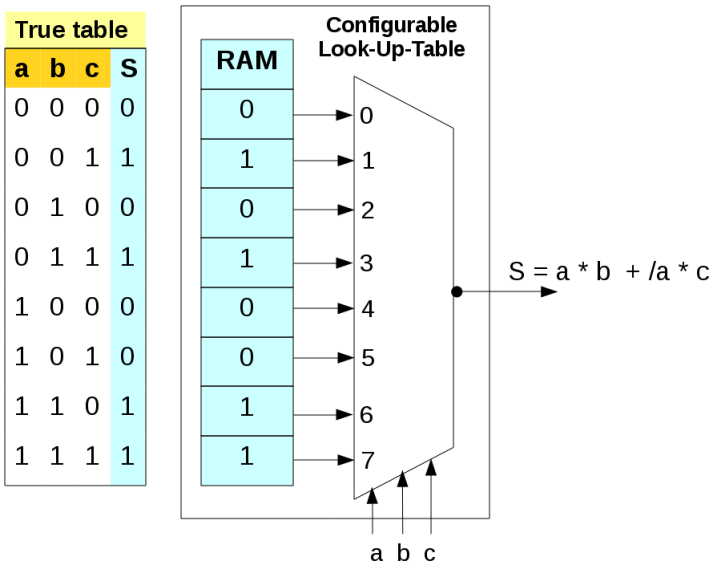


Figure 2.7: Truth table of function S = f (a, b, c) and its mapping using a 3 inputs Look-Up-Table.

As shown in figure 2.8, a LUT-based logic element usually contains an output register which synchronizes, if necessary, the output with a clock. An $n - inputs$ logic element can implement up to 2^{2^n} different functions only by changing the contain of the SRAM.
The size of LUT inputs is typically 4. Many studies have shown that a LUT size between 3 and 6 provides the best trade-off between area optimization and delay (e.g. Rose et al., 1990; Singh et al., 1992; Ahmed and Rose, 2000). More precisely, a 3 to 4 input LUT improves the area, while a 5 to 6 inputs minimizes delay.

Examples of LUT-based basic building blocks in commercial FPGAs

In most of commercial SRAM-based FPGAs, basic logic elements are grouped in a kind of cluster which provides faster connections between the LUTs inside the cluster, in addition to registers (flip-

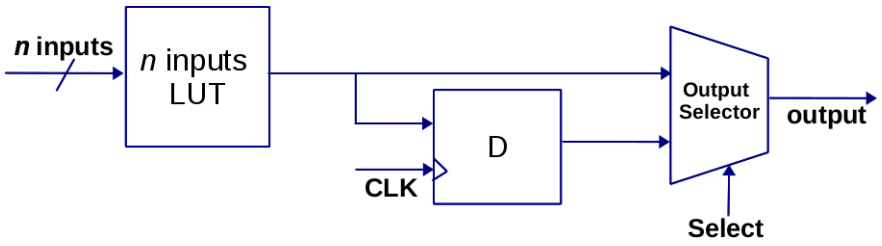


Figure 2.8: A logic element or configurable logic block

flops and latches), multiplexers (used as input and output decoder) as well as some combinational logic for basic computations (e.g XOR-gates, adders, fast carry chain, etc.).

1. ***Adaptive Logic Module in Altera Stratix V architecture (figure 2.9)***

   Figure 2.9 depicts a high-level block diagram of a LUT-based *Adaptive Logic Module (ALM)*. An ALM is the basic building block of logic in the Altera Stratix V architecture. It combines advanced features and efficient logic utilization.

   Each ALM contains a variety of LUT-based resources that can be viewed as two combinational adaptive LUTs (ALUTs) followed by two registers. With up to eight inputs for the two combinational ALUTs, one ALM can implement various combinations of two functions. However, the ALM is completely backward-compatible with four-input LUT architectures. One ALM can also implement any function with up to six inputs and certain seven-input functions. In addition to the adaptive LUT-based resources, each ALM contains two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, and a register chain. Through these dedicated resources, an ALM can efficiently implement various arithmetic functions and shift registers. Each ALM drives all types of interconnects: local, row, column, carry chain, shared arithmetic chain, register chain, and direct link.

2. ***A Slice in a Xilinx Virtex 5 FPGA architecture (figure 2.10)*** In Xilinx FPGAs, Configurable Logic Blocks (CLBs) are the main logic resources. Each CLB element consists of two SLICEs (shown in figure 2.10) and is connected to a switch matrix for access to the general routing matrix. CLBs and therefore slices are arranged array-wise. The two slices within the same CLB belong to different column of slices and do not have direct connections to each other. However, each slice in a column has an independent carry chain which could be connected to the neighboring slice of the same column. As pictured in figure 2.10, every slice contains four look-up tables for logic-function generation, four storage elements, wide-

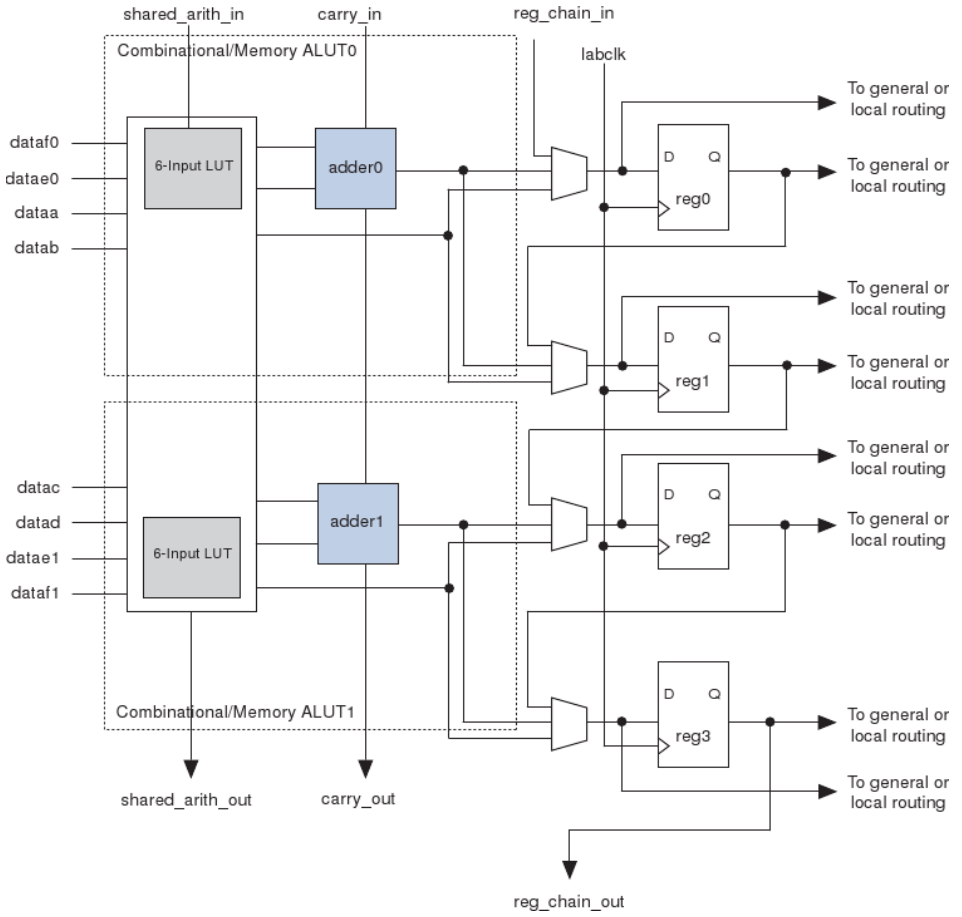


Figure 2.9: An Adaptive Logic Module in Altera Stratix V architecture (courtesy Altera).

function multiplexers, and carry logic. These elements provide logic, arithmetic, and ROM functions. With up to six independent inputs (e.g. A1 to A6) and two independent outputs per LUT in the slice, each of the four function generators can implement any arbitrarily defined six-input Boolean function, or implement two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs. Thanks to multiplexers, the four LUTs could be combined to generate any Boolean function of 7 or 8 inputs in a slice. Furthermore, some slices (SLICEM, not shown in figure 2.10) support additional functions such as storing data using distributed RAM and shifting data with 32-bit registers.

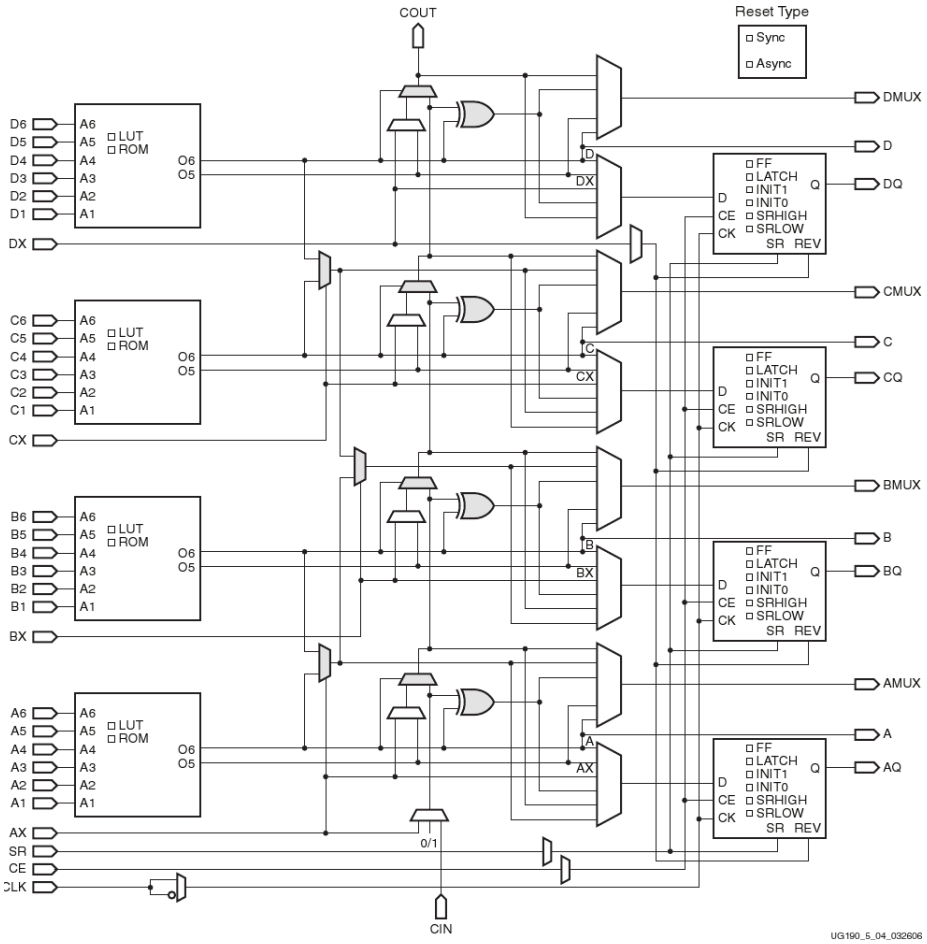


Figure 2.10: A Slice (SLICEL) in a Xilinx Virtex 5 FPGA architecture (courtesy Xilinx).

Routing resources

Most of the chip area (60 to 90%) of an FPGA is occupied by routing used to connect logic blocks between them and to I/O blocks. These programmable interconnects are of different kinds, ranging from short local wires, general-purpose wires, global interconnect and dedicated clock distribution networks. The main reason of using different interconnects is to minimize delay throughout the wire. This is achieved by finding the best trade-off between the area size of the wires, distance separating the buffers inserted along the wires, etc. As they provide low impedance, antifuse-based programmable interconnects are faster than those used in SRAM-based (pass transistors or three-state buffers) or flash-EEPROM based FPGA.

Programmable IOBs

Input/Output blocks control the data flow between FPGA external pins and the internal user logic through programmable interconnects. Input/Output blocks are multistandard, providing different logical levels (e.g. 3.3-V LVTTL[7], multi-voltage LVCMOS, multi-voltage PCI, HSTL[8], SSTL[9], etc.), LVDS[10] channels, SERDES[11] support, EDS[12] protection, DDR[13] compatibility, and numerous programmable features (input, output, tri-state, delay, skew rate, etc.).

Digital Clock Manager

Among routing resources, dedicated clock distribution networks appears as a tree of drivers with buffers distributed throughout it in order to minimize delay skew. DCM digitally manages clocks signals generation and distribution. It also enables delay skew compensation, clock multiplication and division for multi clock designs.

2.5.6 Heterogeneous FPGAs

Today, FPGAs are increasingly used in many computationally intensive applications. Their architecture became more heterogeneous, embedding hardwired blocks as shown in figures 2.11 and 2.12 in order to respond to the market demand. Indeed, its high flexibility comes at the cost of efficiency compared to ASIC. As more gates have to operate in the FPGA than in an ASIC for the same functionality [14], it will consume more power, clock slower and require more silicon area. But more than ASIC and processors, FPGAs leverage the logic density improvements arising from technology scaling (the Moore's law). Hence, they are becoming larger and faster, with on-chip dedicated blocks (memories, multipliers, processors, etc...). These well-designed and well-tested ASIC blocks are more area efficient and faster than their CLB-based counterparts. They avoid the use of considerable amount of logic to implement CLB-resources-greedy functions such as

[7] LVTTL/LVCMOS : Low Voltage TTL/CMOS I/O logic switching levels

[8] High Speed Transceiver Logic, for fast SDRAM

[9] Series Stub Terminated Logic, for reduced latency DRAM (RLDRAM)

[10] Low Voltage Differential Signaling

[11] Serializer/deserializer

[12] Electrostatic discharge

[13] Double data rate

[14] Compared to an ASIC achieving the same functionality and manufactured at the same process technology (e.g. 90nm), an FPGA clocks 10 times slower and uses 50 times larger silicon area per gate. This is due to configuration overhead, logic gates and configuration memories, routing delays, etc....

multipliers or memories. Furthermore, one or many hard core processors are directly integrated within the FPGA[15], along with various network interfaces and communication modules[16]. This new trend could be denoted as *multi-grain* FPGAs, as it combines fine-grain and coarse-grain reconfigurable architecture on the same silicon die. The most common embedded hard blocks are:

Memory Blocks

In dataflow oriented applications like image processing, huge amounts of data need to be regularly and temporary stored during their processing. Consequently, embedding memory blocks within the FPGA became crucial for reducing memory access delays. As depicted in figures 2.11 and 2.12, these blocks are distributed all over the device, providing a very high memory bandwidth suitable for highly parallel applications. These so-called *BlockRAM* (BRAM) modules which are fundamentally 36 Kbits in size, can also be used as two independent 18 Kbit blocks, and provide single port and dual port access. A single FPGA chip may contain up to 16 Mbits[17] memory spread over the chip.

Embedded DSP Blocks

Embedded DSP blocks (figure 2.11) provide many MAC[18] units. Associated with the aforementioned BlockRAM embedded memory modules, they ease the implementation of digital signal processing functions like filters. Hence, a DSP oriented FPGA can run concurrently hundred of MAC units[19] and thereby far exceed programmable DSP processors performance.

High speed I/O transceivers

High-speed serial I/Os are more used today since serial buses are progressively preferred to parallel buses. Effectively, parallel buses need to be synchronized to a clock line, while in serial buses, the clock signal can be implicitly included in the signal. The GX series of Altera Stratix FPGAs family

[15] e.g the PowerPC405-based processor(s) integrated in Xilinx FPGAs (figure 2.12), or the ARM-based processors in Altera FPGAs.

[16] e.g. built-in PCI Express and 100 Gigabit Ethernet in Altera Stratix V family and Xilinx 7 series, both leveraging new 28nm process and design innovations to reduce power consumption

[17] e.g. Xilinx FPGA Virtex-5 XC5VFX200T integrates up to 456 x 36 Kbits blocks or 912 x 18 Kbits blocks

[18] multiply-accumulate

[19] up to 512 multipliers blocks 18-bit x 18-bit in some Stratix V FPGA sub-family

(figure 2.11) and the FX/TX series of Xilinx Virtex FPGAs family (figure 2.12) are examples of FPGAs optimized for high speed serial connectivity. They enable data rate of up to 12.5 Gbps[20], particularly well-adapted to high-throughput telecommunication systems.

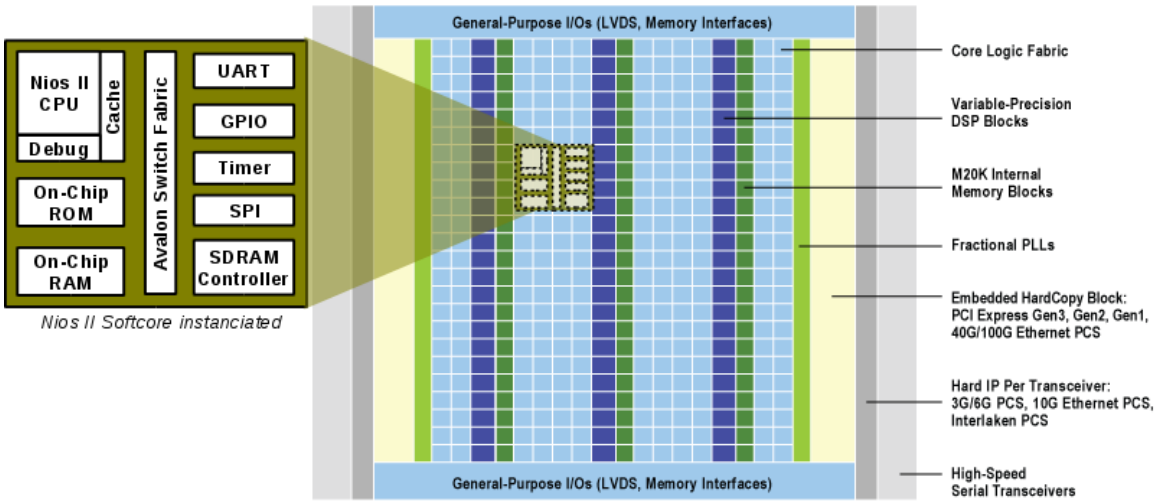


Figure 2.11: Altera Stratix-V floor plan (Altera, www.altera.com).

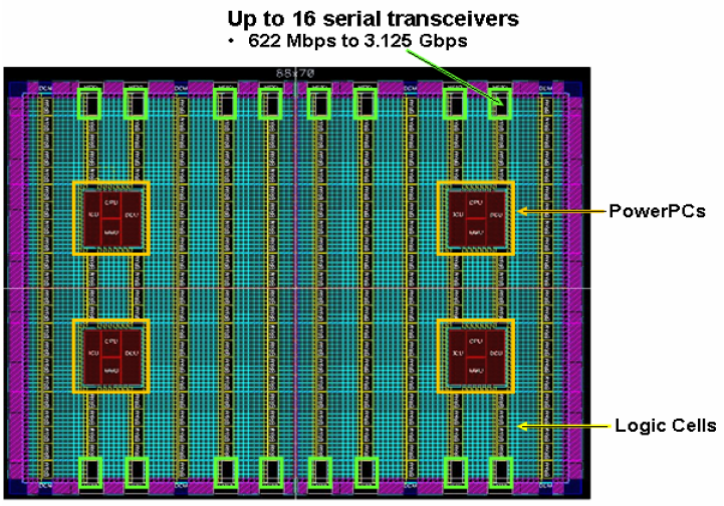


Figure 2.12: Xilinx Virtex II Pro FPGA with up to 4 hard core embedded processors(Xilinx, www.xilinx.com).

[20] 10.3125 Gbps for Xilinx Kintex-7 FPGA Family and 12.5 Gbps for Altera Stratix V FPGA Family

Embedded hard/soft core processors

Figures 2.11 and 2.12 depict the floor plan of Altera Stratix V FPGA and Xilinx Virtex II Pro FPGA. The latter could embed up to four IBM PowerPC 405 RISC hard core processors within its structure (figure 2.12). The other option is to embed instantiated soft core processors instead (e.g. Altera Nios II soft core in figure 2.11), as they provide more flexibility on architecture and therefore instructions set. By merging hard or soft microcontroller cores in FPGA chips, FPGA manufacturers have launched the SOPC[21] age. A SOPC integrates on a one die one or several programmable processors (GPP/RISC, DSP), memory, reconfigurable fabrics and some dedicated blocks. Hence, it combines the programmability and well-tried connectivity of microprocessors (buses, uart, Ethernet) with the high performance of ASICs and the reconfigurability of FPGAs. Besides, the merged soft core microprocessor is customizable (pipeline depth, cache size, etc.) by the designer. Further, as shown in Meyer-Bäse et al. (2006), this soft core allows the designer to extend its instruction set by implementing some acceleration units in the form of custom instructions. This processor is referred to as configurable processor. Tensilica's Xtensa [22], Altera's NIOS and Xilinx's Microblaze soft core processors are a few examples of configurable processors. In the literature, Coarse-grained reconfigurable arrays presented below are also referred to as configurable processors.

2.5.7 FPGA Design Flow

Traditional design flow

In the traditional FPGA design flow pictured in the grey coloured part of figure 2.13, a design is created using a HDL (e.g. VHDL or Verilog) or a schematic capture environment, then synthesized, placed and routed for a specific FPGA. The minimum steps of the design flow are listed below:

1. Design description in HDL (VHDL, Verilog, SystemC), or through a graphical entry tool that generates the corresponding HDL code.
2. Simulation to verify the correct behavior.
3. Synthesis.
4. Mapping, placement and routing for a specific FPGA.

[21] System on a Programmable Chip
[22] Tensilica (www.tensilica.com)

5. Binary configuration file (bitstream) generation and loading on the targeted FPGA.

Nowadays, most of the above mentioned steps are push-button operations. However, as previously stated, the first step (design entry) using HDLs is still an almost manual and time consuming task. Indeed, even if C-like languages along with their C-to-hardware compilers are increasingly investigated[23], the resulting FPGA implementation (in case of a successful synthesis) is far from competing manual design in terms of area efficiency and timing.

The generated bitstream is downloaded into the FPGA through configuration interfaces like JTAG, SelectMap or Slave Serial ports. These ports enable numerous reconfiguration techniques, bitsream encryption and bitstream readback (through the SelectMAP and JTAG interfaces). As SRAM-based FPGAs are volatile, bitstreams are stored in an external non-volatile memory and are used by a CPLD[24] to configure the FPGA at each power on.

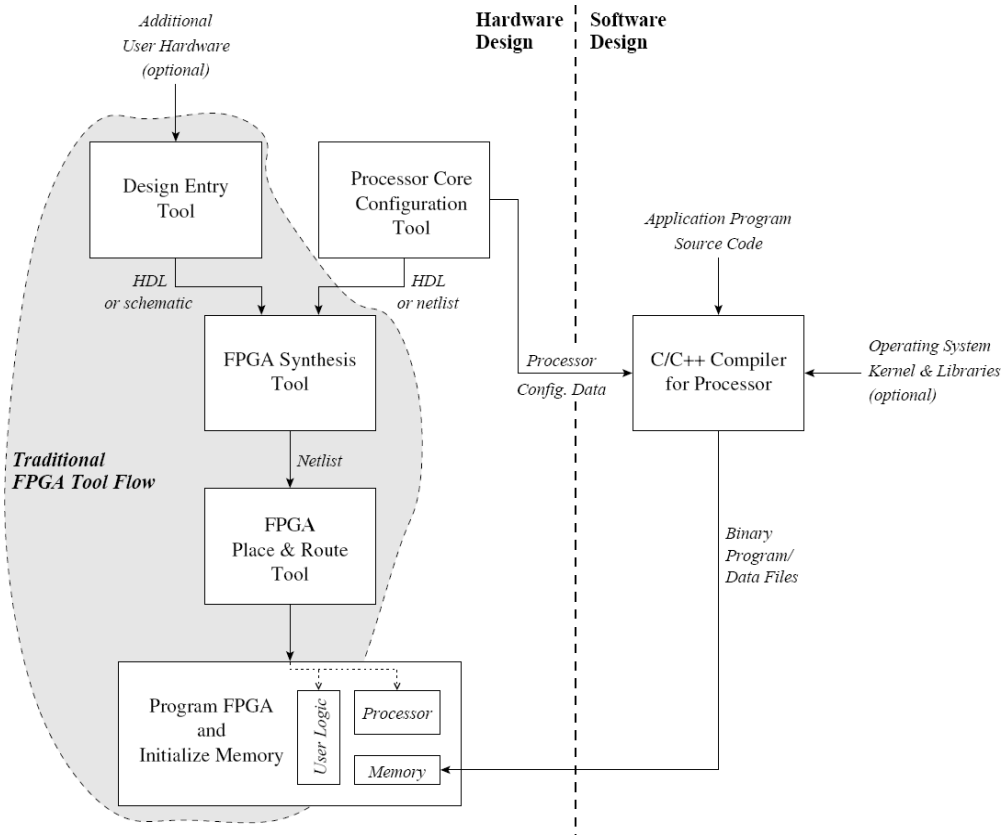


Figure 2.13: Design flow for FPGA-based systems embedding a programmable processor.

[23] SystemC (www.systemc.org), Celoxica (2000), ImpulseC (www.impulseaccelerated.com), etc...)
[24] Complex Programmable Logic Device

SOPC design flow

As FPGAs enable a SOPC design approach, the design flow in that case is slightly different. The design of the software part of the SOPC is integrated in the flow and the whole process is depicted in figure 2.13. The main FPGA manufacturers tools and other third party companies provide complete IDE[25] for SOPC design. Such IDEs are capable of designing both the hardware part and the software part of the design by:

- supporting the traditional design flow that automatically runs the whole process, resulting in a single binary file (e.g. *.bit* file) which contains the configuration data for the FPGA.
- providing the environment for writing the code and developing the application to run on the embedded processor (generating the executable file), generating all communication media (bus, UART, JTAG, Ethernet, etc.), and finally producing the binary file containing the configuration data for FPGA and the binary code for the processor.

2.5.8 FPGA Modular Design for Runtime Partial Reconfiguration

Modular design flow is an incremental design approach that differs from the above mentioned traditional design flow. The main drawback of the latter approach is that if a slight change is made on an implemented design, the five design steps (grey coloured part of figure 2.13) have to be completely redone for the entire design, making the redesign or modification process very long.

Originally, *Modular design flow* aimed to partition an application into its natural functional units in a way that each unit corresponds to an independent module. Hence, each module can be separately designed, tested, modified, validated, implemented and even reused in another design. Modules can also be third party IPs (intellectual properties) released either as HDL-described circuits or pre-synthesized netlists.

As *Modular Design* requires a clear partitioning of the design in different modules, a team of designers can work independently and concurrently on different modules and later merge them into one FPGA. This concurrent and hierarchical approach saves time and allows for independent modules modification and validation while leaving others unchanged and stable. The latter feature is exploited while designing dynamically and partially reconfigurable designs for SRAM-based FPGAs. Indeed, reconfiguring a module corresponds to changing a functionality by swapping functional units on the FPGA, each functional unit performing a specific task.

[25] Integrated Design Environment

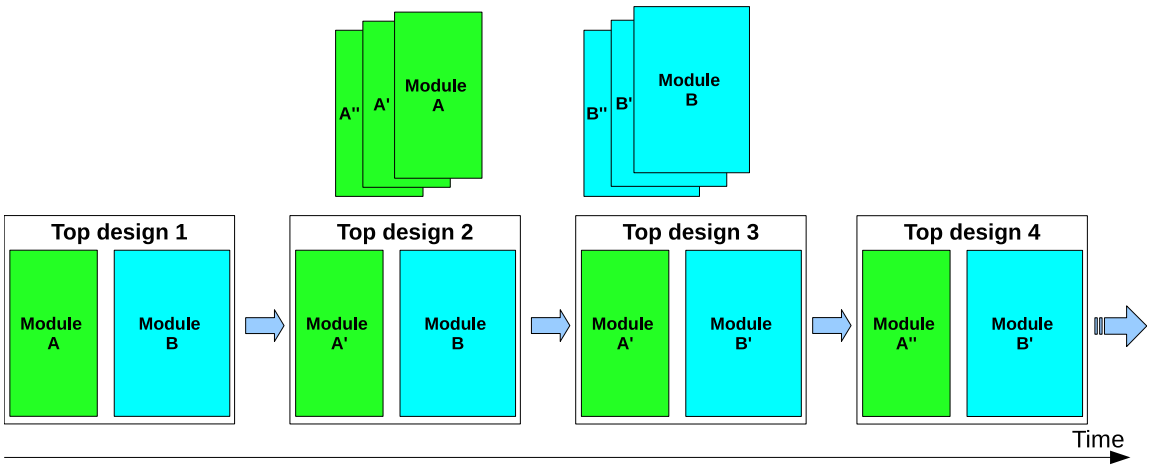


Figure 2.14: Modular design enables dynamic module swapping.

An example of dynamic reconfiguration is pictured in figure 2.14 where the FPGA can contain at most two modules at a time. One can switch from one design to another by reconfiguring at least one module of the FPGA. However, because of current lacks in FPGA design tools for partial runtime reconfiguration, the FPGA is block-partitioned and reconfigurable modules are position-constrained at design time. In the example of figure 2.14, the FPGA is two-blocks partitioned to accommodate two classes of reconfigurable modules, module class A and module class B. Modules of the same class (e.g. modules A, A' and A") are allocated the same block and the size of the module cannot exceed the size of the containing block. There is at most one module in a block at a time and only the modules of the same class can be swapped on their allocated and position-constrained block. Hence, at a certain point in time and if necessary, one can switch from *top design 1* to *top design 2* by swapping *module A* for *module A'* without affecting the *module B*.

The Xilinx ISE encompassing the PlanAhead tool is one example of an enabling design environment for partial and dynamic reconfiguration of Xilinxs' FPGAs. In addition, the Xilinx Application Note (Xilinx, 2004) provides details on two ways for designing partial runtime reconfiguration : *difference-based* and *module-based*. The latter approach uses the above-mentioned modular design flow and generates partial bitstream for different modules.

Let's F being an *m-blocks* partitioned FPGA which can therefore hosts m modules at a time. Given n the number of physically pre-placed modules per block. If it is assumed that all the modules are reconfigurable [26], then the FPGA can implement $n \cdot m$ different designs or parts of

[26] This is rarely the case as in real design and with current design tools (e.g. Xilinx ISE) there is always

designs as shown in figure 2.14.

Hardware virtualization and hardware multitasking

When the dynamically loaded and unloaded modules belong to the same application, this process is referred to as *hardware virtualization* otherwise *hardware multitasking*. Hardware virtualization stands in that, thanks to dynamic reconfigurability, the total amount of logic on an FPGA virtually appears bigger than it really is. Indeed if the modules of an application are properly scheduled, an FPGA of size $W \cdot H$ can fit the application even if the sum of the areas of the modules composing the application is bigger than the size of the FPGA as expressed in equation 2.1.

$$Virtualization \implies Appl_{(total\_size)} = \sum_{i=1}^{m} w_i \cdot h_i > FPGA_{size} = W \cdot H \tag{2.1}$$

where W and H (resp. w_i and h_i) are respectively the width and the height of the FPGA (resp. of *module i*), $W \cdot H$ the total size of the FPGA (resp. $w_i \cdot h_i$ the size of *module i*) and m the number of modules composing the application *Appl*.

As stated in the previous chapter, Software Defined Radio is an example of a promising application field for reconfigurable architecture. The flexibility and the high performance required by the radio may be achieved by modules swapping as described earlier. Hence with proliferation of standards, the radio can dynamically switch from one standard to another by swapping corresponding functional units in and out of a limited and cost effective silicon die, where the cost of an integrated circuit is related exponentially to die size.

2.5.9 Coupling with the Host Processor

Reconfigurable hardware devices are used in several ways as hardware acceleration units to boost the performance of a computing system.

Shown below are different scenarios of coupling the reconfigurable hardware fabric to the host processor. These coupling scenarios also define communication cost. Indeed, communication is of high cost in embedded systems as it has a great impact on the global performance of a system. Coupling scenarios tend to complicate the design process, as the designer faces a new programming model for targeting a new computing resources model. One of the main issues is to identify which coupling approach is likely to yield particular performance benefits in an application domain,

at least one fixed parts of the design which is not reconfigurable and which hosts a hard-core or a softcore processor that manages the reconfiguration process for example.

and to know whether or not using reconfigurable hardware is the most profitable. A significant challenge is to find a suitable architecture for the communication media which interfaces different parts of the system (FPGA, programmable processors, etc...). Here below are three ways of using reconfigurable logic in a SoC:

1. ***Reconfigurable Hardware used as a peripheral co-processor*** *(figure 2.15)*
   In this design approach, any process intensive code (encryption/decryption, pattern recognition, etc.) within a given application, migrates to the reconfigurable hardware device (e.g. FPGA). The reconfigurable fabric accelerates the computations via devices such as an FPGA board, or a multi-FPGA board. Consequently, a programmable GPP or DSP is relieved of some complex processing tasks (mostly dataflow oriented tasks) to increase the overall system throughput. In this case, the programmable processor and the reconfigurable logic could be connected to the same bus (or some kind of I/O bus) and therefore communicate through the bus protocol. Communicating through a bus leads to a higher communication cost, and consequently worths only if the speed improvement brought by the reconfigurable logic exceeds the overhead of transferring data through the communication media. This is achieved either by implementing applications which do not need regular transfer of huge amount of data between the programmable processor and the co-processor, or by implementing a whole computationally intensive algorithm on the co-processor. Peripheral single-function co-processors have been implemented using dedicated ASIC blocks which provide faster, power-efficient and area-efficient implementation compared to the corresponding FPGA implementation. However, improvement in FPGA technology fills these gaps by providing flexibility (static or dynamic reconfigurability) and by improving upgradability and Time-To-Market.

2. ***System-on-a-Programmable Chip*** *(SOPC - figure 2.16, left)*
   In this approach, the whole system is designed on a single reconfigurable hardware device (e.g. FPGA) embedding some hardwired and optimized IP cores such as softcore and hardcore processors, Input/Output devices, memories, and DSP blocks within its structure. Low communication cost could be achieved especially when using configurable processors where parts of functional units are made of reconfigurable logic, and enable configurable instruction set. Thanks to advances in FPGA technology, the main FPGAs manufacturers (Xilinx, Altera, etc...) have made such reconfigurable systems design possible by releasing products such as Virtex-II Pro, Virtex 4, Virtex 5, Stratix-II, Stratix-5, etc, along with

softcore processors (NIOS II, microblaze, etc...).

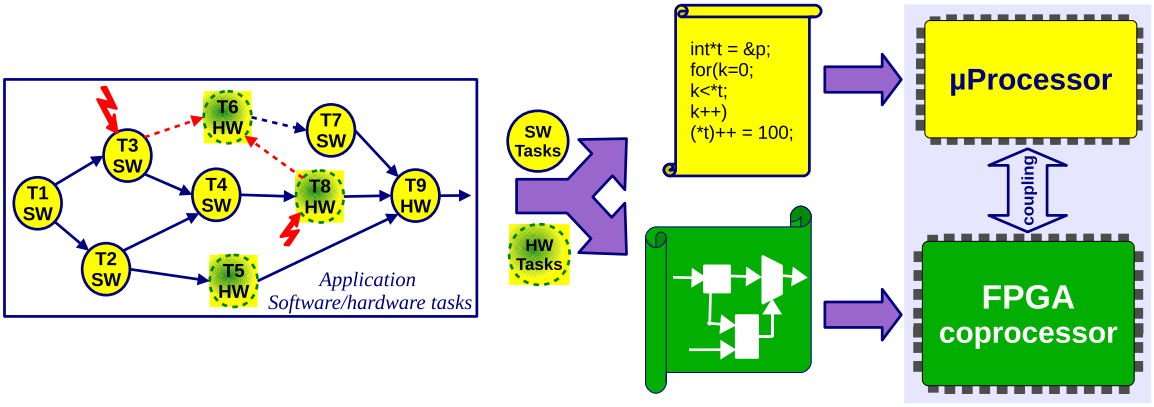
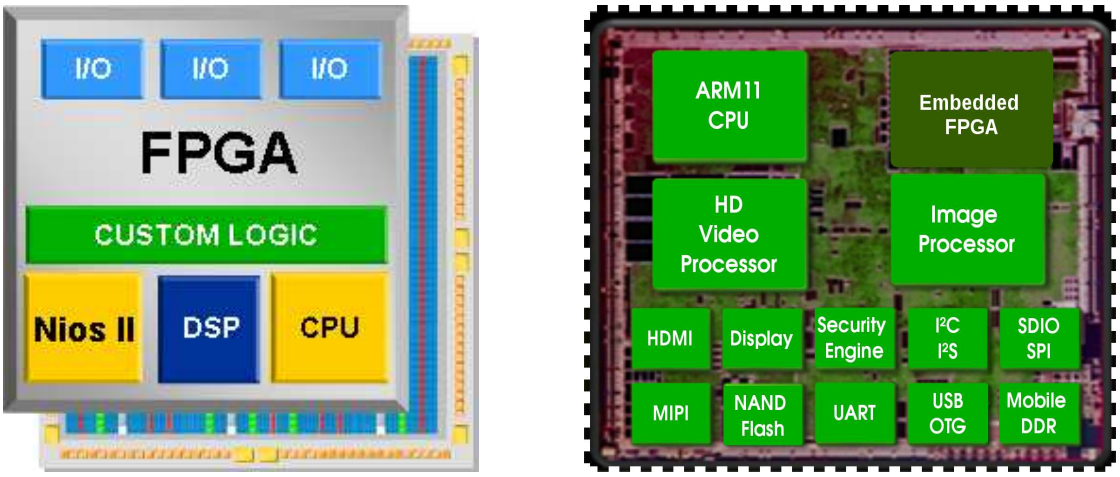


Figure 2.15: FPGA as co-processor



FPGA based SOC (courtesy of Altera) *Soc including an eFPGA*

Figure 2.16: FPGA based SOPC (left) and embedded FPGA (eFPGA)

3. ***Embedded FPGA*** *(figure 2.16, right)*

An FPGA fabric is embedded in a complex heterogeneous chip containing other elements such as GPP and/or DSP processors, memories, I/O and communication modules along with hardwired IP cores. Such FPGA coprocessor can be used as hardware extension for custom instructions. The reconfigurable fabric behaves like an extended datapath of the processor. Once again, the main advantage of integrating the reconfigurable fabric and the processor on the same silicon die is to reduce the communication cost as stated earlier. This

category could also be classified as Reconfigurable System-On-a-Chip (RSoC) which is a heterogeneous SoC containing a reconfigurable fabric.

2.5.10 Types of Reconfiguration

There are mainly two types of reconfiguration : static and dynamic.

Static (or compile-time) reconfiguration

Static reconfiguration is the most common way for implementing an application on reconfigurable logic using a classical design flow (HDL description or schematic capture, synthesis, place and route, download the bitstream in the FPGA) and conventional CAD[27] tools. Each application consists of one configuration. Once implemented, the application is supposed to run to completion without being interrupted. Static reconfiguration is mainly used when high performance, cost advantage (e.g. NRE costs) and upgradability are the main goals, and when there is no need to switch from one configuration to another in a relatively short amount of time (e.g. less than an hour).

Dynamic (or runtime) reconfiguration

Dynamic reconfiguration is the ability to reconfigure totally or partially the reconfigurable hardware while it is running. Hence, dynamic reconfiguration implies dynamic re-allocation of hardware blocks at run-time. It enables the concept of hardware virtualization where the physical hardware is smaller than the sum of the resources required to implement the whole application. Hardware virtualization is performed by time-sharing the same reconfigurable hardware (partially or totally) to different functions of the application which do not need to be run concurrently. This is known as temporal partitioning, and is still very challenging because of the lack of CAD tool support. In addition, swapping from one configuration to another brings additional challenges similar to context switching problem in traditional operating systems. In real-time constrained applications, reconfiguration time is the main bottleneck. It should be short enough to enable runtime hardware task switching. Despite the above-mentioned problems on implementing dynamically and partially reconfigurable systems, reconfigurable computing is referred to as the most suitable platform for DSP applications (e.g. jui Chou et al., 1993; Petersen, 1995; DeHon, 2000; Tessier and Burleson, 2001), and far justifies global research on OS for reconfigurable systems.

[27] Computer Aided Design

Single-context reconfiguration

In single-context reconfiguration, the reconfigurable hardware device has only one configuration downloaded on the device each time. In this case, any change on reconfigurable hardware functionality requires the complete reconfiguration of the entire chip, and therefore leads to high reconfiguration overhead. This feature makes this reconfiguration scheme more suitable for static reconfiguration where reconfiguration overhead is not a big concern.

Multi-context reconfiguration

In multi-context reconfiguration, many configurations are downloaded on the devices and are stored as planes of configuration information. Only one plane of configuration is active at each time, and the architecture could quickly switch from one configuration to another, just by activating one of the configuration planes. Obviously, switching from one configuration plane to another reduces configuration overhead as reconfiguration is not done sequentially, as during the download. Here, configuration switching is a matter of nanoseconds, where single-context needs milliseconds or more. Multi-context could be viewed as a kind of configurations prefetch approach which drastically reduces configuration overhead, but which requires more silicon area to build as many on-chip configuration planes as there are contexts.

Partial reconfiguration

Some reconfigurable hardware devices enable partial reconfiguration. Indeed, sometime, either only a part of a configuration requires some change or the incoming configuration is not big enough to fill the complete chip. By providing the ability for targetting a specific region of the chip while keeping other regions unaffected, partially reconfigurable hardware improves area efficiency and configuration overhead. The amount of reconfiguration data is smaller when it targets only a portion of the chip. One interesting feature of partially reconfigurable hardware is the smaller reconfigurable unit granularity. For example, in Xilinx FPGA Virtex II Pro, the smallest reconfigurable unit is a full column of the reconfigurable array. With its Virtex[28] family , Xilinx has been leading partially reconfigurable FPGAs market for years. However, Altera has entered this market in 2010 with its first 28nm FPGA chip, the Stratix V FPGA. The latter includes partial reconfiguration along with 28 Gbps transceivers and embedded hard IP blocks.

[28] Virtex II Pro, Virtex-4, Virtex-5, Virtex-6, Virtex-7, and other families (Artix-7 and Kintex-7).

In this thesis, the proposed scheduling strategies assume that the reconfigurable hardware device enables partial and runtime reconfiguration.

2.5.11 Configuration Hierarchy

An hierarchical model of reconfiguration is meaningful for heterogeneous architectures of reconfigurable devices presented above. This hierarchical approach aims to exploit the possibility of instantiating softcore IP blocks (e.g. processors) or hosting hardwired blocks (processors, memory, multipliers) on FPGAs. Indeed, an FPGA instantiating a softcore processor is capable of running both hardware tasks and software tasks. Hence, a task may be implemented on its hardware form (synthesized, placed and routed digital circuit) on the FPGA, or run on its software form (executable) on the processor instantiated on the FPGA.

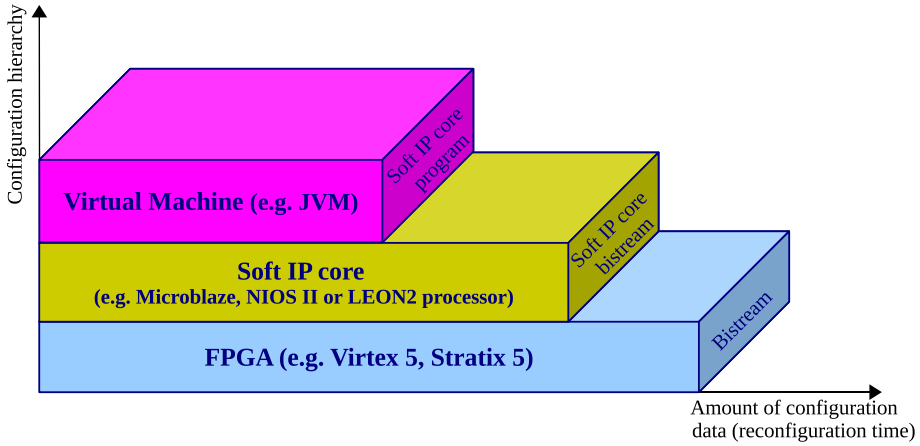


Figure 2.17: Configuration hierarchy model.

As stated throughout this thesis, reconfiguration overheads are among the main bottlenecks in real-time multitasking on reconfigurable hardware devices. However reconfiguration overheads could be reduced by using coarse-grained reconfigurable array operators (presented in section 2.6 below) as the basis of reconfigurable computing machines. Such operators or blocks, pre-built or pre-instantiated into the reconfigurable fabric could be more easily programmed at runtime with a minimal amount of configuration data. Reconfiguration overheads are thus reduced. This is especially true when pre-instantiated IP blocks are softcore processors. Configuration hierarchy denotes here the fact that the reconfigurable hardware device functionality could be changed at different hierarchy levels. Figure 2.17 maps an example where there are three levels

of reconfiguration : at bitstream level by reconfiguring the FPGA, at an intermediate level by instantiating another processor core, or at software level by changing the program executed by the instantiated softcore.

Related work

Through the hierarchical configuration concept, a few works (Schaumont and Verbauwhede, 2003; Nollet et al., 2006) have demonstrated the usefulness of this feature wish could improve flexibility and performance of heterogeneous platforms that include reconfigurable tiles. In addition, hierarchical configuration provides the possibility of running some hardware tasks in their software form at the cost of QoS[29], instead of rejecting them or keeping them in an increasing waiting queue. Furthermore, some applications use less hardware resources when implemented as a sequential machine on an instantiated softcore processor, while still meeting the necessary or a reasonable performance requirements.

Schaumont and Verbauwhede (2003) illustrate the configuration hierarchy in an FPGA through many figures. Figure 2.17 depicts a summary of these figures on a single graph. The Thumbpod system presented in Schaumont and Verbauwhede (2003) is a perfect application example which points out the assets of configuration hierarchy concept. It consists of an embedded Java virtual machine executed by a Leon2 softcore processor instantiated on a Virtex-II FPGA. The Leon2 soft IP core acts as program code (bitstream) for the Virtex-II FPGA and as hardware (microprocessor) for the user program (Java code). Configuration hierarchy provides more design and programming solutions ranging from harder (hardware design process) to easier one (software design process). Schaumont and Verbauwhede (2003) demonstrated that a hierarchical approach of configuration could reduce the reconfiguration time. Indeed, by increasing the configuration hierarchy the amount of configuration data to be processed is reduced. For example, reconfiguring a system designed according to figure 2.17 [30] at the highest level (by changing the user program executed by the Java virtual machine) is faster and easier than reconfiguring the FPGA fabric to implement a new softcore or a new design, since this operation does not need a new bitstream to be sent on the device.

More widely, by instantiating free available soft IP cores on FPGAs, designers take advantage of an easier and faster software design process (instead of hardware design process), leverage communication possibilities (e.g. buses) provided by microprocessors. This is also pointed out by

[29] Quality of Service

[30] figure 2.17 is deduced from the ThumbPod system, Schaumont and Verbauwhede (2003).

Nollet et al. (2006) through a literature review on exploiting hierarchical configuration to improve run-time task assignment on a Multi-Processor System-on-a-Chip (MPSoC). Unlike the above cited work on configuration hierarchy, most of similar work use configuration hierarchy at design time. Hence, when implementing an application, functionalities are mapped at design time in building blocks pre-placed within the reconfigurable logic. The purpose of using such compiler techniques to map functionalities is to design in a more efficient way by separating complex issues in a faster way by using a software design process, and in a more resource efficient way.

2.6 Coarse-grained Reconfigurable Arrays

2.6.1 Raison D'être

While presenting FPGAs above, some drawbacks of fine grain reconfigurable hardware devices have been pointed out and are listed below:

i). *Area utilization, power consumption and speed;* programmability at bit level implies using exclusively configurable logic to implement logic functions, which leads to a lesser area efficiency, a lower operation frequency and a higher power consumption.

ii). *Configuration overhead ;* the finer the configuration granularity, the more configuration data to process, and the longer the (re)configuration time.

iii). *Development ease ;* it is more difficult to target such devices with high level languages.

The aforementioned drawbacks can be reduced or overcome by using coarse-grained reconfigurable arrays operators as the basis of reconfigurable computing machines. In DSP applications, operations are performed on word-size data. Therefore, there is a need of word-size operators and datapaths. Implementing such functionalities using fine grain configurable resources requires to built them at bit-level. This leads to a tremendous use of logic blocks and programmable interconnections along with configuration data needed for the configuration. Configuration data are consequently bigger.

2.6.2 Presentation

A coarse grain reconfigurable architecture consists of hardwired word-size operators achieving nearly ASIC level features (high throughput, low power consumption, better area utilization, etc.) along with special purpose interconnections providing enough flexibility for targetting a given

class of applications. This approach is justified by the fact that for a given class of application or program to run, about 90% of the computation effort in terms of execution time or power consumption is due to about 10% of the code describing the application. Hence, great performance are achieved by designing high performance operators to cope with these repetitive and computationally-intensive parts of the application. Moreover, these parts are made of common functions (e.g. DSP) which are coarse enough to be identified even at high abstraction level, making coarse grain hardware more easy to target with hardware description languages than their fine grain counterparts.

Furthermore, the good performance of ASSPs is currently inspiring FPGA vendors. Indeed, FPGAs architectures are increasingly integrating more hardwired functionality (e.g. high speed transceivers, DSP blocks, etc.) in order to accommodate specific markets. This market-focused approach is seen as a move toward the ASSP path in terms of the targeted applications and the integration of cost-optimized peripherals that meet the needs of those targeted applications. These embedded coarse grain customized modules optimize one or many criteria and the FPGAs are classified in sub-families accordingly.

Summarizing, coarse grain reconfigurable hardware partly sacrifices its flexibility to provide a better performance for a given class of application while overcoming the drawbacks of fine grain reconfigurable architecture listed above. These drawbacks explain attention paid on coarser grain dynamically reconfigurable hardware to leverage the runtime dynamic reconfiguration feature and improve performance. Raw (Taylor et al., 2002), PipeRench (Goldstein et al., 2000), RaPiD (Ebeling et al., 1996), ADRES (Mei et al., 2003), PACT–XPP (Baumgarte et al., 2003) and Montium (Heysters et al., 2003) are a few examples of these architectures. They are classified as application domain-specific coarse grain reconfigurable systems. Today, commercial FPGAs manufacturers are following the trend by proposing different application domain-specific FPGAs at each new family release.

2.7 Platform-Based Design

2.7.1 Introduction

Choosing the right platform to implement an embedded application is getting more complicated. Indeed, as stated before, advances in technology bring new considerations. On one hand, those advances are the enabling technology for System-on-a-Chip (SoC) design approach. Hence, im-

plementation platforms cited above could be all integrated on a single silicon die and, thereby, the design architectural space is getting enlarged. On the other hand, novel architectures are proposed. They combine features of those implementation platforms in order to overcome their drawbacks, or to achieve a given trade-off.

2.7.2 Definition

Platform-Based design is a SoC design methodology, instead of being an implementation platform. According to the VSIA[31] working group, a Platform is "An integrated and managed set of common features, upon which a set of products or product family can be built. A platform is a virtual component (VC)". Hence, the VSIA working group defines Platform-based Design as "An integrated oriented design approach emphasizing systematic reuse, for developing complex products based upon platforms and compatible hardware and software virtual components (VCs), intended to reduce development risks, costs and time to market". For Martin (2003) "Platform-Based design is an organized method to reduce the time required and risk involved in designing and verifying a complex SoC, by heavy reuse of combinations of hardware and software IP. Rather than looking at IP reuse in a block by block manner, platform-based design aggregates a group of components into a reusable platform architecture". Vincentelli and Martin (2001) state that a platform is built to provide to designer libraries of hardware and software components, software drivers, hardware and software design environment, and references designs that are easily used as basis to rapidly develop many products within a reduced application space. In the light of those definitions, Platform-Based design approach is essentially an IP-reuse based design of SoC. It is leveraging the development of IP-reuse methodology. IP-reuse methodology aims to ease the design of complex ASICs by partitioning the design into smaller IP blocks with well-defined functionalities. That way, a validated block can be re-used in many designs instead of building the whole system from scratch. Design Reuse is the generalization of IP-reuse principle, and the main pillar of the Platform-Based design methodology. Here below are a few examples of existing

[31] Virtual Socket Interface Alliance; the VSI Alliance, founded in 1996, was an open, international organization comprised of representatives from all segments of the SoC industry. Its mission was to dramatically enhance the productivity of the SoC design community by providing leading edge commercial and technical solutions and insight into the development, integration, and reuse of IP.
Following 12 successful years of developing IP and electronics standards, in 2008 the VSI Alliance dissolved operations and transferred ongoing work of the VSI Alliance to other industry organizations.
Source : http://www.vsia.org

Platforms:

- Philips Nexperia (for digital TV applications);
- TI OMAP (from Texas Instruments, for mobile terminals applications);
- Nomadik (for mobile terminals application domain);
- ARM PrimeXsys (for processor-centric applications);
- Altera's SOPC (e.g. Excalibur ARM or NIOS reconfigurable platform);
- Xilinx Virtex Platform FPGA (e.g. Virtex II Pro, Virtex 4 and 5 reconfigurable platforms).

With these application oriented platforms, the design space is easier to explore. The designers directly derive their end product(s) from an existing platform(s); they use the complete hard IPs and software packaged (soft IPs, drivers, etc...) solutions provided by the platform and modify application software according to their needs. They have to customize hardware and software components and/or developed new ones to achieve the given purpose. They work at the application level by developing software and using available IP from existing libraries.

2.7.3 OS for Reconfigurable Platforms

The use of Platform-based Design approach in the so-called Reconfigurable SoC design rises the challenge of designing a real-time operating system (RTOS) for the platform. At a technology independent level and no matter which PEs[32] are targetted, a RTOS provides services such as scheduling different tasks on different targets under certain constraints, managing memory and communication media, providing special services for the reconfigurable hardware part of the platform if exists, etc. As pictured in figure 2.18, the set of services provided by the RTOS allows the designer to abstract the underlying platform details while implementing an application. The application to be implemented is divided into tasks to be scheduled on the PEs on the platform.

The OveRSoC methodology for DSE

There are two main approaches in designing the RTOS for RSoC. In the first approach, the RTOS is derived from an existing RTOS which is tailored with additional services that manage the reconfigurable part of the platform. The second approach is to design the RTOS from scratch, by introducing RTOS services exploration at system level. This thesis relies on the approach

[32] Processing Elements

presented in Miramond et al. (2009a), which corresponds to the second approach. Doing so, the RTOS is viewed as a flexible component which features will be explored and tuned as same as the application and the platform architecture. Furthermore, in the OveRSoC design methodology (depicted in figures 2.18 and 2.19 and presented in Miramond et al., 2009a), each PE likely to be on the platform is modeled as a RTOS which is connected to the rest of the system. Hence, the methodology (figure 2.19) automatically explores tasks distributions on a scalable multi-RTOS architecture with respect to application requirements and system constraints in order to assess and refine different services which allow the RTOS(s) to efficiently manage PEs of the platform. In this thesis, such a methodology is denoted as *OS-centric* or *OS-based*, as it relies on a third element, the *OS*, unlike other design methodologies. These latter rely only on the *application* and the *architecture*.

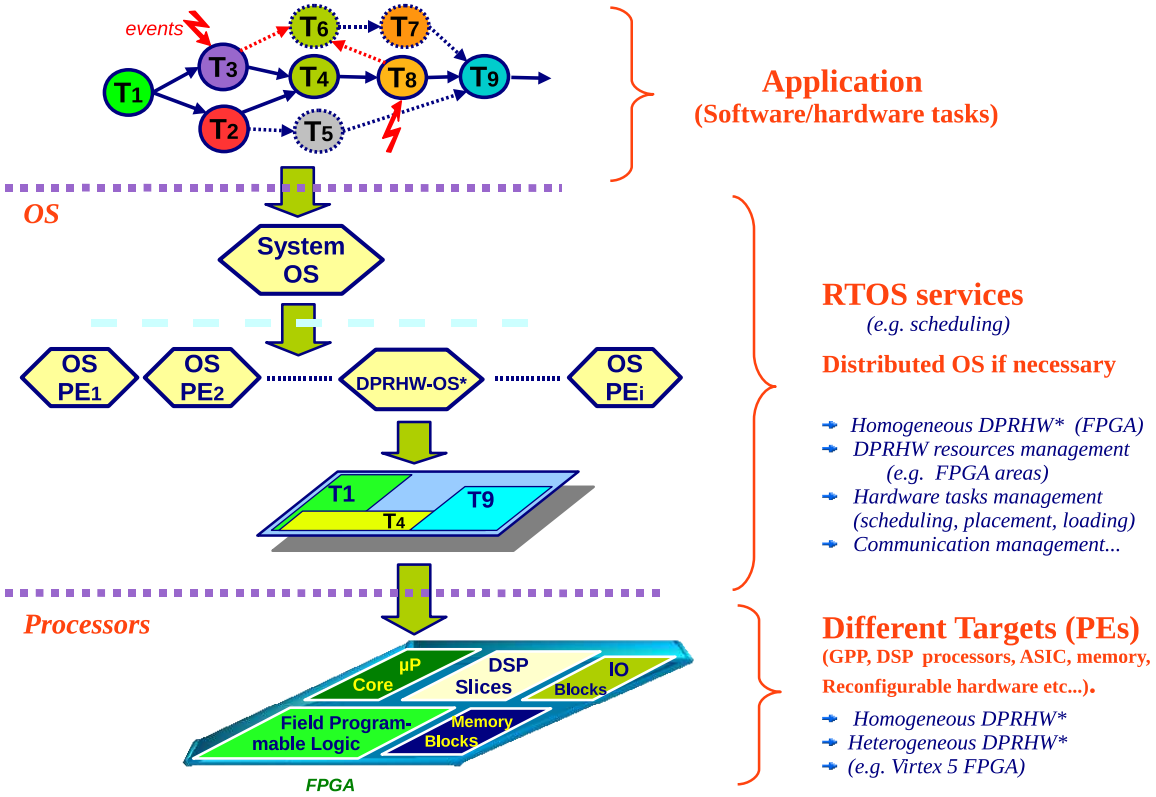


Figure 2.18: A view of the OveRSoC methodology with emphasis on DRA (dynamically reconfigurable architecture) management.

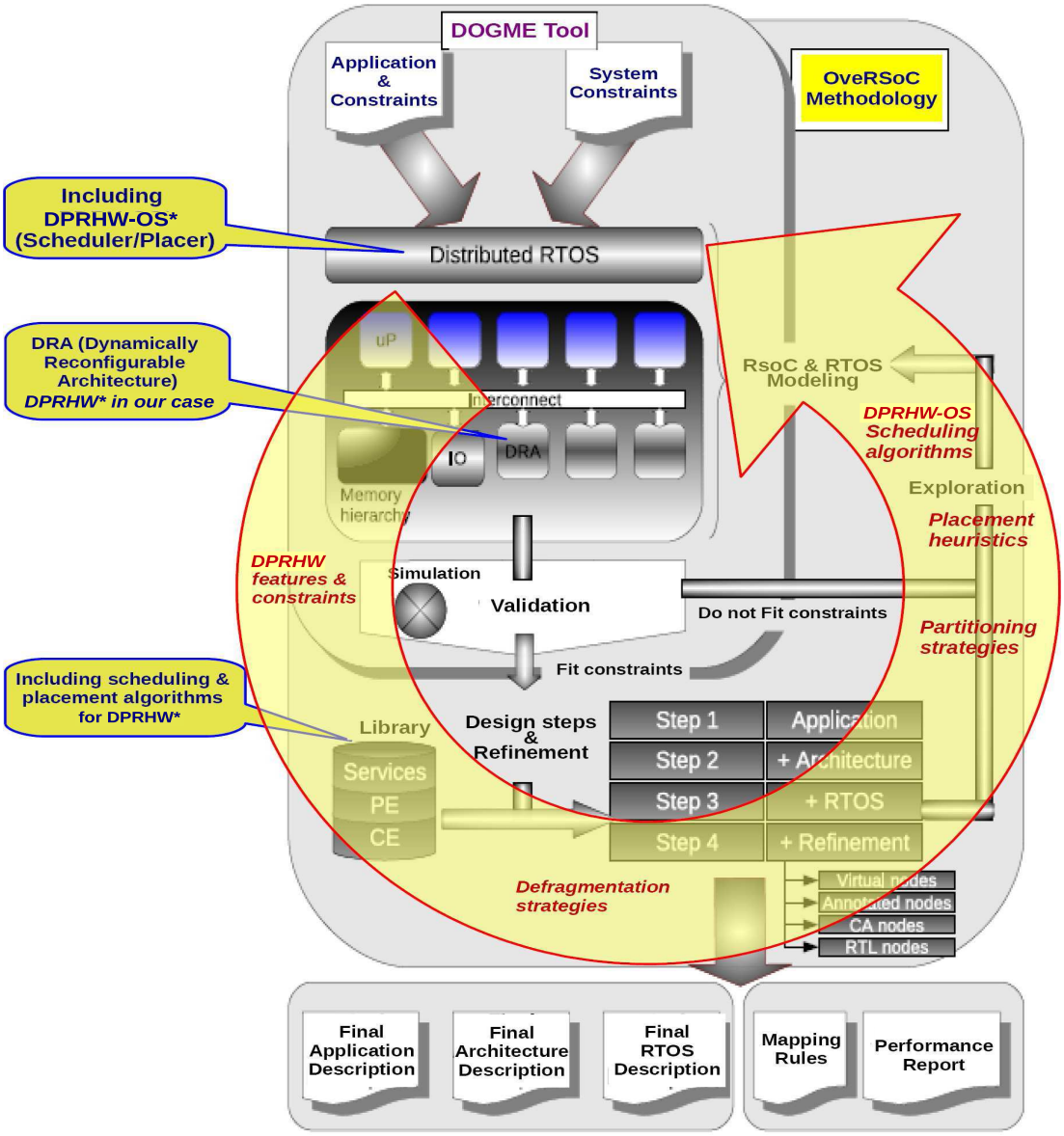


Figure 2.19: OS services exploration in OveRSoC design methodology (Miramond et al., 2009a) which maps the system level part of the generic design flow of SoC (see figure 1.4).

The concept of scalable multi-RTOS architecture is more detailed in figure 2.19 through the design flow of the OveRSoC methodology (Miramond et al., 2009a) and DOGME (Miramond et al., 2009b), its dedicated graphical user environment.

This thesis focused on the management of the dynamically reconfigurable hardware part of the platform and its corresponding DPRHW-OS[33], as shown in figure 2.18. Numerous services such as tasks creation, tasks scheduling, tasks placement, tasks migration, tasks preemption and quality of service assessment could be involved here. A task preemption arises when a task T_i running on a processor PE_i is stopped in order to assign PE_i to another task T_j of higher priority. T_i is then resumed later. A task migration arises when the task T_i preempted on processor PE_i is resumed later on a different processor PE_j.

As stated at the end of Chapter 1, the thesis deals with *scheduling* and *placement* services in DPRHW, as they are among OS services that need a different approach compared to conventional OS services for programmable processors. The two objectives remain as follows:

1. Proposing new on-line real-time scheduling and placement strategies on DPRHW (denoted as DRA[34] in figure 2.19), and that suit to on-line real-time context. The latter aim is achieved by finding a reasonable trade-off between scheduling and placement algorithms complexity and their performance in terms of chip utilization ratio, tasks rejection ratio, and runtime overhead.

2. Designing a set of scheduling and placement algorithms for DPRHW which could be used in the OveRSoC design methodology. Figure 2.19 shows the complete design flow of the methodology. Simulation is done at system level in order to refine the scalable architecture along with distributed OS, with respect to system and application constraints. In order to allow the methodology to perform a more accurate partitioning of the application, this work also aims at providing scheduling and placement algorithms for the DPRHW (denoted as DRA in figure 2.19) part of the architecture, along with the associated DPRHW models and metrics (utilization ratio, tasks rejection ratio, algorithms runtime overhead, partitioning and defragmentation strategies, etc.). These algorithms are of various complexity and runtime overhead, and help on finding the best trade-off depending on the parameter(s) to optimize. In addition, as the simulation environment for the OveRSoC methodology (Miramond et al., 2009b) is SystemC-based, the C++ language is used to design and implement the models and the algorithms in order to insure a full compatibility.

[33] Dynamically and Partially Reconfigurable Hardware - Operating System
[34] Dynamically Reconfigurable Architecture

2.8 Conclusion of the Chapter

This chapter presented the most common DSP implementation technologies and their derivatives. The chapter discussed strengths and weaknesses of each technology. Table 2.8 proposed by Adam (2002) shows a quick comparison. Thanks to experiences earned over the years in designing operating systems and compilers, GPP and MCU are the best in terms of time-to-market and flexibility. However, they are not suited neither to high-performance DSP applications with hard real-time constraints (figure 2.20) nor to power aware systems.

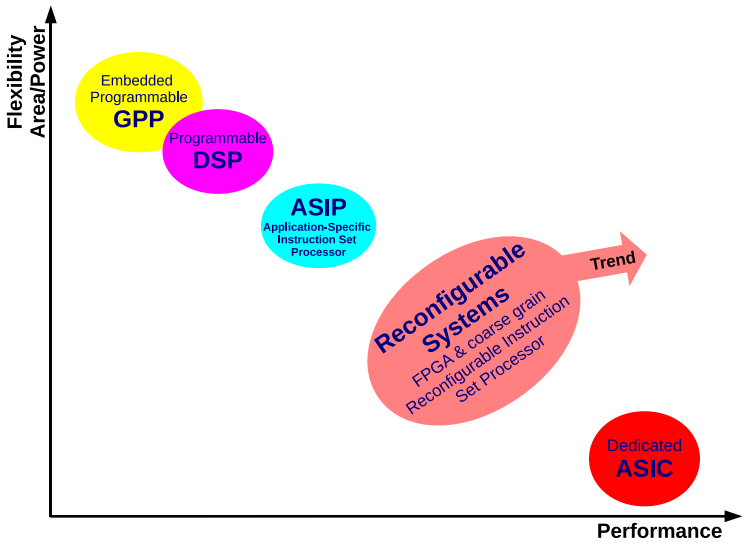


Figure 2.20: Flexibility vs Performance of implementation platforms.

	Performance	**Flexibility**	**Power**	**TTM**	**Price**	**Development ease**
GPP/MCU	fair	exc.	fair	exc.	exc.	good
DSP	exc.	exc.	exc.	exc.	good	exc.
FPGA	exc.	good	poor	good	poor	exc.
ASSP/ASIP	good	poor	exc.	fair	good	fair
ASIC	exc.	poor	good	poor	exc.	fair

exc. : *excellent*

Table 2.1: Comparative table of implementation platforms for DSP applications (Adam, 2002)

Programmable DSP processors are enhanced for DSP operations and low power consumption. Today, they are still the best solution for many DSP applications. However, the gap between them and GPP is narrowing, and programmable processors are reaching a plateau in their performance increase. High-performance and power-efficiency is the realm of ASICs. But the lack of flexibility, in addition to high Non-Recurring-Engineering (NRE) cost has limited the use of this technology to high-volume products. FPGA technology combines programmability and high-performance. Using FPGA-based reconfigurable computing systems or coarse-grained reconfigurable systems to achieve future embedded systems requirements is a promising solution. A modern trend is to manufacture application-oriented families of reconfigurable hardware devices. The increasing success of configurable processors also confirms this trend. But as it appears in table 2.8, the research has to improve the two main lacks of configurable processors : the Time-to-market and the development ease.

Since architectures are becoming more heterogeneous, spreading DSP applications within a system across an ever increasing architectural design space has complicated the design process. This chapter briefly introduced a new trend in embedded SoC design, the Platform-based Design approach, on which relies the OveRSoC methodology. The chapter has discussed how introducing scheduling and placement algorithms for DPRHW in a Platform-based Design methodology like OveRSoC, contributes to efficiently scour the design space in search for a good solution (e.g. a more accurate system partitioning and RTOS services refinement).
The next chapter will give a background on real-time scheduling and then will emphasize the online real-time scheduling for reconfigurable hardware devices through a wide literature review.

Chapter 3

Background and Related Work

3.1 Introduction

This chapter gives a background and presents related work on real-time scheduling. The discussion starts by reviewing the scheduling problem in general, and, subsequently, emphasizes the online scheduling of real-time hardware tasks on reconfigurable platforms. In the latter scheduling, an underlying placement problem for the reconfigurable hardware arises. The scheduling and placement problems are difficult to study separately, as they always interfere. Different paradigms of real-time scheduling are also presented, especially in online context. The review constantly emphasizes the similarities and differences between programmable processors scheduling and reconfigurable hardware scheduling. Hence, whenever possible, the review relies on the knowledge previously experienced in uniprocessor and multiprocessor scheduling. The reason for this being that scheduling hardware tasks on reconfigurable hardware devices shows some similarities with multiprocessor scheduling as many tasks can run concurrently. However, reconfigurable hardware scheduling is more complex to study because of the everchanging number and size of tasks that could concurrently fit on the reconfigurable fabric. Since the research domain is very wide, the second half of literature review is restricted to topics in various layers that are relevant to the *online real-time scheduling for dynamically and partially reconfigurable hardware devices*.

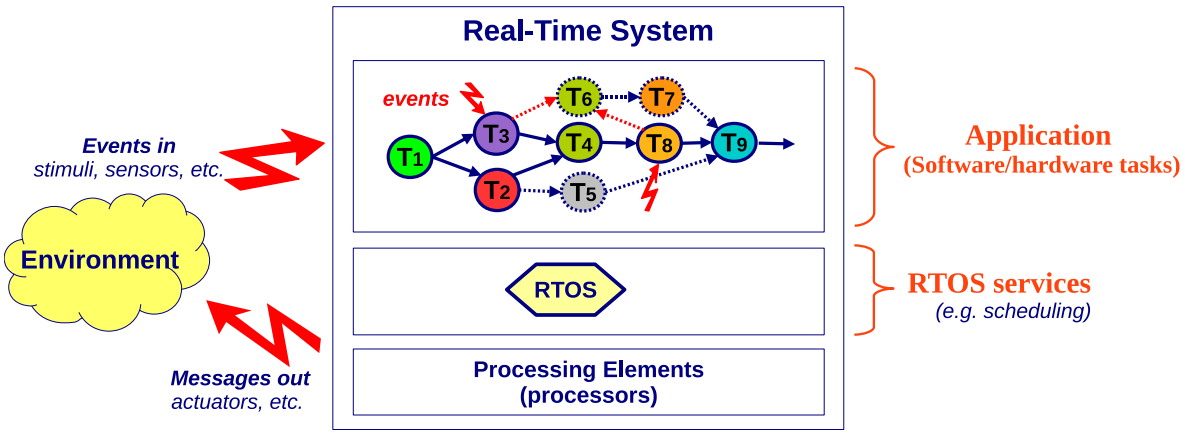


Figure 3.1: Model of a Real-Time system

3.2 Real-Time Systems

A real-time system is a system that is subject to a real-time constraint. Its primary performance is to perform critical operations within a set of user-defined critical time constraints (Locke, 1986). As pictured in figure 3.1, a real-time system is primarily reactive as it continuously reacts to stimuli coming from its external environment. In reaction to these external events, the results provided by a real-time system are only valid if they are delivered within a predetermined time frame. Hence, the correctness of a real-time system depends on two conditions (Stankovic, 1988):

1. *its logical correctness*; the system must compute correct outputs based on its inputs.
2. *its temporal accuracy*; output results must be delivered at the right time (a specified deadline). Failure to do so results in invalid results. In other words, a result arrived after its deadline is necessarily false or useless, and can lead to serious consequences in some cases.

Real-time computer systems are becoming ubiquitous in many applications such as control process system in factories, in aeronautics through on-board flight systems for aircraft and satellites, in IT[1] systems through multimedia communication, games development and virtual reality and even increasingly in finance through HPC[2] for real-time market data processing.

[1] information technology
[2] high performance computing

3.2.1 Hard *vs* Soft Real-Time

In a real-time system, time scale is application-dependent and can range from a few milliseconds (e.g. airbag system, automatic pilot system, etc.) to several hours (e.g. weather forecast). This means that real-time is not only a matter of average speed of the system, and that all timing constraints have to be met otherwise the system will fail. Depending on whether a system fails to meet its time constraint is vital or not, one can distinguish *hard real-time* and *soft real-time* systems:

Hard Real-Time does not tolerate any excess of time constraints, as such overflow can lead to critical situations with catastrophic consequences. For example, if an airplane autopilot system, a nuclear power station monitor, an airbag system or a medical systems such as heart pacemakers reacts beyond its strict time limits, it can seriously endanger the safety of human lives . In order to avoid such dangerous situations, the designer of a hard real-time system should be able to prove that the time limits will never be exceeded whatever the situation (even in the worst case situation, regardless of system load). Hence, designing a hard real-time system which interacts with its environment assumes that all possible behaviours of the system are predictable and time-bounded. A single task that misses its deadline constitutes failure of the whole system. Hard real-time systems are submitted to acceptability tests (e.g. threshold test, feasibility analysis, admission control, etc.) for their validation.

Soft Real-Time is less restrictive; it tolerates deadline overruns at the cost of the quality, as far as they remain within certain limits beyond which the system becomes useless. Obviously, unlike hard real-time, missing the deadline affects the QoS (e.g. telephone, video conferencing, network games, etc.) without leading to catastrophic consequences. Sometime, the system allows some exceptional time limits excesses to be compensated during the next execution (e.g. video frames). What is important in most of the cases is the average number of deadline overruns that needs to be below a given threshold in order to insure a given QoS in a given situation. Most of the time, a statistical study of the system behaviour is enough to design a trustworthy soft real-time system.

3.2.2 Requirements for Real-Time Computer Systems

As time constraints are essential, designing a real-time system assumes that each element of the system itself is real-time constrained. Hence, services response time and algorithms runtime

overheads are necessary time-bounded. It also assumes that the time flows in the system and can be measured. A real-time system consists on one hand of resource consumers (tasks, application) and on the other hand of resources providers (processors, memories, etc.). As detailed in figure 2.18 page 59 and summarized in figure 3.1, different parts of a real-time (reconfigurable) system may be seen as layers, with a central part denoted as *OS* or *RTOS* sandwiched between the application layer and the resources layer. The OS which hosts the scheduler acts as an interface between the application and the Processing Elements.

3.3 Real-Time Scheduling

3.3.1 Introduction

The scheduling problems are present in many systems ranging from factory systems through process control, to embedded systems. It appears in any domain where there is a need to organize the allocation of finite resources to a given application which consists of a sequence of tasks. It is then necessary to coordinate the use of the resources in order to run the application to completion and as efficiently as possible. This efficiency means optimizing one or many criteria. Such criteria could be to minimize the schedule length (or makespan, defined in section3.3.4), to maximize the resources utilization ratio, to maximize the number of accepted tasks (e.g. tasks which meet their deadline), etc.

In general, a scheduling problem is described by a triplet $\{\alpha, \beta, \gamma\}$ where α represents the machine or processor environment, β the application to process on the machine along with its time constraints, and γ the objective function to be optimized. Put it simple, the application consists of n tasks that have to be processed on m processors while optimizing the objective function γ. Many scheduling problems have been shown to be NP-complete optimization problems, and many scheduling heuristics of lesser complexity have been proposed.

3.3.2 Real-Time Tasks

A task is a set of instructions to be executed on a processor. It provides a given service to the application. In order to perform an efficient scheduling without violating any time constraints, the scheduler has to know timing characteristics of the tasks. These characteristics are of different importance depending on the scheduling policy used. In general, a timing parameter is given as a positive integer, multiple of the smallest indivisible time unit (denoted as *tick* in time-aware

systems). The most common parameters of a real-time tasks T_i are (see figure 3.2, page 70):

- a_i : *arrival time* (sometimes denoted as *release time* or *request time* r_i) is the time when task T_i is created and is ready to be run on the processor.
- e_i : *execution time* (or *computation time* or *processing time*) is the duration needed by a task to run to completion on a given processor. Therefore, execution time is processor-dependent. In most of hard real-time scheduling problems, execution time e_i is assumed to be equal to the WCET[3] on the considered processor. The WCET of a task T_i is the maximum time that the task will require to run to completion on the considered processor.
- P_i : *period* for a periodic task.
- *deadline* is the time at which the task must have been completed. There are 2 types of deadline :
  - the *relative deadline* D_i if the deadline is relative to the release time of the task instance.
  - the *absolute deadline* $d_i = a_i + D_i$.
- l_i : *laxity* (or *slack time*) of a task T_i is the difference between its relative deadline and its execution time and is defined as

$$l_i = D_i - e_i \tag{3.1}$$

  Once released at time a_i, a task T_i cannot wait more than its laxity l_i before starting, otherwise it will not meet its deadline.
- p_i : *priority* if priorities are used. Scheduling algorithms based on the priority of the tasks are denoted as *priority-driven* or *priority-based*.
- s_i : *start time* corresponds to the date which task T_i will start its execution on the processor. s_i is assigned by the scheduler.
- f_i : *finishing time* or *completion time* c_i corresponds to the date which task T_i will end. Obviously it depends on its starting time.
- rt_i : *response time* is the difference between the arrival time and the finishing time, and is given by

$$rt_i = f_i - a_i$$

[3] Worst Case Execution Time

- u_{Ti} : the *utilization ratio* of task T_i is the ratio of its execution time e_i to its period or its minimal inter-arrival time P_i and is given by equation 3.2

$$u_{Ti} = \frac{e_i}{P_i} \tag{3.2}$$

  u_{Ti} reflects the chunk of time occupied by task T_i when executed on a single processor.

A *job* is an instance of a task. Hence the j^{th} instance of task T_i is denoted either as $T_{i,j}$ or as $J_{i,j}$. In this thesis, a job $T_{i,j}$ or $J_{i,j}$ may sometime be referred to as task T_i or J_i if there are any possible ambiguities.

Periodic Real-Time Tasks

A task T_i is denoted as periodic if instances $T_{i,j}$ of the tasks are released periodically and with a fixed periodicity P_i as shown in figure 3.2. In real-time computing systems, an application is sometime made of computation tasks that have to be performed periodically. Therefore, the model of periodic task proposed by Liu and Layland (1973) based on a WCET is widely used (e.g. in Danne, 2006). If the first release date of a periodic task is unknown (resp. to be known), the task is denoted as *non concrete* (resp. *concrete*). In addition, periodic tasks that are concrete are said *synchronous* if their first release date is identical (e.g. T_i and T_j in figure 3.2), otherwise they are *asynchronous*. Systems with non concrete tasks (i.e with some unknown first release time) show an *event-driven* behaviour while concrete tasks systems are *time-driven*. If equation 3.3 (where P_i is the period of task T_i) is verified, the real-time system is said *harmonic*.

$$\forall i, j \in \mathbf{N}, \quad P_i > P_j => \exists n \in \mathbf{N} : \quad P_i = n \cdot P_j \tag{3.3}$$

If $D_i = P_i$ (resp. $Di < P_i$), then T_i is said to have implicit deadline (resp. constrained deadline). A system exclusively made of tasks with implicit deadlines (resp. with constrained deadlines) is denoted as an implicit-deadline system (resp. constrained-deadline system). In an arbitrary-deadline system some tasks could have their deadline greater than their period ($Di > P_i$). In periodic tasks systems, implicit-deadline system model is the most widely used (e.g. Danne, 2006).

Let Γ_n be a set of n tasks $[T_1, T_2, ..., T_n]$ and $[P_1, P_2, ..., P_n]$ the corresponding periods. The *hyper-period* Hp of the periodic tasks set is the least common multiplier of the periods of the tasks set, as defined in the following equation 3.4.

$$\exists Hp : \forall T_i \in \Gamma_n,\ i \in \mathbf{N},\ Hp \bmod (P_{i=1..n}) = 0 => \min(Hp) \text{ is the hyper-period.} \tag{3.4}$$

The pattern of jobs activation is repeated identically in time intervals equal to the hyper-period.

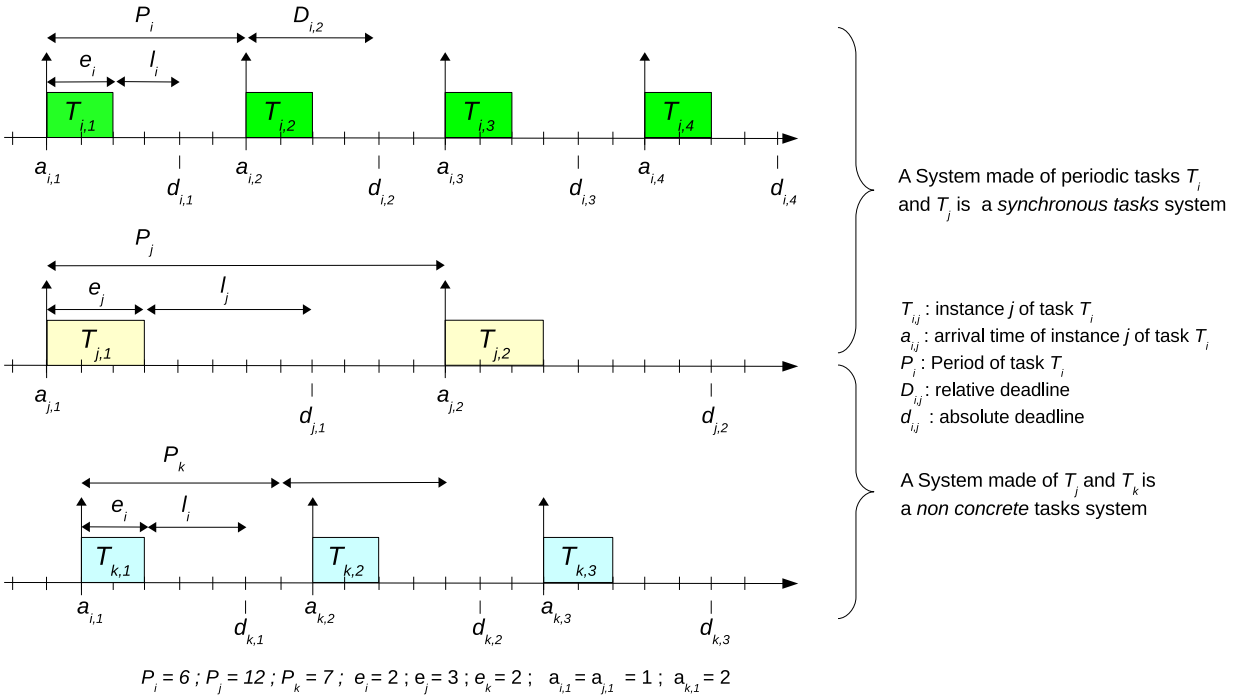


Figure 3.2: Different periodic real-time task according to their release time

As a feasible schedule found and validated for the first hyper-period is used indefinitely, a system consisted only of periodic tasks is therefore much more easy to schedule.

Aperiodic and Sporadic Real-Time Tasks

A practical real-time system hardly consists of periodic tasks exclusively. Indeed, in a real life scenario some non-periodic tasks could appear at any time in addition to periodic tasks. External events such as sensor activation or target detection are such tasks that pop up and have to be urgently handled by the system. A task is *aperiodic* when instances of the task are released randomly in time (figure 3.3, down). However, as shown in figure 3.3 (top), a task is denoted as *sporadic* if there is a minimum inter-release time P_i between instances of the tasks.

As summarized in figure 3.4, a periodic task is a special case for a sporadic task where inter-arrival time remains constant, and a sporadic task itself is a special case for an aperiodic task.

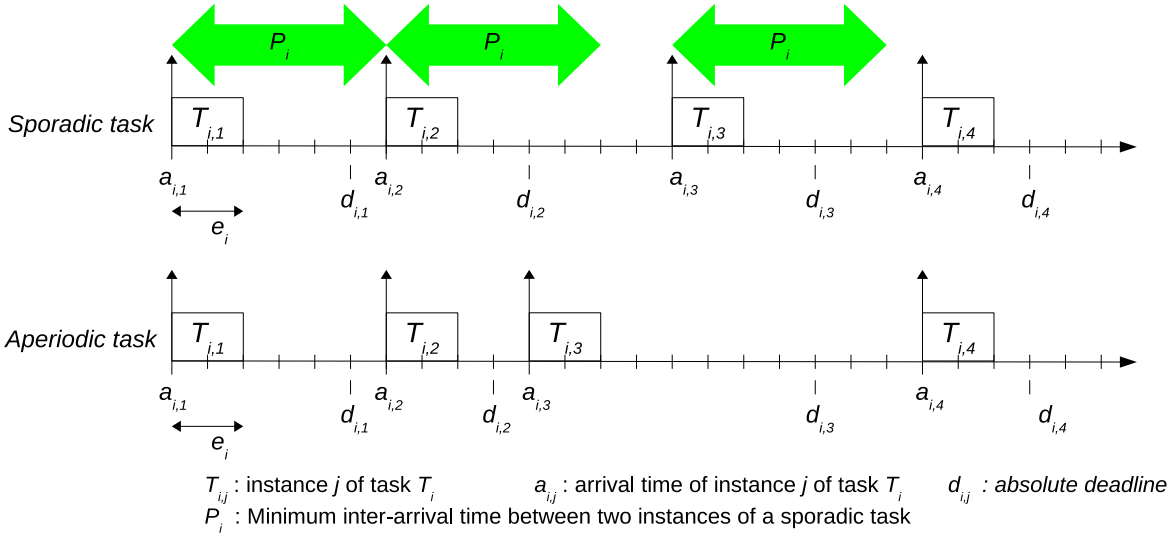


Figure 3.3: Aperiodic task and sporadic task.

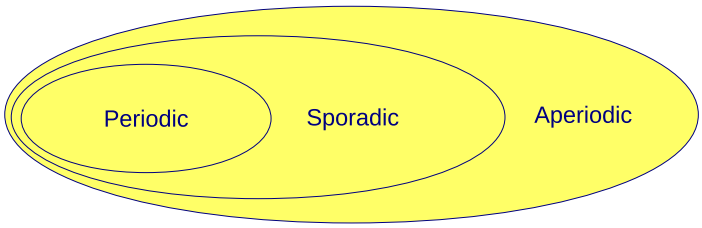


Figure 3.4: Periodic, sporadic and aperiodic tasks

3.3.3 Different Scheduling Problems

As previously said, a scheduling problem generally consists of processing a sequence of n jobs $J_{1..n}$ on m machine $M_{1..m}$ with respect to a given objective function to optimize. Consequently, a scheduling problem is expressed according to the jobs, the processor(s) and the objective function.

Uniprocessor *vs* Multiprocessor Scheduling

The distinction between uniprocessor and multiprocessor scheduling, depends on whether there is one or many processors available to process the jobs in the system. In both cases, each processor can process only one job at a time, and each job can only be processed by one processor at a time. Uniprocessor scheduling is the simplest scheduling approach and has been widely studied. Scheduling is much more difficult in a multiprocessor system. The number of processors are

sometimes denoted as m. Processors may or may not be identical.

Soft *vs* hard Real-Time Scheduling

Hard and soft real-time systems has been previously presented in section 3.2.1 page 66 above. Depending on which kind of real-time system is considered, there's *hard real-time scheduling* and *soft real-time scheduling*.

Static *vs* Dynamic Scheduling

In a static scheduling, tasks parameters (e.g. priority) are assigned beforehand and remain unchanged during the system life time. In contrast, dynamic scheduling assumes that scheduling relies on tasks parameters that vary at runtime. For example in *priority-driven* scheduling, the scheduling is denoted as *static* if a priority assigned to a task on release, cannot be changed throughout its life (e.g. RM[4] scheduling algorithm). However, if a dynamic scheduler is used, then the priority assignment of the task can be changed at runtime (e.g. priority inversion, EDF[5] algorithm, etc.).

When applied to reconfigurable hardware scheduling and placement, Ahmadinia et al. (2004) defined a static scheduling and placement as being when the same scheduling and placement rules apply to every single arriving task and the entire reconfiguration area is available for the placement of any task. Their algorithm is dynamic in the sense that their scheduling and placement heuristics are adjusted at runtime depending on some parameters of arriving tasks. In addition their FPGA is divided in clusters, and a task could be placed in a given cluster depending on its completion time.

Fixed *vs* Dynamic Priority Scheduling

A fixed priority scheduling algorithms can be either at *tasks level* or at *jobs level*. In the first case, the priority of each task is assigned at design time and remains unchanged (e.g. Rate Monotonic). In the second case, at different jobs of the same task could be assigned different priorities. However each job's priority remains unchanged during its execution. Example, EDF scheduling algorithm.

Unlike *fixed priority* algorithms, the priority of any job could be changed at any time in *dynamic priority* algorithms. Example, LLF - least laxity first scheduling algorithm.

[4] rate monotonic

[5] earliest deadline first

Preemptive *vs* Nonpreemptive Scheduling

Preemption arises when a task is interrupted before its completion (and resumed later) in order to assign its processor to another task of higher priority. Preemption only happens when every processor is busy. The preempted task is resumed later. In a nonpreemptive tasks system, once a task starts its execution it runs to completion.

Nowadays, preemption is a mechanism widely used in operating systems. However, preemption makes scheduling much more difficult as the scheduler has more operations to perform. Indeed, resuming a task that has been suspended earlier needs a context switching. The latter consists of tasks context saving and restoring necessary to insure continuity in scheduling. Preemptive algorithms can only be applied on tasks that are preemptable[6]. In addition, a task migration is necessary if the task is to be resumed on a different processor. *Task preemptions*, *context switches*, *task migrations*, and the *scheduling* itself are system overheads as they take time in addition to the application tasks execution time. Furthermore, these system operations are very challenging and time consuming on heterogeneous platforms[7].

In general, a cost is associated to each of these system operations while coping with preemptive scheduling. However, most of the studies assume that preemption cost could be neglected as far as tasks execution times are far higher than context switching runtime overheads. Obviously, preemptive scheduling is much more difficult, compared to nonpreemptive scheduling, and even worst in heterogeneous multiprocessor system especially when a suspended task is resumed on a different kind of processor.

Throughout this thesis, it is assumed that system overheads are included in tasks execution time, which is the WCET.

Precedence Constraints

In general, a system could be represented as a functional block diagram. The functional blocks are identified and characterized tasks. They are sometime dependent as some consume (consumers, successors) outputs computed by others (producers, predecessors), imposing a network

[6] some tasks are preemptable at any time, others at precise times, and others simply too difficult to preempt properly, because the context saving and restoring is difficult to insure.

[7] e.g. hardware and software tasks running on heterogeneous processors including programmable, dedicated and reconfigurable. Hardware tasks preemption is a very challenging research concern in reconfigurable computing. So are software-to-software and hardware-to-software tasks migrations on heterogeneous processors.

of interdependencies. Precedence constraints require that one or more jobs have to be completed before another job is allowed to begin its processing. Precedence constraints clearly appear in dataflow graph (DFG) modeling commonly used in signal processing.

Given two jobs J_i and J_j , if a J_i has to be completed before J_j starts, then J_j is the successor of J_i while J_i the predecessor of J_j. Constraints are referred to as *chains* if each job has at most one predecessor and at most one successor. If each job has at most one successor, the constraints are referred to as an *in-tree* or *join type*. They are *fork type* or an *outtree* if each job has at most one predecessor.

This thesis mostly considers independent tasks. However, it also considers the jobs arrive over time online tasks model (described later in section 3.4.2) that may implicitly model some precedence.

3.3.4 Objective Functions

To measure the quality of scheduling, different indicators are at disposal. Depending on the aim, these indicators appear as *objective functions*. Hence, the scheduling problems usually consist of optimizing (by minimizing or maximizing) a given *objective function*. Objective functions could be either the sum of these factors or their average values over all the jobs. The most common of them are :

(i). **Makespan** or **length of the schedule;** the makespan C_{max} or mk is the finishing time of the last job completed in the system. The objective function is generally a function of the completion times of each job, consequently the makespan which is to be minimized. A minimum makespan leads to a high processor utilization ratio, which is desirable for optimal use of resources. The makespan could be also defined as the length of the schedule or the maximum completion (or finishing) time, not to be mistaken for the *sum of the completion times* of all jobs defined as $\sum_{i=1}^{n} c_i$ for a set of n completed jobs.

(ii). **Response time** or **total flow time;** it reflects the time that flows from the release of the job in the system to its completion time.

(iii). **Waiting time;** is the time that flows from the release of a job in the system to its starting time.

(iv). **Lateness and Tardiness;** let's J_i being a job with an absolute deadline d_i and completed on time c_i; its *lateness* L_i and its *tardiness* δ_i are defined respectively in equation 3.5 and

equation 3.6 as follows:

$$L_i = c_i - d_i \tag{3.5}$$

$$\delta_i = \max \{ L_i, 0\}; \text{ with } \delta_i \geq 0 \tag{3.6}$$

where L_i is positive when J_i completes late and negative when J_i completes before its deadline. In the latter case, J_i can be associated with some *value* v_i which is obtained only if the task is completed prior to its deadline. Therefore, the higher the values, the smaller the makespan, and the better the scheduling.

However, the lateness (hence tardiness) is very meaningful in soft real-time systems where the system could still insure a given QoS even when some tasks miss their deadline. One example is when a deadline violation of a job J_i can be tolerated as long as it does not affect or delay the release of the next job J_{i+1} of the same task.

The QoS can be expressed by appointing a tardiness bound to every task in the system. That is, each job J_i of task T_i must be completed at most δ_i time units after its deadline in the worst case. The tardiness could be also the same for all the tasks in the system. Viewed as an objective function, the tardiness of a job could be expressed with respect to a given algorithm.

In preemptive tasks systems, minimizing the number of times a job is preempted could be among objective functions.

3.3.5 Offline Scheduling

Depending on whether scheduling decisions are taken offline or online, scheduling problems can be distinguished into two categories: *offline* and *online*.

A scheduling is denoted as *offline* if the flow of the program is known beforehand. This assumes that parameters (e.g. release time, execution time, deadline, size, etc.) of all the tasks are known in advance, allowing the scheduling to plan resources allocation at design time. Most scheduling problems are NP-complete. High performance scheduling algorithms and heuristics are used to optimize objective functions (e.g. makespan, resources utilization, waiting time, response time, etc.). However, one could afford to run such computationally intensive algorithms as analysis and computations are done offline [8].

One example of an offline scheduling arises when an application is made of a set of well known

[8] e.g. on a host PC or a high performance computer

periodic tasks. The scheduling simply consists of finding a feasible schedule over the *hyper-period*. Consequently, at each time interval equal to the hyper-period, the same feasible schedule is applied over the hyper-period. A feasible schedule is statically stored in a scheduling table that indicates, on each clock cycle which job is to be run.

Offline scheduling is not in the scope of this thesis. However, it can be used to provide an optimal solution for competitive analysis of its online counterpart. An interesting survey on offline scheduling could be found in Graham et al. (1979).

3.4 Online Scheduling

3.4.1 Introduction

Online scheduling is used when the flow of the program is either totally or partially unknown beforehand. In other words, parameters of tasks to be scheduled are partially or totally unknown prior to their release time. Hence, tasks are scheduled as they arrive, without a priori knowledge of future tasks. The scheduler relies on incomplete information to assign jobs to resources. Such a scenario is even much more realistic, compared to the offline scenario. Indeed, even if offline scheduling algorithms provide optimal solutions, they are less suitable for most real-time systems. Generally, as such systems are mainly reactive, there is a lack of information on future tasks (e.g. their release time) and sometimes even on tasks not yet completed (e.g. their execution time). This lack of information is the main difference between offline scheduling and online scheduling. It explains why unlike online scheduling, offline scheduling leads to an optimal scheduling. In general, online scheduling is priority-driven scheduling. The highest priority job is chosen by the scheduler among the ready jobs in order to be executed on the available resource. The scheduler is invoked each time a job is released, or a running job is stopped, or finished, or even at regular time interval for quantum-based scheduling.

3.4.2 Different Online Paradigms

As stated above, online scheduling relies on incomplete information about input instances to take scheduling decisions. Depending on the type of information lacking and on the way how new information becomes known, different online paradigms are possible. Some are widely presented in Sgall (1998) and here below are a few that drew the attention to the scope of this work :

(i). **Scheduling jobs one by one**;

also denoted as *jobs arrive over list*, there is no job release time in this online paradigm. Jobs appears as an ordered list and are presented to the decision maker (scheduling algorithm) as is, one by one. However, as soon as a job is presented, all its characteristics are known, including its running time. Therefore, the job could be assigned to a time slot with an immediate or a delayed start. When dealing with the currently presented job in the list, the assignment of the previous job cannot be changed anymore, leading to a situation where previously scheduled jobs cannot be rescheduled. To summarize, jobs are absolutely scheduled as ordered in the list, but are not necessarily executed in that order. The online feature is the fact that parameters of future tasks are unknown before they are presented. LS[9] algorithm is based on this paradigm.

(ii). **Clairvoyant Scheduling**;

unlike the previous paradigm, release time is meaningful in clairvoyant scheduling algorithms. The algorithm only becomes aware of a job when it arrives. Hence, it knows the running time of a job (along with its other parameters, e.g. its size in the case of hardware tasks) as soon as it arrives. *Clairvoyant scheduling* is also denoted as *jobs arrive over time* online paradigm, as jobs become available to the algorithm over time as soon as they are released.

(iii). **Non-Clairvoyant Scheduling**;

in *non-clairvoyant scheduling*, the online algorithm only becomes aware of a job when it arrives. In addition, unlike clairvoyant scheduling, the processing time of a running job remains unknown until the job finishes. Indeed, the algorithm only becomes aware of the end of the job at its completion. Consequently, non-clairvoyant scheduling is sometimes described as *blind scheduling*. This paradigm is also referred to as *unknown processing times* as the latter is its main online feature. Jobs are released over time according to their arrival time or their precedence constraint. However, if there are other characteristics of the task (e.g. size of a hardware task), they are known when the job arrives. Sometimes in this paradigm, it is assumed that at every moment the number of pending jobs is known. Non-clairvoyant scheduling has been widely studied (e.g. Rajeev Motwani and Torng, 1994; Edmonds, 2000). Its online features rely on the lack of knowledge on future tasks (e.g arrival time and running time).

[9] *list scheduling* algorithm, paragraph 3.5.5 on page 82

(iv). **Interval Scheduling**;

this paradigm assumes that each job has to be executed in a precisely given time interval, otherwise it will be rejected. Consequently, the number of accepted tasks is much more meaningful here than the makespan or the total completion time.

This thesis mainly considers the clairvoyant scheduling as the online paradigm. Hardware jobs become available over time according to their release time. Other parameters of each job such as execution time, deadline, size (width and height) etc. are available as soon as the job arrives.

3.4.3 Performance Analysis

Optimal solutions are not expected in online problems because of lack of priori knowledge of the problems. However, it is important to measure the performance of a scheduling with respect to a given objective function. To achieve this, one way is to perform an *average-case analysis* by considering average performance of the scheduling algorithm or heuristic over all its possible inputs. The latter analysis may even be performed on classes of inputs in order to classify the algorithm behaviour with respect to the range of tasks parameters. Indeed, even in online scheduling, the range value of some tasks parameters could be partially or totally known beforehand (e.g. number of jobs, range of execution times, range of sizes, minimum inter-tasks arrival time, etc.). In addition, as the algorithm or the heuristic learns about jobs piece by piece over time, another way to improve its behaviour is to make some probabilistic assumptions on future jobs, based on current and past jobs. However, an *average-case analysis* cannot trap worst case scenario where the algorithm performs very poorly.

Competitive Analysis

A failure in an online system is avoidable by measuring the minimum guarantee that an online scheduler could provide, even in its worst case behaviour. This is achievable by using *competitive analysis*, first introduced by Sleator and Tarjan (1985). Based on a *worst-case analysis*, competitive analysis of an algorithm consists of comparing the worst solution[10] provided by the algorithm with the optimal solution. For example in an online scenario, the worst solution of the online algorithm $\tilde{A}_{on}$ will be compared with the optimal solution $\tilde{A}_{opt}$ obtained in the corresponding offline case. Consequently, competitive analysis is a more intuitive way of assessing the performance of an online algorithm.

[10] with respect to a given objective function

Competitive Ratio

The performance of an online algorithm is measured by its *competitive ratio*. As scheduling aims at optimizing a given objective function, the latter could be a cost to minimize (e.g. makespan) or a benefit to maximize (e.g. utilization ratio, number of accepted tasks, etc.). Let $\gamma(\tilde{A}_{on}\ ,\ \Gamma_n)$ be the objective function resulting from the online algorithm $\tilde{A}_{on}$ applied on Γ_n, a set of n tasks T_1, T_2,..., T_n. Let $\jmath_n$ be an input jobs sequence J_1, J_2,..., J_n of tasks set Γ_n. Algorithm $\tilde{A}_{on}$ is said $c - competitive$ if for any input instance $\jmath_n$, the objective function γ produced by $\tilde{A}_{on}$ on Γ_n) is at least c times better than that obtained with the optimal algorithm $\tilde{A}_{opt}$, as shown in equation 3.7.

$$\forall \jmath_n\ ,\ c = \begin{cases} sup\frac{\gamma(\tilde{A}_{on},\jmath_n)}{\gamma(\tilde{A}_{opt},\jmath_n)} \text{ if } \gamma \text{ is a cost to minimize} \\ \\ sup\frac{\gamma(\tilde{A}_{opt},\jmath_n)}{\gamma(\tilde{A}_{on},\jmath_n)} \text{ if } \gamma \text{ is a benefit to maximize} \end{cases} \tag{3.7}$$

As *competitive ratio* is based on a *worst case analysis*, c is chosen as the supremum of ratio in equation 3.7 over all possible inputs instances $\jmath_n$. In such a case, c is also denoted as the *upper bound* of the competitive ratio of the algorithm $\tilde{A}_{on}$.

Equation 3.7 also shows that c is at least equal to 1 ($c \geq 1$). The closer c to 1, the better the online algorithm $\tilde{A}_{on}$.

The *competitive ratio* c of a given algorithm $\tilde{A}_{on}$ expresses the fact that there is no any other algorithm capable of performing over c times better than $\tilde{A}_{on}$, even if considered the offline case where the entire problem instance is known in advance. Let's notice that the latter case enables optimal solutions. Therefore the competitive ratio clearly reflects the advantage of knowing the entire problem beforehand.

The *upper bound* reflects the worst competitive ratio that could achieve the algorithm on any input instance. Hence, to obtain a more accurate upper bound, it could be necessary to carefully build some input instances in order to make the online algorithm performs the worst possible. One way of improving the algorithm design is then to reduce its upper bound.

In contrast, the lower bound (of the competitive ratio) of an online algorithm indicates that for an online problem, the competitive ratio of that algorithm cannot be less than its lower bound. In other words, the lower bound of an algorithm reflects the best performance achievable by the algorithm on an online problem.

3.4.4 Schedulability Analysis

Feasible Schedule: The basic challenge of *scheduling* is to assign a starting time and a processor to any single task or instance of task in the system. For example in a microprocessor based system, the scheduler decides in which order and at which date the single-task processor will be assigned to different tasks of the application. This aim is not always achievable as processing resources are finite and probably cannot run all the tasks in the system to completion while meeting their deadline, no matter which scheduling policy is used. Hence, a schedule is denoted as *feasible* on a given computing resources (e.g. one or many processors) if it allows all the tasks of the application to be successfully scheduled on the resources without violating their time constraints (e.g. deadlines). A set of tasks is *schedulable* on a real-time system if there is a *feasible schedule*. In the latter case, the system is *underloaded*, otherwise *overloaded*. The *schedulability* of the system highly depends on :

(i). the constraints of the application (tasks).

(ii). the available resources (number, size and type of PEs, etc...).

(iii). the scheduling policy used. It consists of algorithms and heuristics.

3.5 RT 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. Liu and Layland (1973) provide a foundation reference on the subject. Many optimal uniprocessor scheduling algorithms have been proposed along with their schedulability analysis. Here below are some scheduling algorithms. The end of this section discusses how microprocessor scheduling could be applied to reconfigurable hardware scheduling.

3.5.1 Rate Monotonic (RM)

RM scheduling was first introduced by Liu and Layland (1973). It is a preemptive and fixed priority scheduling for periodic and independent tasks systems. Priorities are assigned to tasks in a way that the shorter the task period, the higher the priority. Liu and Layland (1973) have proven RM optimal among all fixed priority scheduling when applied on the above-mentioned preemptive, periodic and independent tasks system. They also provided a sufficient but not necessary scheduling test of RM algorithm on a set n tasks as defined in expression 3.8 below,

where c_i is the processing time of task T_i, P_i its period and $\sum_{i=1}^{n} \frac{c_i}{P_i}$ the processor utilization ratio.

$$\sum_{i=1}^{n} \frac{c_i}{P_i} \leq n \cdot (2^{\frac{1}{n}} - 1) \tag{3.8}$$

However, RM is not optimal for non-preemptive scheduling.

The RM scheduling is not used in this thesis as periodic tasks systems are not used.

3.5.2 Deadline Monotonic (DM)

DM is a scheduling algorithm that gives the highest priority to the task with the least relative deadline D_i. In general, DM is optimal for systems that consist of preemptive synchronous independent tasks whose relative deadline is less to their period ($D_i \leq P_i$). DM could be used with periodic, aperiodic and sporadic tasks systems.

A sufficient schedulability test for DM scheduling of n tasks $\Gamma_n = [T_1, T_2, ..., T_n]$ using DM algorithm is expressed as :

$$\forall i : 1 \leq i \leq n : c_i + \sum_{j=1}^{i-1} \frac{D_i}{P_i} \cdot c_j \leq D_i \tag{3.9}$$

3.5.3 Earliest Deadline First (EDF)

Also first introduced by Liu and Layland (1973), EDF could be applied to various tasks models (preemptive, non-preemptive, periodic, non-periodic, etc.). EDF scheduling is a dynamic assignment 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 all kind of tasks models, including preemptive scheduling of periodic and independent tasks sets on a microprocessor. The necessary and sufficient scheduling condition is given by :

$$\sum_{i=1}^{n} \frac{c_i}{P_i} \leq 1 \tag{3.10}$$

with $P_i = D_i$, where P_i (resp. D_i) is the period (resp. the relative deadline) of task T_i.

$\sum_{i=1}^{n} \frac{c_i}{P_i}$ is the *time utilization factor* of the processor and reflects the fraction of time the processor is really running a task. Obviously, this fraction may not exceed 100%.

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.

3.5.4 Least Laxity First (LLF)

LLF is a priority based scheduling where the task with the smallest laxity is scheduled first. This algorithm is quite close to EDF algorithm, with a similar necessary and sufficient scheduling condition $\sum_{i=1}^{n} \frac{c_i}{P_i} \leq 1$ for a set of n preemptive tasks, and for $P_i = D_i$. However, in LLF algorithm, many tasks are likely to have the same laxity (which means same priority), leading to a situation where many preemptions are performed in a short time frame, which is not desirable. The latter situation is less problematical in a multiprocessor system.

3.5.5 List Scheduling (LS)

Ronald L. Graham was the first to prove competitiveness on an online scheduling algorithm in 1966, through the so-called *list scheduling* algorithm. LS[11] is the most commonly used online scheduling approach thanks to its simplicity. Indeed, it relies on the *scheduling jobs one by one* online paradigm described on page 76. In the LS algorithm, tasks to be processed are listed in order. When a resource is available, the first task listed is elected, computed and removed from the list. A task is available if heading the list, and free of precedence constraints if any. As there is no prior knowledge of task in the list, LS is a suitable and simple model for online problems.

3.5.6 Uniprocessor Scheduling Model for Reconfigurable Hardware

In this scheduling model the entire reconfigurable hardware device is viewed as a single processor. Therefore the device is assigned to only one task (or set of tasks taken as a whole and active at nearly the same time) at a time. One example is shown in figure 3.5 where tasks T_1, T_2, T_3 and T_4 are sequentially executed on the reconfigurable hardware, on a time-sharing basis. In order to achieve this, the tasks or jobs $(\tau_1, \tau_2, ..., \tau_{10})$ are time-partitioned into four tasks T_1, T_2, T_3 and T_4. The reconfigurable hardware device is not space partitioned in this model.
Hardware tasks scheduling on a time sharing model of the reconfigurable hardware device is the simplest, thanks to its similarities with the well-studied scheduling on uniprocessor systems. Hence, the scheduling problem is reduced to a uniprocessor scheduling. The same above-mentioned scheduling algorithms (EDF, LLF, RM, etc.) along with their results could be directly applied.

However the intrinsic parallelism provided by hardware implementation is not exploited here. Moreover, this approach leads to a high *internal fragmentation* (presented later in section 3.9.1, page 124) and a low reconfigurable hardware device *utilization ratio* as the whole device is assigned

[11] list scheduling

to a single task at a time, no matter how small the size of the task compared to the size of the reconfigurable array.

Compound tasks

A compound task is a task composed of many tasks or subtasks. In order to exploit the parallelism provided by reconfigurable hardware and improve the reconfigurable devive area utilization ratio, one solution is to concurrently run many tasks instead of one at a time as previously described. This could be achieved by forming subsets of tasks that could be treated as one global task, as long as they share some timing and/or geometric similarities.
Let Γ_n be a set of n jobs $\tau_1, \tau_2, ..., \tau_n$ to be scheduled on the DRHW. As depicted in figure 3.5 and 3.7 there are two ways of forming subsets of jobs that could be executed concurrently on the partially reconfigurable hardware device.

1. ***Time sharing (space overlapping) compound tasks***
   As pictured in figure 3.5, one way of overcoming the aforementioned drawbacks is to partition Γ_n into m subsets of compound tasks $T_1, T_2, ..., T_m$ in such a way that :

   (i). $m \leq n$

   (ii). $\forall \tau_i \in T_j$,

   - $a_i \in [t_a - \epsilon, t_a]$ => all arrival times a_i in the subset T_j are within a given interval; $t_a = max(a_i)$ is the arrival time that is assigned to the tasks subset T_j.
   - $t_d = min(d_i)$ => the smallest deadline d_i in the tasks subset T_j is assigned as the deadline of T_j, which is t_d.

   where a_i (resp. d_i) is the arrival time (resp. deadline) of job τ_i, and t_a (resp. t_d) the arrival time (resp. deadline) of the tasks subset T_j.

   (iii). $\forall i \in [1, n], \forall j \in [1, m]$,

   (a). $w_j \cdot h_j \leq W \cdot H$ => the reconfigurable hardware must be able to fit the compound task T_j.

   (b). if $\tau_i \cap T_j \neq 0$ then $w_i \cdot h_i \leq w_j \cdot h_j$ => compound tasks T_j are bigger.

   where $w_i \cdot h_i$ (resp. $w_j \cdot h_j$) is the size of task τ_i (resp. task T_j) and $W \cdot H$ the total size of the reconfigurable hardware.

(iv). $\tau_1 \cup \tau_2 \cup ... \cup \tau_n = T_1 \cup T_2 \cup ... \cup T_m$

(v). $\tau_1 \cap \tau_2 \cap ... \cap \tau_n = T_1 \cap T_2 \cap ... \cap T_m = 0$

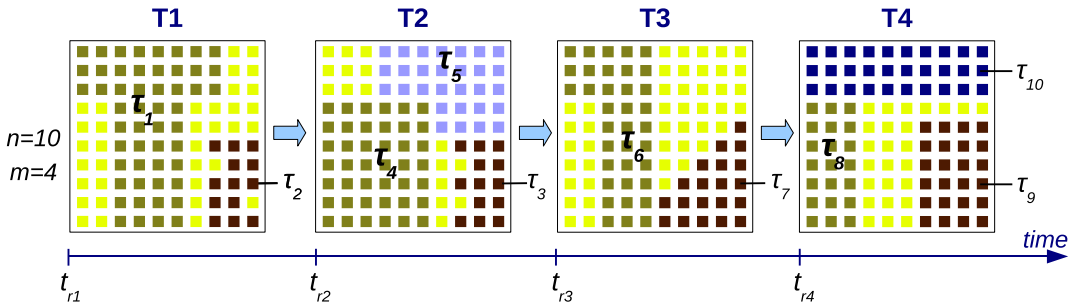


Figure 3.5: Uniprocessor model for reconfigurable hardware devices with *time sharing* compound tasks.

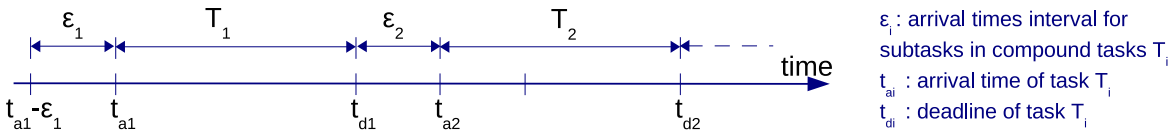


Figure 3.6: Compound tasks timing characteristics

Condition (1i) expresses the fact that each subset T_j is a compound task that contains one or many tasks τ_i, making therefore T_j bigger (as also expressed by condition (1(iii)b)). This is seen in figure 3.5 where each task T_j contains 2 or 3 jobs τ_i and therefore improves the utilization ratio of the reconfigurable array. Furthermore, the number of reconfiguration is bounded by m.

Conditions in (1ii) express the fact that timing similarities of jobs τ_i are the main factors that govern tasks grouping. Hence all the jobs τ_i of a given compounded task T_j must be active at nearly the same time in order to be managed as a single task T_j. This is also illustrated in figure 3.6 where ϵ_i is the time interval within which all the subtasks of compound task T_i arrive. Conditions (1iv) and (1v) emphasize that compound tasks T_1, T_2,...,T_m differ each other. As T_1, T_2,...,T_m do not overlap in time, they can be sequentially executed on the reconfigurable array on a time sharing basis, as illustrated in figure 3.5. These timing constraints could lead to situations where the reconfigurable array is poorly occupied. The reconfigurable array is fully reconfigured between two tasks execution.

However, on one hand partitioning a set of n tasks in m distinctive subsets of tasks as described above is not easy to perform. On the other hand, if most of the subtasks τ_i of task T_j are mostly idle during the running time of T_j (as they are probably not strictly all active at the same time), the real utilization ratio of the reconfigurable array will remain poor.

One main advantage provided by this time sharing uniprocessor model is that the fragmentation problem is eschewed. Each compound task T_j is designed as a whole and its bitstream file is used to fully reconfigure the device on demand. Each compound task T_j can contain as many jobs τ_i as possible as long as the total amount of resources does not exceed the amount of resources available on the reconfigurable hardware device, so a high device utilization ratio is achievable. As this design approach is free of rectangular partition constraints that govern the *module-based design methodology* described in Chapter 2 section 2.5.8, there is no internal fragmentation. Figure 3.5 shows that a job τ_i could be of any shape without preventing the placement of other jobs τ_k that overlap with its surrounding rectangle (see section 3.9.1, page 124 on internal and intra-task fragmentations).

Partial reconfiguration is of no use in this model. Full reconfiguration results in higher reconfiguration time overheads and complex tasks switching. However the number of full reconfiguration (which is m as shown in figure 3.5, and bounded by n in the worst case, if preemption is not allowed) is lower compared to the following uniprocessor model.

2. ***Space sharing (time overlapping) compound tasks***

Figure 3.7 depicts another variant of uniprocessor model. Once again, Let Γ_n be a set of n jobs $\tau_1, \tau_2, ..., \tau_n$. Γ_n is partitioned into m subsets of compound tasks $T'_1, T'_2, ..., T'_m$ in such a way that :

(i). $m \geq n$ if at most one job is released at a time.

(ii). $T'_1 \cap T'_2 \cap ... \cap T'_m \neq 0$

(iii). $\forall i \in [1, n], \forall j \in [1, m]$,

(a). $w_j \cdot h_j \leq W \cdot H$ => the reconfigurable hardware must be able to fit the compound task T'_j.

(b). if $\tau_i \cap T'_j \neq 0$ then $w_i \cdot h_i \leq w_j \cdot h_j$ => compound tasks T'_j are bigger.

where $w_i \cdot h_i$ (resp. $w_j \cdot h_j$) is the size of task τ_i (resp. task T'_j) and $W \cdot H$ the total size of the reconfigurable hardware.

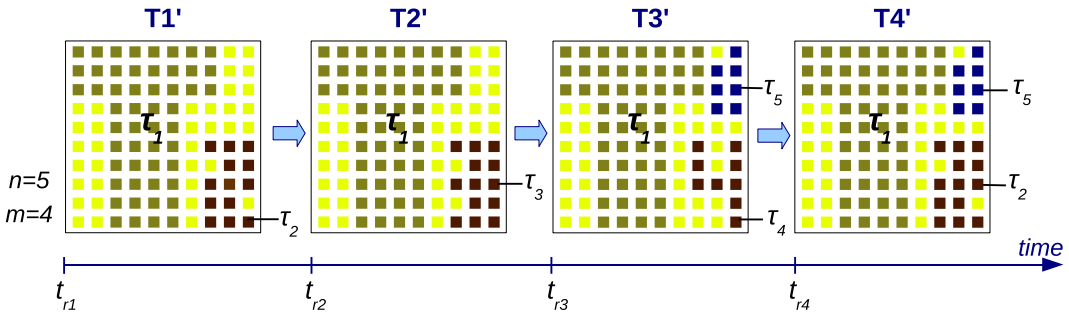


Figure 3.7: Uniprocessor model for reconfigurable hardware devices with *space sharing* compound tasks

As each compound task T_j contains one or many jobs τ_i, T_j is therefore bigger (as also expressed by condition (2(iii)b)) and improves the device area utilization ratio.

Condition (2i) expresses the fact that there are at least as many compound tasks (T_1, T_2,...,T_m) as jobs (τ_1, τ_2,..., τ_n). The main reason is that two or more compound tasks can share some common jobs τ_i. This is also shown by condition (2ii) which means that compound tasks are very likely to be partly similar. For example in figure 3.7, each of the four compound tasks T'_1, T'_2, T'_3 and T'_4 contains job τ_1, making them slightly similar. However, they are considered as 4 distinctive tasks that need a complete reconfiguration of the device at every task switching. The idea is to treat any change on the list of currently running tasks as task switching. Hence, as τ_2 is replaced by τ_3 and then τ_3 by τ_4, these replacements correspond to tasks switching from T'_1 to T'_2 and then from T'_2 to T'_3 as pictured. One advantage in fully reconfiguring the device at every task switching is that some tasks can be relocated in order to make bigger contiguous free space, hence improving the device utilization ratio.

The main drawback of this model is that the device is often and fully reconfigured, which is time consuming. As shown in figure 3.7 the similarities between T'_1 and T'_2 are not taken into account during the reconfiguration. Indeed the device is fully reconfigured everytime a new job τ_i is removed and/or added on the device (4 full reconfigurations for 5 jobs, compared to 4 full reconfigurations for 10 jobs in figure 3.5). For example, for n jobs $\tau_1, \tau_2, ..., \tau_n$, if we assume that at most one job arrives at a time, then the device needs to be fully reconfigured at least n times throughout the execution of n jobs (or the m compound tasks). In any case, each task ending and each task beginning (and even each

task preemption, if allowed) is likely to induce the full reconfiguration of the device. Thus, the number of reconfigurations drastically increases with the number of tasks and each new task corresponds to a reconfiguration or bitstream file to be downloaded on the device. As shown in Chapter 2 section 2.5.8 while describing the FPGA modular design flow, each compound task T'_j is designed and synthesized off-line and stored as a bitstream file that fully reconfigures the reconfigurable array at the appropriate time. Thus, all possible scenarios of tasks that are likely to run concurrently on the device are prepared beforehand. As the number of reconfigurations is huge (at least equal to m, with $m \geq n$), the resulting bitstream files storage memory is tremendous.

However, if the reconfigurable device enables runtime partial reconfiguration, the amount of configuration data and therefore the reconfiguration time overhead could be reduced. Depending on the similarities between two tasks T'_j and T'_{j+1} to be sequentially executed on the device, the partial bitstream can be orders of magnitude smaller than the full bitstream and can be quickly loaded. In fact for a set of n tasks $\tau_{i=1..n}$, one only needs to generate and store beforehand n partial bitstreams, one per task. In such a case, the switch from task T'_j to task T'_{j+1} is performed by reconfiguring only the part of the reconfigurable hardware device that needs to be changed. If applied on the example in figure 3.7, throughout the execution of T'_1, T'_2, T'_3 and then T'_4, only the area of the device where the new task τ_i is to be hosted is reconfigured on-the-fly while the remaining part occupied by τ_1 is not reconfigured and remains unchanged.

Partial reconfiguration capability fully exploits tasks redundancies that could appear in some compound tasks. Either a module-based or difference-based design methodology presented earlier in Chapter 2 section 2.5.8 can be used here to implement partial reconfiguration. However the module-based methodology induces more fragmentation as tasks are rectangular shape constrained.

3.6 RT Scheduling for Multiprocessor Systems

The problem of scheduling tasks on many processors cannot be seen as a simple extension of the uniprocessor scheduling. Before focusing on reconfigurable hardware devices scheduling in the next section, the current section gives a short taxonomy of multiprocessor systems scheduling along with most significant results. The reason is that in a certain way reconfigurable hardware scheduling shares some similarities with multiprocessor scheduling. These similarities will be pointed up as

they appear.

An increasing number of real-time applications require many processors to achieve their performance goals. In uniprocessor systems, the performance is improved by increasing the speed and unfortunately the power consumption. However, a multiprocessor system achieves a better performance and is of a lower cost compared to its fastest uniprocessor counterpart. Consequently, on one hand advances in technology are enabling Multiprocessor Systems-On-a-Chip (MPSoC), and on the other hand in embedded systems many processors could be distributed in different parts of the system, each dedicated to a specific task. As stated in the first chapter, examples could be found in modern car control systems which are usually divided into fifty-plus processor-controlled subsystems (e.g. ABS-Anti-lock Braking System, airbags systems, fuel injection system, etc.).

3.6.1 Multiprocessor Scheduling Problem

In a multiprocessor system there are $m \geq 2$ processors available in the system for computations. These processors are denoted as $M_1, M_2, ..., M_m$. In a multiprocessor scheduling problem, a list of n jobs J with nonnegative processing times have to be scheduled on m processors with the aim of completing all the jobs without violating their time constraints and while minimizing a given objective function (e.g. makespan). Additional constraints are :

- the type of tasks (periodic, aperiodic or sporadic)
- the preemption (allowed or not)
- task migration
- the precedence constraints (independent tasks or not)
- task parallelism; if allowed, different jobs of a task can be executed concurrently on different processors.
- processor M_i capability of executing a single tasks at a time (e.g. sequential processors) or more (e.g. reconfigurable hardware devices).

By default, this thesis assumes that tasks are aperiodic, non-preemptive and independent.

3.6.2 Multiprocessor Platforms

Unlike uniprocessor systems it is possible to execute many jobs of the application concurrently. There are mainly two types of multiprocessor platforms:

1. ***Homogeneous Processor System:*** Processors in the platform are *identical* in terms of speed. In such a system, any task could be executed on any processor with an execution time which is therefore processor-independent as speed of processors are strictly similar. *Identical processors* approach is the simplest multiprocessor model but would lead to a huge internal fragmentation (as discussed later below in section 3.9.1) if applied to hardware tasks scheduling on partially reconfigurable hardware devices.

2. ***Heterogeneous Processor System :*** Processors in the platform are of different types and speeds. Tasks processing time is therefore processor-dependent. On one hand if any task could be run on any processor with different execution times the platform is called *uniform processors platform*. Hence, each processor M_i is characterized by its speed s_i. On the other hand when some tasks can only be run by some processors, the heterogeneous platform is denoted as *independent processors platform*. Consequently, each task is characterized by the processors capable of running the task, along with the execution time required by each processor.

   As stated in the introduction of the thesis, embedded platforms are turning heterogeneous MPSoC. As this thesis considers SoC with reconfigurable parts, *heterogeneous processors system* can perfectly be such a platform in some special cases where the reconfigurable fabric contains a fixed number of m slots or clusters. The tasks could be therefore as heterogeneous as the platform and consist of software and/or hardware versions.

3.6.3 Partitioned *vs* Nonpartitioned Scheduling Strategies

Traditionally, there are two classes of scheduling algorithms for multiprocessor platforms: partitioned scheduling and global scheduling. Partitioned scheduling assigns each task to a particular processor. The task is scheduled on the local processor using a uniprocessor strategy. Task migration is not allowed. Global or nonpartitioned scheduling shares all tasks across all processors. At every moment the m highest-priority tasks ready for execution are scheduled on m processors. Thanks to task migration, global scheduling is likely to achieve higher schedulability than partitioned scheduling. However, task migration and preemption increases runtime overhead. This section presents the *partitioned scheduling* and the *non-partitioned scheduling*. Throughout this section, it is assumed that Γ_n is a set of n tasks to schedule on m processors $M_1, M_2, ..., M_m$.

1. ***Partitioned or Local Scheduling***

   Let Γ_n be a set of n jobs $\tau_1, \tau_2, ..., \tau_n$. Γ_n is partitioned into m subsets of tasks $T_1, T_2, ..., T_m$

in such a way that :

(i). $\forall T_j, j \in [1, m]$, T_j constains at least one task τ_i.

(ii). $T_1 \cap T_2 \cap ... \cap T_m = 0$

(iii). $\Gamma_n = \tau_1 \cup \tau_2 \cup ... \cup \tau_n = T_1 \cup T_2 \cup ... \cup T_m$

Each subset T_j is scheduled on the same processor M_j using a monoprocessor scheduling approach. Task migration is not allowed. Hence, all instances of a task is scheduled on the same processor by monoprocessor scheduling algorithms such as EDF, LLF or RM.

Indeed, as monoprocessor scheduling has been widely studied in terms of schedulability analysis, the same results are easily transposed to each tasks subset T_j and its associated processor M_j. Moreover, a monoprocessor approach avoids task migration and the resulting context switching which is very challenging and time consuming in multiprocessor systems.

2. ***Non-partitioned or Global Scheduling***

Unlike the local approach presented above, any task or any job in the set Γ_n could be executed on any processor. Furthermore a task could be preempted on one processor and resumed later on another processor (which is task migration, if preemption is allowed). The strategy is global as there is any partitioning done beforehand, and at every decision time, all the m processors of the platform are assigned to the m higher priority tasks.

3.6.4 Multiprocessor Scheduling Model for Reconfigurable Hardware

The previous section has presented the partitioning strategies both from multiprocessor platform and task system perspectives. This section will discuss how these strategies could be adapted to hardware tasks scheduling on reconfigurable hardware devices.

Partitioned Scheduling

Partitioned scheduling could be transposed to dynamically reconfigurable hardware device. Indeed, this could be done by partitioning the reconfigurable fabric in m equal slots as pictured in figure 3.8 (a), and by partitioning the tasks system Γ_n accordingly. If we assume that any slot can fit any hardware task (e.g. if the thinnest slot can fit the widest task, or if the width of slots is equal to the width of tasks), then a monoprocessor assignment scheme could be applied to each slot. The set of n tasks Γ_n is partitioned into m subsets following the three conditions (1i), (1ii) and (1iii) as described above, m being the number of slots. Each slot is therefore assigned to a

subset of tasks as shown in 3.8 (a) where arriving tasks are divided in three subsets.
In such an equal-slot partitioned strategy, the number of slots m corresponds to the number of processors, and the whole reconfigurable array is seen as the aforementioned *homogeneous multiprocessor system*. With the assumption that each slot is big enough to fit any task of its subset, monoprocessor scheduling algorithms along with theories could be directly and successfully applied to executed each subset on its corresponding slot.

This assignment scheme could be extended to the case where the above-mentioned assumption is not verified. Indeed, as illustrated in figure 3.8 (b) the reconfigurable array could be unequally partitioned with a fixed number of slots m. If m is fixed, the model is much closer to the *heterogeneous multiprocessor system* presented earlier. In addition if some tasks don't fit the slots, then the reconfigurable array may be viewed as an *independent processors platform*. Each task is therefore characterized by the processors (or slots) that are capable of processing it. Tasks are assigned to different slots according to their size and eventually a given scheduling strategy. Roman et al. (2006) is an example where the area of the reconfigurable array is divided into four unequal partitions, each holding one task at a time. A queue of tasks Q_i is associated to each partition P_i. The size of each partition is adjusted during run-time according to the profile of the tasks set being processed. Depending on its size, each arriving task is added to the queue of the partition that fits best. Each tasks queue Q_i is then scheduled on its corresponding partition P_i. Another example is shown in Ahmadinia et al. (2004) where the reconfigurable array is 1D-partitioned and each task is associated to a partition according to its finishing time. A monoprocessor assignment strategy along with theories could be then separately applied to each slot and its corresponding tasks queue.

To summarize, if applied to scheduling for multitasking or hardware virtualization on reconfigurable hardware devices, the *partitioned scheduling* strategy is similar to an m monoprocessors scheduling, where each processor or slot is locally managed by a monoprocessor scheduling scheme. Hence, hardware tasks scheduling strategies could be directly derived from partitioned scheduling theory for heterogeneous multiprocessor systems.

Global Scheduling

As global or non-partitioned scheduling is meant to use a global assignment scheme, the tasks are not partitioned, and any task could be executed on any processor of the multiprocessor platform. Hence, a global scheduling could be used even if the reconfigurable array is managed in a slot-

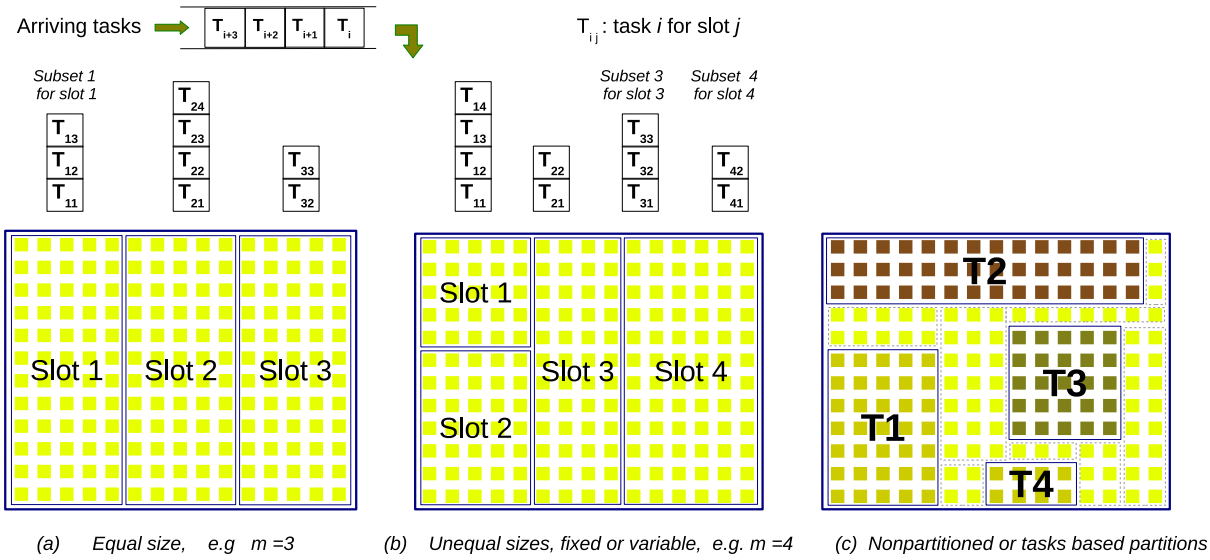


Figure 3.8: Equal sizes and unequal sizes partitioning of a DPRHW (dynamically and partially reconfigurable hardware device)

based approach. Once again the basic assumption being on one hand the fixed number of slots, and on the other hand the ability for the smallest slot to accommodate the biggest task. Such a situation would correspond to the well studied global scheduling of software tasks on m identical processors. Unfortunately, hardware tasks scheduling on partially reconfigurable hardware devices is more challenging.

In the model in figure 3.8 (b), the slots are not equally partitioned. Furthermore the number of slots could be dynamically changed over time. For example in Ahmadinia et al. (2004) which used a 1D partitioning, the number and the width of the slots are dynamically modified according to the characteristics of the incoming tasks.

In Lu et al. (2008) the reconfigurable array is first pre-partitioned in slots of various heights. Afterwards, they are merged vertically and/or horizontally on demand in order to fit wider or taller tasks (e.g. in figure 3.8 (b), slot 3 and slot 4 can merge to fit a wider task).
Figure 3.8 (c) depicts another model where there are any predefined partitions. As tasks are placed and removed, the array is split and merged accordingly. The number and size of partitions (if considered as) are continuously variable, depending on the number and the shape of the placed tasks. Hence, unlike the above-mentioned array partitioning strategies, neither the number (m) nor the size of processors are known beforehand.

It is much more difficult to clearly map any multiprocessor scheduling approach on these above-mentioned models. However if considered as an heterogeneous multiprocessor systems with a variable number (m) and size of processors, a *non-partitioned or global scheduling* strategy is more likely to be used. Indeed, the latter strategy reflects a more realistic behaviour of a reconfigurable array which is not pre-partitioned in slots. As it will be seen later, the main drawback is that the underlying area management is very difficult to handle.

To summarize, hardware tasks scheduling on a dynamically and partially reconfigurable hardware is akin to scheduling on a multiprocessor system, but with a continuously variable number and speed (size) of processors over time. Therefore, it is more complicated by consideration of this dynamicity. In addition, the resulting area management is highly complex because numerous splitting and merging operations. However, a partitioned approach is more likely to fit in a multiprocessor scheduling strategy and leads to a more simple area management. This results in a higher reconfigurable hardware device area fragmentation and a lower utilization ratio.

3.7 Online Real-Time Scheduling on Reconfigurable Hardware Devices

Introduction

As previously stated, scheduling policies can be classified as *static* or *dynamic*. In static scheduling tasks parameters are assigned beforehand and remain unchanged during the tasks life time, unlike dynamic scheduling where the scheduling relies on tasks parameters that change over time. *Offline scheduling* is also differentiate from *online scheduling*. The latter introduces a certain dynamicity in the system, making it difficult to find optimal scheduling solutions.

Online here means that the flow of the program is unknown in advance and hence task characteristics (arrival time, shape, size, execution time, deadline, etc...) are unknown before its arrival. This corresponds to the clairvoyant paradigm presented earlier. Hence, the scheduler has to dynamically reassess at runtime the task(s) to be placed (e.g. O. Diessel and Schmidt, 2000; Ahmadinia et al., 2004).

In this thesis, online real-time scheduling algorithms are classifies in two main families, depending on whether the scheduling algorithm exploits or not the fact that online real-time tasks are clairvoyant. Indeed, as the execution time of each task is known at its release, the scheduler

could rely on currently running tasks to determine current and future states of the reconfigurable array, in order to properly place or plan each new task. This family of scheduling is denoted as *looking-ahead scheduling*, in opposition to *without-looking-ahead scheduling*.

3.7.1 Online Scheduling *Without-Looking-Ahead* and Related Work

While scheduling *without-looking-ahead*, ready tasks are scheduled only on areas that are currently available on the reconfigurable device. If there is not enough free area to accomodate the task at current time and if it can still meet its deadline, the task is managed according to the scheduling policy (e.g. simply rejected or kept in a waiting list in order to be placed later if worthy). This approach is the most used in online and offline scheduling of hardware tasks on reconfigurable hardware devices (e.g. Bazargan et al., 2000; Ahmadinia et al., 2004; Danne and Platzner, 2005).

Without-looking-ahead scheduling approach is detailed in figure 3.9 where T_1, T_2, T_3 and T_4 are to be scheduled on the FPGA. a_i, e_i, l_i, s_i and f_i are respectively the arrival time, execution time, laxity, starting time and finishing time of task T_i. Tasks T_1 and T_2 are placed at time $t = 1$ as soon as they arrive. At time $t = 2$, tasks T_3 and T_4 arrive and cannot fit immediately on the reconfigurable hardware. In the *without-looking-ahead* approach, the scheduler (through the placer or area manager) checks areas that are available only at current time $t = 2$ without prospecting the future state of the reconfigurable hardware. Hence, tasks T_3 and T_4 are kept in a ready list as long as they can still meet their deadline, for further attempts. At time $t = 6$ when T_2 is completed, it could be replaced by T_3 or T_4 depending on the scheduling policy. If T_3 is the next task elected and if the system is submitted to hard real-time constraint, T_3 will be placed at time $t_{p3} = 6$ while T_4 will be rejected at time $t_{rej4} = 7$, because T_4 won't be able to meet its deadline anymore.

One of the main drawback of *without-looking-ahead* approach in hard real-time scheduling is to keep the tasks that will never be placed anyway in a ready or waiting list and reject them too late at time t_{rej}, expressed by equation 3.11

$$t_{rej} = a_i + l_i \qquad \Rightarrow \qquad Rd_i = t_{rej} - a_i = l_i \tag{3.11}$$

where t_{rej} is the rejection time of the rejected task T_i, a_i its arrival time, l_i its laxity, and Rd_i its rejection delay (the waiting time for a rejected task, detailed in Chapter 4, equation 4.23 page 162). Such a late rejection is not desirable as it prevents the operating system from considering

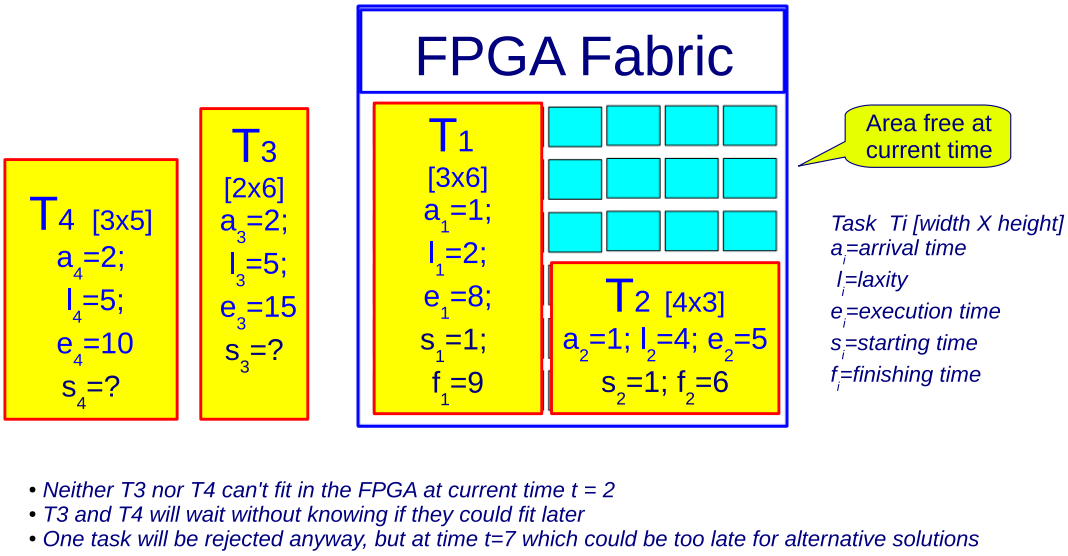


Figure 3.9: *Looking-ahead* vs *without-looking-ahead* scheduling approaches

tasks implementation on other resource than the reconfigurable hardware[12]. In addition, keeping the tasks lengthens the ready tasks list and, the longer the ready tasks list, the more difficult it is to scan and/or sort it, especially in an online real-time scenario.

Related Work

Bazargan et al. (2000) presented online and offline scheduling of hardware tasks on reconfigurable hardware devices through a bin packing approach. For online scheduling, their studies are mainly focused on 2D placement strategies along with area partitioning and management. The simulation results are provided with respect to different classes of tasks, where a class refers to the sizes (width, height) of tasks. However, there is no time-based scheduling strategy (e.g. EDF) and runtime overhead of online algorithms are not measured. As these studies mainly coped with placement, they are presented in depth later below.

Danne (2006) dealt with the problem of scheduling real-time and periodic tasks on partially reconfigurable hardware devices. He formalized the real-time scheduling problem. In its model, the reconfigurable hardware device is seen as a homogeneous multiprocessor platform that consists

[12] alternative resources could be hardcore and/or sotfcore CPUs. Sometime, the remaining reconfigurable resource may not be enough to accommodate a given hardware task, but may fit an instantiated softcore processor that can run the task.

of m identical processors, each processor being capable of running any task. Consequently, m tasks instances can concurrently run on the device as long as the sum of their area does not exceed the area of the device (expressed by equation 3.12 for the 1D model).

Danne (2006) proposed three preemptive scheduling algorithms: global EDF (Earliest Deadline First), partitioned EDF and server based. These algorithms are adaptation of well-known software tasks scheduling algorithms for multiprocessor platforms.

Danne (2006) presented two variants of global EDF scheduling for reconfigurable hardware devices, denoted respectively as *EDF-First-k-Fit* and *EDF-Next-Fit*:

- At each scheduling time, *EDF-First-k-Fit* selects the k first jobs (in the sorted list of active jobs) that can fit on the array. The k first jobs are selected in such a way that the k^{th} job J_k may be placed on the array only if all the jobs J_1 to J_{k-1} preceding J_k in the list can also fit on the array. This leads to a situation where a remaining free area may be kept idle while there are some active jobs J_{k+i} that can fit on it.
- *EDF-Next-Fit* scheduling overcomes this limitation of *EDF-First-k-Fit*, as the algorithm scans the list of active jobs and places as many jobs as possible on the array, as far as they fit in the array. Therefore *EDF-Next-Fit* outperforms *EDF-First-k-Fit* as in the latter, a free area that can fit an active job may remain unused.

However, Danne (2006) relied on the 1D model of reconfigurable hardware devices. This model requires rectangular-shaped tasks and assumes that each hardware task spans the entire device width instead, tasks' heights being proportional to tasks' areas. The model is simpler to manage compared to 2D model. It eases task relocation and therefore reduces external fragmentation. But it induces internal fragmentation that leads to a lower device utilization ratio.

Using a 1D model makes the condition for simultaneously running n jobs $J_{i=1...n}$ on the reconfigurable array simple as follows:

$$\sum_{i=1}^{n} w_i \leq W \tag{3.12}$$

where w_i and W are respectively the width of task J_i and the width of the reconfigurable hardware device. Therefore, the schedulability analysis is simpler and can rely on schedulability analysis for multiprocessor scheduling. For example, Danne and Platzner (2006b) relied on schedulability analysis for EDF algorithm upon multiprocessor to provide an efficient though pessimistic schedulability analysis for global EDF scheduling. This scheduling test is of linear complexity $O(n)$, n being the number of tasks. The test guarantees at design time that no deadline will be missed. Hence, any tasks set which passes the test will be feasibly scheduled by EDF algorithm. However,

as the test gives a sufficient but not necessary condition, a tasks set that fails may be feasibly scheduled.
At that period, the other advantage of using 1D placement was technological limitations of FPGAs. FPGAs were providing only a column-wise partial and runtime reconfiguration. Hence, the only way of performing a 2D placement was to totally reconfigure all the columns (and therefore the tasks in these columns) that were interfering with the task to place. Such a process were likely to affect many tasks at each task placement, making reconfiguration overheads higher.
However, as discussed later in section 3.9.1 page 124, any advantage of 1D placement (including its lower algorithm complexity) comes at the cost of an internal fragmentation that lowers the reconfigurable array utilization ratio and increases tasks rejection. Fortunately, using a 2D placer is getting more meaningful as nowadays, many FPGAs are enabling 2D partial reconfiguration.

Partitioned EDF scheduling is presented in Danne and Platzner (2006a) where an extended model of ILP (Integer Linear Programming) for bin-packing problem is developed in order to compute the optimal partitioned schedule for a given set of tasks. The reconfigurable array is horizontally partitioned instead (in opposition to vertical partitions as mapped in figure 3.8-(a) page 92), following the 1D area model. At any time, at most one hardware task is allocated to a slot. Consequently, the width of the task spans the entire width of the slot, leading to a low device utilization ratio. Each task is modeled using many variants. Therefore, ILP selects a variant for each task and a partitioning that minimize the overall required device area. That is, the smallest area that allows to feasibly schedule a tasks set is found. Albeit the optimal partitioning for medium-sized tasks sets can be computed in reasonable time, ILP-based partitioning scheduling is not suitable for online scheduling.

Server-based approach is detailed in Danne et al. (2006) through MSDL scheduling technique. In MSDL technique, tasks are grouped and executed on the reconfigurable array according to the time sharing monoprocessor scheduling model presented earlier and mapped in figure 3.5 page 84 and figure 3.7 page 86. The reconfigurable device is fully reconfigured and a server corresponds to what was earlier denoted as a compound task. As stated while discussing about that model, one of the challenge here is to minimize the number of device configurations, which corresponds to the number of required bitstream files. This number of configurations is bounded by the number of tasks. This scheduling technique offers an offline schedulability test. Hence, different combination of tasks that have to be run concurrently are computed beforehand and synthesized in order to be loaded at the appropriated time, following a given scheduling policy (e.g. global EDF).

Ahmadinia et al. (2004) presented a cluster based dynamic scheduling in which the FPGA is partitioned into slots of dynamic size. Tasks which will complete at nearly the same time are placed in the same slot in order to free up contiguous spaces at the same time and create large empty rectangles for later placement. Even if this scheduling strategy takes into account the future state of the reconfigurable array by assigning slots to tasks according to their completion time, it remains a without-looking-ahead approach as at each time, the tasks are scheduled only on the currently available areas.

Conclusion on *online scheduling without-looking-ahead*

To summarize, the section discussed different work on *online scheduling without-looking-ahead* of real-time tasks on reconfigurable hardware devices. Bazargan et al. (2000) first presented both optimal and nonoptimal approaches in area management for online and offline placement. Danne (2006) formalized real-time scheduling of periodic and preemptive tasks, and provided few algorithms along with efficient schedulability analysis that derived from multiprocessor scheduling. Ahmadinia et al. (2004) proposed a cluster-based scheduling approach that tends to free contiguous areas at nearly the same time in order to accommodate next tasks. These different approaches showed the tightness that exists between scheduling and placement problems. The coming section introduces *looking-ahead* scheduling and emphasizes the aforementioned tightness.

3.7.2 Online *Looking-Ahead* Scheduling and Related Work

Presentation

In *looking-ahead scheduling*, if there is not enough place at current time to place a given task, the scheduling algorithm goes further by looking into the future, mimicking the end of certain running tasks to see if the space thus freed can fit the task (e.g. Steiger et al., 2004; Chen and Hsiung, 2005; Marconi et al., 2008). By doing so, the scheduler either accepts or rejects the task immediately and, therefore, allows the Operating System to find alternative implementation solutions in case of rejection.

In an online real-time context, this immediate rejection of any task T_i that cannot fit in the array is the main advantage of a *looking-ahead* approach. The rejection time t_{rej} of a task T_i scheduled with a *looking-ahead* scheduling is expressed in equation 3.13 where a_i is the release time of T_i and ϵ_i the scheduling runtime overhead or time required by the scheduling algorithm to schedule T_i. Rd_i is the corresponding rejection delay (or the waiting time for a rejected task, as detailed

later in equation 4.23 page 162).

$$t_{rej} = a_i + \epsilon_i \qquad \Rightarrow \qquad Rd_i = t_{rej} - a_i = \epsilon_i \tag{3.13}$$

Equation 3.13 implies that the rejection delay is equal to the *looking-ahead* scheduling algorithm runtime overhead, which must be far lesser than a scheduler tick or time unit ($\epsilon_i \ll T_{tick}$).
Looking-ahead algorithms certainly improve the online real-time scheduling quality by taking rapid scheduling decisions. However when a 2D placement is used as in Steiger et al. (2004), too many areas splitting and merging operations are performed at runtime, which is prejudicial in online real-time scenarios.

Related Work

Steiger et al. (2004) proposed two looking-ahead scheduling algorithms denoted as *Horizon* and *Stuffing*. Both algorithms are online and clairvoyant. They schedule tasks to arbitrary areas that are either currently free or that will be free at a given time (or time interval) in the future. A task that is assigned a future free areas is put on a reservation list while the currently running tasks are recorded on an execution list.
As pictured in figure 3.10, the algorithms differ from each other in the way they manage the areas. At any scheduling time, before assigning any area to tasks, the area manager makes sure that the area is not conflicting with any other area currently occupied or already booked for another task. This verification is made using the two above-mentioned lists, in addition to a third list that records the state of the reconfigurable hardware device. In *horizon* scheduling, once an area is assigned to a task, no matter when the tasks starts its execution, the area is marked as occupied from current time until the end of the task. Hence, as shown in figure 3.10(a), areas are marked as either "totally free", or "occupied from the current time until a given point in time". This leads to situations pictured on the left of figure 3.11 where some reserved areas are available within a time interval but cannot be used. The shaded parts in figure 3.11(a) are such lost areas. As depicted by the latter figure, task T_7 that arrives at time 3 is planned to start at time 18 when task T_6 will end. Indeed, T_6 has been planned earlier at time 2 to be started at time 15. Hence, from time 2 when T_6 has been planned, the whole space that will be occupied by T_6 cannot be assigned to any other newly arrived task within the interval that spans from the reservation of task T_6 (at time 2) until its end at time 18.
In order to draw a parallel between hotel management and horizon scheduling, let's assume that a room is booked for a client who is arriving in one week time and who is planning to stay one

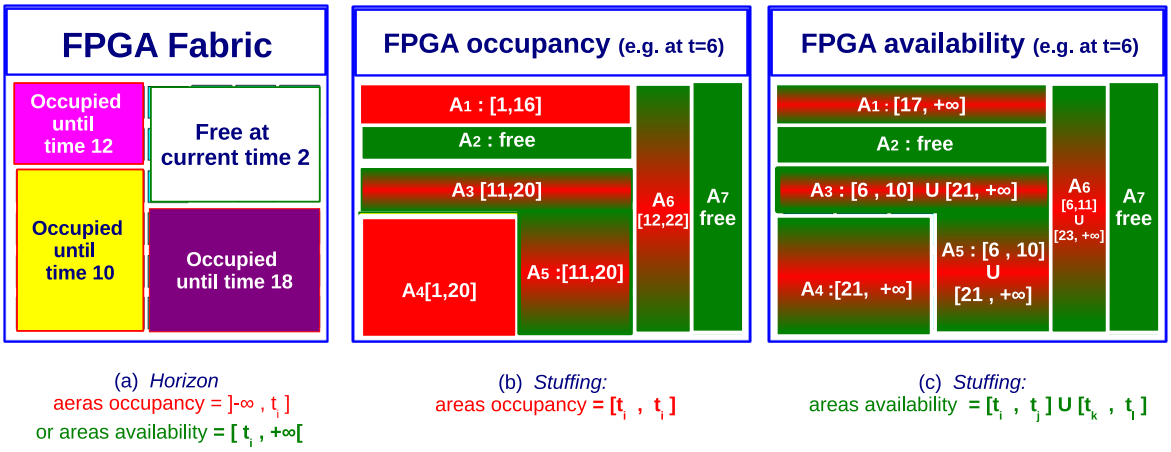


Figure 3.10: Managing areas availability or occupancy in *looking-ahead scheduling* (e.g. horizon and stuffing algorithms, Steiger et al., 2004)

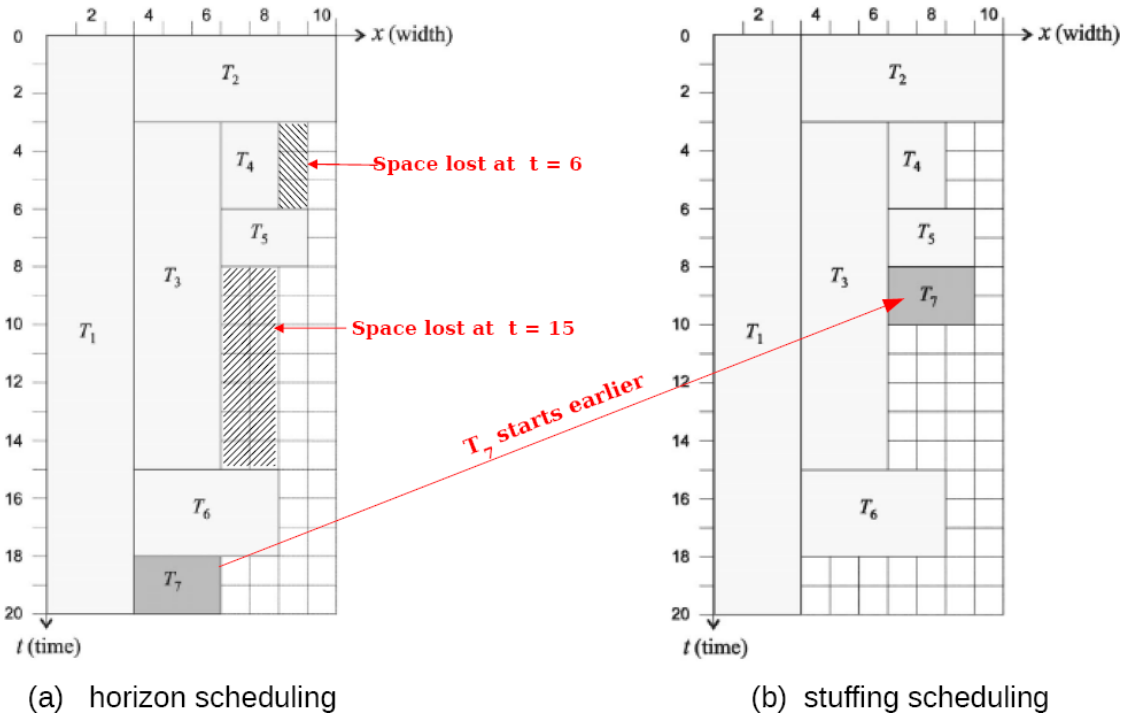


Figure 3.11: An example of 1D Horizon and Stuffind scheduling algorithms (Steiger et al., 2004)

week in the hotel. In horizon scheduling, the room is no longer available from the booking time until the end of the stay which is in two weeks. This scheduling approach prevents the room from being rented during the week preceding the client arrival. Therefore, the occupation ratio of the hotel is lower, where this ratio corresponds to the reconfigurable hardware utilization ratio in our present case.
The aforementioned lack of *horizon* scheduling is overcome by *stuffing* scheduling. Indeed, as illustrated in figure 3.10(b), areas on the reconfigurable hardware are marked either as "totally free" or as "occupied in a precise time interval". Figure 3.10(c) shows the same information, but from availability point of view. This is similar to rooms management in hotels where rooms are marked according to their availability. Hence many clients (resp. tasks) may share the same room (resp. area) as long as it is on a time-sharing basis. In figure 3.11 the same sequence of tasks is scheduled using *horizon* and *stuffing* algorithms. Task T_7 starts earlier with stuffing algorithm. Consequently, as shown by Steiger et al. (2004) both for 1D and 2D placement, for the same placement strategy, stuffing scheduling algorithm reduces the makespan and the tasks rejection ratio while increasing the device utilization ratio compared to horizon scheduling. These improvements come at the cost of higher scheduling algorithm runtime overhead.

Chen and Hsiung (2005) proposed the *Classified Stuffing (CS)* algorithm as an improvement of the 1D version of the original stuffing scheduling (Steiger et al., 2004). In Classified Stuffing, hardware tasks are classified in two types and the placement location of each class of tasks is different. This differs from the original stuffing (OS) algorithm where when an available area is found, the task is placed on its leftmost in the case of 1D placement (on its bottom left corner in the case of a 2D placement). Classified Stuffing can place a task either on the leftmost or the rightmost of the available area depending on the task Space Utilization Rate (SUR), given by :

$$SUR_i = \frac{w_i}{e_i}$$

where SUR_i is the space utilization rate of the task to place T_i, w_i and e_i respectively its width and its execution time. Tasks with a $SUR > 1$ are placed starting from the leftmost available area while tasks with $SUR \leq 1$ are placed from the rightmost available columns. As shown through simulations performed with a large number of tasks sets with various SUR (4, 1 and 0.25), Chen and Hsiung (2005)'s 1D stuffing approach reduces the fragmentation (from $\sim -5.5\%$ to $\sim -23.3\%$) and the makespan (from $\sim -4\%$ to $\sim -21\%$) for the same tasks rejection ratio and almost the same algorithm runtime overhead, compared to original stuffing. Obviously, best

results are obtained with tasks set with low SUR. Indeed, as Chen and Hsiung (2005) uses a 1D placement, the smaller the width of the task, the lower the resulting internal fragmentation.

Cui and Deng (2007) proposed a One-Level Look-Ahead scheduling strategy. Indeed as fragmentation is one of the main problem to overcome, their work coped with finding a fragmentation-based scheduling policy, as presented later below in section 3.9, page 122. Their ultimate proposition is a *looking-ahead* approach in tasks placement, which aims at reducing the area fragmentation of the reconfigurable hardware device by delaying the placement of a task to the next event, increasing the solution search space. Consequently, the placement of an arriving task could be delayed even if there is enough place on the device to place the tasks at its release time. The basic idea behind this strategy is that this delay is sometimes more benefitial in terms of area fragmentation. In their so-called *one-level looking-ahead*, a task released at time t_a could be delayed at most until time $t_a + \delta$ if the task can still meet its deadline, and if the resulting overall fragmentation of the device is lower compared to an immediate placement at time t_a. One-level here means that within the time interval $[t_a\ ;\ t_a + t_l]$ where t_l is the laxity of the task, the algorithm checks the first time instant $t_a + \delta$ when one (or more) currently running task finishes its execution and therefore frees more spaces. The fragmentation resulting from a placement at time t_a is then compared to the one produced by a placement at time $t_a + \delta$, the starting time that is more benefitial in terms of fragmentation is choosen accordingly.

Cui and Deng (2007) was actually an improvement of a similar scheduling approach that has previously been proposed by Tabero et al. (2006). Both approaches were fragmentation-based as they both tried to minimize the fragmentation. However they differed on the way they managed their free areas. Tabero et al. (2006) used a *Vertex List* to keep the track of the free areas in the FPGA while Cui and Deng (2007) used a MER-based approach. Albeit fragmentation-based scheduling approaches, neither Tabero et al. (2006) nor Cui and Deng (2007) are not fully a *looking-ahead* approach as the algorithm only looks a single (and closest) time instant in the future, instead of looking at all the events in time interval $[t_a\ ;\ t_a + t_l]$ which are likely to free more space on the reconfigurable device. However, it outperforms Handa and Vemuri (2004a) and Tabero et al. (2006) in terms of tasks rejection ratio and device utilization ratio.

Marconi et al. (2008) proposed the *Intelligent Stuffing (IS)* algorithm as an improvement of the original stuffing (Steiger et al., 2004) and the classified stuffing (Chen and Hsiung, 2005). Therefore, *IS* is a 1D looking-ahead stuffing algorithm which provided better results compared to

the previously mentioned 1D stuffing algorithms. As in the classified stuffing, IS algorithm differs from the original 1D stuffing in the criteria used to place a task either on the leftmost or on the rightmost of the available area. While tasks positions are SUR-based in CS algorithm (Chen and Hsiung, 2005), Marconi et al. (2008)'s approach introduces the so-called alignment status of a free space. Hence, each free space (FS) is assigned a boolean parameter (alignment status). The latter indicates the side (leftmost or rightmost) of the free area where to place the next task that will be accomodated to the area.

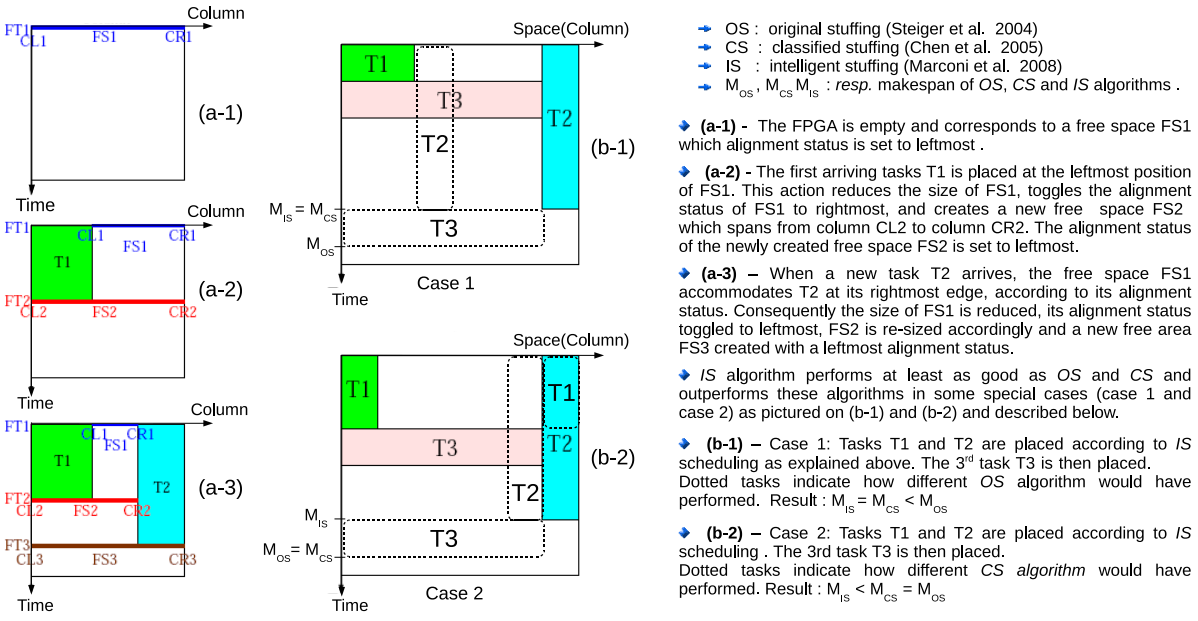


Figure 3.12: Intelligent Stuffing (IS) scheduling algorithm using 1D placement (Marconi et al., 2008)

Everytime a task is placed in a free area, the alignment status of the remaining free area is toggled. One example is mapped in figure 3.12-(a-1) where the whole area of an empty FPGA is represented by a single area denoted as FS_1, spanning from column CL_1 and CR_1 and with an alignment status set to leftmost. That is, the first arriving tasks T_1 is placed at the leftmost position of FS_1 as pictured in figure 3.12-(a-2). This action reduces the size of FS_1, toggles the alignment status of FS_1 from leftmost to rightmost, and creates a new free space FS_2 which spans from columns CL_2 and CR_2. As for any newly created area, the alignment status of FS_2 is set to leftmost. If a new task T_2 arrives, the free space FS_1 will accomodate the task at its rightmost edge, according to its current alignment status. Consequently as illustrated in figure 3.12-(a-3),

the size of FS_1 is reduced, its alignment status toggled to leftmost, FS_2 is resized accordingly and a new free area FS_3 created with a leftmost alignment status.

Figures 3.12-(b-1) and (b-2) show two scenarios of 3 tasks (T_1, T_2 and T_3, not dotted) scheduled using IS scheduling algorithm. In figure 3.12-(b-1) the dotted tasks T_2 and T_3 show how the original stuffing (OS) would have scheduled tasks T_2 and T_3. The makespan resulting from OS scheduling is greater compared to IS scheduling ($M_{OS} > M_{IS}$). A similar result ($M_{OS} = M_{CS} > M_{IS}$) arises from the second scenario mapped in figure 3.12-(b-2) where the dotted version of tasks mimics a classified stuffing scheduling. IS provides smaller makespan as it allows future tasks to be placed earlier, thanks to the area management based on the alignment status. The latter is easier to compute than the SUR used by CS algorithm. Consequently, IS reduces the average waiting time (up to 12.8%) and decreases the total wasted area (up to 53%) for slightly the same algorithm runtime overhead, compared to CS scheduling.

Conclusion on *online looking-ahead* scheduling

To summarize, this section discussed different works on *online looking-ahead* scheduling of real-time tasks on reconfigurable hardware devices. Steiger et al. (2004) first presented two main approaches denoted respectively as *horizon* and *stuffing*, and that used 1D and 2D placement strategies. For obvious reasons recalled earlier, stuffing algorithm outperforms horizon algorithm with respect to the same placement strategy. In addition, as it produces less fragmentation, 2D placement provides better results compared to 1D in terms of reconfigurable hardware utilization ratio and tasks rejection ratio.

Chen and Hsiung (2005) and Marconi et al. (2008) have respectively proposed *CS* and *IS* algorithms as improvements of the 1D version of stuffing algorithm. Their algorithms provide a better quality placement and a shorter makespan for slightly the same runtime overhead. However, the results remain far from being optimal as the quality placement of a scheduling strategy is highly tied to the quality of the underlying placement algorithm (which is 1D here) along with its area management strategy.

Looking-ahead scheduling requires to regularly mimic future tasks placement and withdrawal in order to assess future states of the reconfigurable array. This leads to numerous area splitting and merging operations that can be highly complex depending on the placement algorithm. Consequently, unlike Steiger et al. (2004) other related works in looking-ahead scheduling in an online real-time context tend to use a simple 1D placer. The resulting runtime overhead and overall complexity are then affordable in this context, but come at the cost of reconfigurable hardware

device utilization ratio.

3.8 Tasks Placement and Related Work

3.8.1 Online Placement Issues

Placement aims at efficiently allocating reconfigurable area to different modules (tasks) of an application to implement. The problem of placing modules on the reconfigurable device is simpler if the device is big enough to concurrently fit all the modules of the application. In this case, a placed module is permanently on the device and the placement problem consists of finding a position (if one exists) to each task in a way that all the tasks fit on the reconfigurable array. However, if the chip is not big enough, there is a need to perform a hardware virtualization (or multitasking) where modules could be dynamically swapped in and out the chip according to their idle or operating time. This process of assigning both a position and a starting time to each task makes the placement problem more challenging by turning it into a scheduling problem as previously described.

To achieve a good placement, one needs to have a model for each component involved as detailed in Chapter 4. In most studies, the reconfigurable fabric (e.g. FPGA) is seen as an array of resources with a given size (area) and hardware tasks as rectangular modules to be placed on the array. Basically the modules and the reconfigurable area are rectangular-shaped. Hence, the problem of placing modules on an FPGA is similar to the well-known 1D and 2D bin-packing problems as presented in Coffman et al. (1997). 1D and 2D bin-packing correspond respectively to 1D and 2D placement; a 3rd dimension is commonly added for time flow, in order to bring out the scheduling problem.

In hardware tasks scheduling for reconfigurable platforms, the placement problem appears as a scheduling sub-problem. Hence, a *Scheduler* assigns the management of the reconfigurable array to a *Placer*, sometimes denoted as *resource manager*. The placer reacts upon scheduler requests. In general, the placement problem is usually divided in two main parts (detailed below and depicted in figure 4.12, page 157, Chapter 4):

(i.) ***area manager***

most of the time, the *area manager* is a *free space manager* as it manages the free space still available on the reconfigurable fabric. Hence, it maintains a data structure that stores information about the state of the reconfigurable fabric (free spaces, occupied spaces). Con-

sequently, the manager is invoked to update the state of the chip at every change (task placement or withdrawal). Most work on placement differ from each other depending on the data structure that stores the state of reconfigurable area, and on the strategy used to manage it. The data structure and its update deeply impacts on the complexity of placement algorithms and heuristics.

(ii.) ***area finder***

the *area finder* checks and assigns a rectangular area on the reconfigurable array that could accommodate a task, on request of the scheduler. It uses a given rule (e.g. bin-packing rules) to fit the task to one of the free spaces maintained by the free space manager. In addition, it uses different fitting strategies (see section 3.8.4, page 113) to decide which area among the possible candidates will be assigned to the task to be placed.

The *area finder* and the *free space manager* are tied as the strategy of finding and choosing one area among others depends on the data structure that keeps a record of the available areas. Consequently, related work on placement will be presented first, through different free areas management strategies, sometimes referred to as areas partitioning strategies. Afterwards, another comparative study of placement strategies will be done, from reconfigurable area fragmentation perspective.

The special case of offline placement

If the flow of the program is predictable at design time, and all the parameters (timing information, size, etc.) of different tasks or modules to be inserted are known beforehand, the placement of different modules on the reconfigurable hardware could be planned at compilation time. Hence, the reconfigurable resources are statically and deterministically assigned to different modules at the compilation time. In offline scheduling and placement, high performance algorithms and heuristics can be developed to optimally manage the reconfigurable fabric area. So, even if these very efficient algorithms compute slowly, they remain affordable as computations are done offline (e.g. on a host PC before the system starts running). Today, compilation times of FPGA-based designs are dominated by placement and routing. FPGAs are becoming denser. The designs to implement are more complex, requiring tremendous amount of interconnections to be routed. EDA tools require great amounts of CPU time (even hours) to achieve a high quality placement and routing. Different high-quality P&R[13] heuristics have been developed over the years to solve

[13] Place and Route

placement and routing in FPGAs. Further, EDA tools enable incremental placement and modular placement for modules based designs. In a modular placement, a rectangular portion of the FPGA is assigned to each module. The module is placed and routed within this portion. This feature can be exploited in online placement. However, these optimal P&R heuristics used in EDA tools are too greedy to be used in an online scenario. They are suited to offline placement (e.g. running on a desktop PC) and are not in the scope of this thesis.

3.8.2 Free Area Partitioning

As stated earlier, the *area manager* keeps and updates a data structure representing the current state of the reconfigurable area in terms of empty spaces or inserted modules. It also divides the empty space in empty rectangles. As detailed below, there are many ways of splitting and managing an area that accomodate a smaller task.

Nonoverlapping *vs* overlapping area partition

The area manager keeps the track of available free areas as a list of rectangles. These rectangles result from different partitions that are performed after tasks placement. Depending on the partition strategy applied on the remaining empty space, there are two types of generated rectangles as pictured in figure 3.13:

(i)- *Nonoverlapping rectangles* that result from a nonoverlapping partition, as depicted in figure 3.13-(a) and 3.13-(b).

(ii)- *Overlapping rectangles* that result from an overlapping partition, as depicted in figure 3.13-(c).

As seen in figure 3.13-(a) and 3.13-(b), when a task T_1 is placed in a rectangle, the remaining free area is split up into several (two here) nonoverlapping rectangles, R_{11} and R_{12}. The split could be either *vertical* or *horizontal*.
In nonoverlapping partition, the number of rectangles to manage grows linearly with respect to the number of placed tasks. Hence the time complexity of placing or removing a task is in general $O(n)$ where n is the number of tasks currently placed. The reason is that only the accommodating rectangle is involved in the placement/withdrawal process.

Figure 3.13-(c) depicts another option where the resulting rectangles are overlapping. Indeed, as detailed in figure 3.14, generating overlapping rectangles improves the placement quality. The example given shows that task T_2 would have been rejected if rectangles R_{11} and R_{12} resulting

from the placement of task T_1 were not overlapping. More precisely, T_2 would have been rejected in the event of a vertical split. T_2 is placed in rectangle R_{11}. R_{11} is then split up into two rectangles R_{21} and R_{22}. This simple example shows that overlapping partitioning decreases the number of rejected tasks by providing more solutions for fittings.

However, overlapping partitioning induces numerous resizing processes. For example, as task T_2 overlaps with rectangle R_{12}, the latter has to be resized. Consequently, the number of rectangles involved in a task placement/withdrawal is one order of magnitude greater if compared to the case of nonoverlapping partitioning. In addition, when using overlapping partitioning the resulting rectangular space partition is quadratic with respect to n, the number of tasks currently placed on the chip. Consequently, the time complexity of inserting/removing a task is $O(n^2)$.

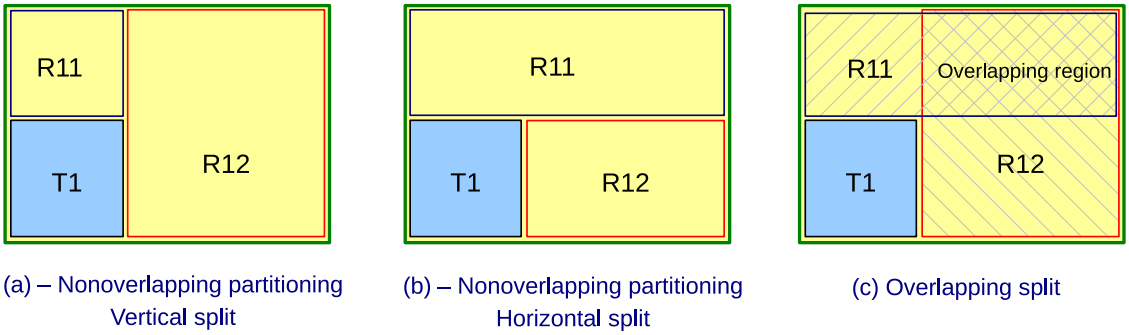


Figure 3.13: Nonoverlapping *vs* overlapping partition; vertical *vs* horizontal split for overlapping partition

MER (Maximum Empty Rectangles)

A maximum empty rectangle is an empty rectangle that cannot be completely covered by any other empty rectangle. Figure 3.14-(c) shows one way of managing the remaining free area by splitting it up using the overlapping partition approach. Furthermore, another overlapping approach is mapped in figure 3.14-(d) where all possible maximum empty rectangles that could be built in the remaining free area are identified. R_{12}, R_{21} and R_{22} are these so-called MERs (Maximum Empty Rectangles).

The difference between figure 3.14-(c) and figure 3.14-(d) is on the size of R_{22}. R_{22} is bigger in the latter figure thanks to a MER-based partition. Therefore, a MER-based partition provides more candidates for placing a new task. The placement is optimal since the list of MERs will contain a placement solution for any task, if one exists. However, inserting/removing a task has

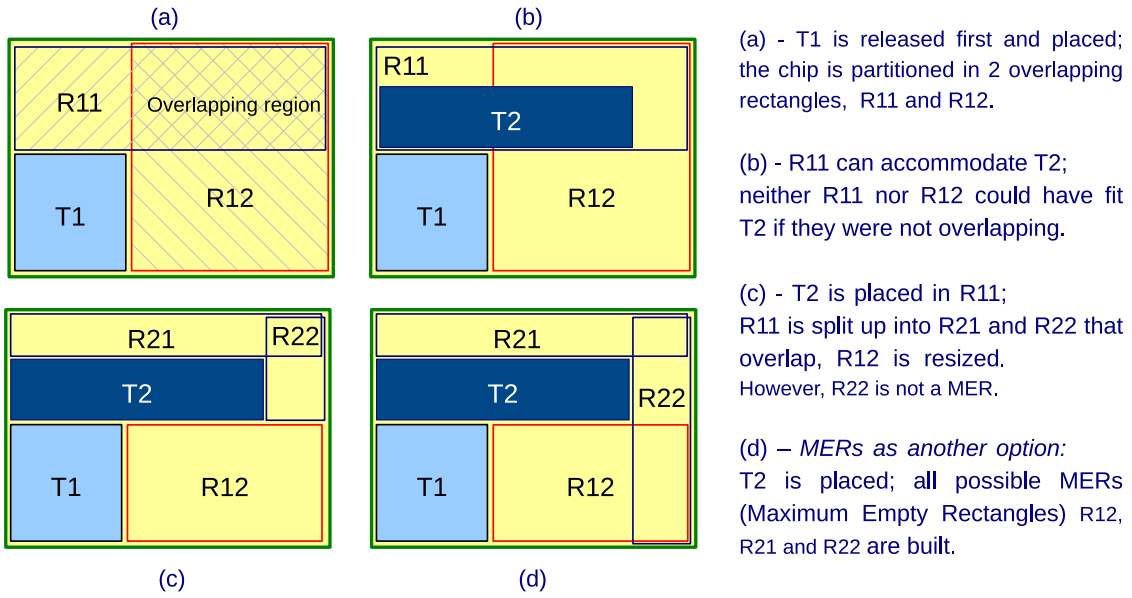


Figure 3.14: Placing a task in an overlapping rectangle

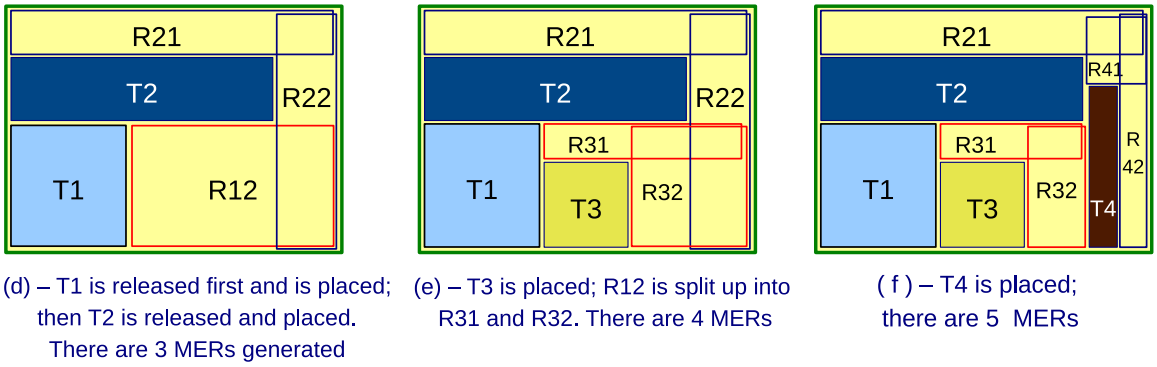


Figure 3.15: Maximum empty rectangles

time complexity $O(n^2)$, n being the number of currently placed tasks.

Another example of tasks placement in MERs is pictured in figure 3.15. It shows how the number of rectangles involved in each task placement or withdrawal grows drastically compared to the nonoverlapping case. Identically, the number of rectangles grows quadratically with respect to the number of placed tasks. In figure 3.15-(d) there are 5 rectangles for two currently placed tasks, T_1 and T_2.

3.8.3 Data Structure to store the State of the Reconfigurable Array

The most challenging part of placement problem is probably to build a data structure that represents the state of the reconfigurable array, e.g. in terms of available resources and their location. While trying to place a task, the algorithm checks existing free areas in the data structure. In addition, at each task placement or withdrawal on the reconfigurable array, the data structure is updated in order to reflect the new state of the array. The placement algorithm complexity and performance highly depends on the data structure. A good data structure should keep a complete record of existing areas, should be easy to scan for finding free areas, and should be easy to update. These objectives are conflicting. Indeed, the quantity of information required to represent the free rectangles grows with the accuracy of the representation. For example, recording all the MERs will require far more information than recording nonoverlapping rectangles. Hence, a reasonable trade-off has to be found while choosing the area management strategy.
No matter which data structure is used to represent the free of rectangles, remember that the number of rectangles involved in a task placement or withdrawal depends on the partitioning strategy used, as mentioned earlier. The updating process is consequently closely related. Depending on the way the data structure keeps the state of available areas and the resulting update process, one distinguishes 3 main data structures of various algorithmic complexity : list of overlapping/nonoverlapping rectangles, list of MERs and Vertex-List.

List of overlapping and nonoverlapping rectangles

In this approach, the reconfigurable array is represented by a list of overlapping or nonoverlapping rectangles. Each rectangle is represented by its coordinates and its size.

A binary search tree as adjacency graph :

Bazargan et al. (2000) mentioned a *binary search tree* used as an area adjacency graph that stores the state of the reconfigurable array. Figure 3.16 depicts an example of this tree. In this example, the tree stores nonoverlapping rectangles. When a task is placed in a rectangular area that is bigger than the task, the area is partitioned up into two or three parts according to vertical, horizontal or overlapping split. In the tree, each node represents an area and could generate up to two children nodes when a task is placed on it. R_1 represents the whole reconfigurable array and is the root of the tree. Its size is 10 X 6 where 10 is its width. Task T_1 is first placed on the bottom left of R_1. R_1 is then split up into 3 parts denoted as T_1, R_2 and R_3. R_2 and R_3 are

inserted in the tree as children rectangles of R_1 while the part denoted as T_1 host the task T_1. Then T_2 arrives and is placed in rectangle R_2. As T_2 spans the entire width of R_2, only one child rectangle R_4 is generated and inserted in the tree. The main observation at this stage is that each internal node of the tree corresponds to an area hosting a task, while leaves of the tree represent the current free areas on the chip. At this point in time after placing tasks T_1 and T_2, available areas are R_3 and R_4. Then arrives T_3 which is placed on R_3. R_3 is vertically split, R_5 and R_6 are generated. At this point, free areas are R_4, R_5 and R_6 and running tasks are T_1, T_2 and T_3. T_4 arrives and is placed on R_4 which generates R_7. The tree is updated accordingly. At this point, the tree is in the state pictured in figure 3.16, without the area R_X which is the second child area of R_2.

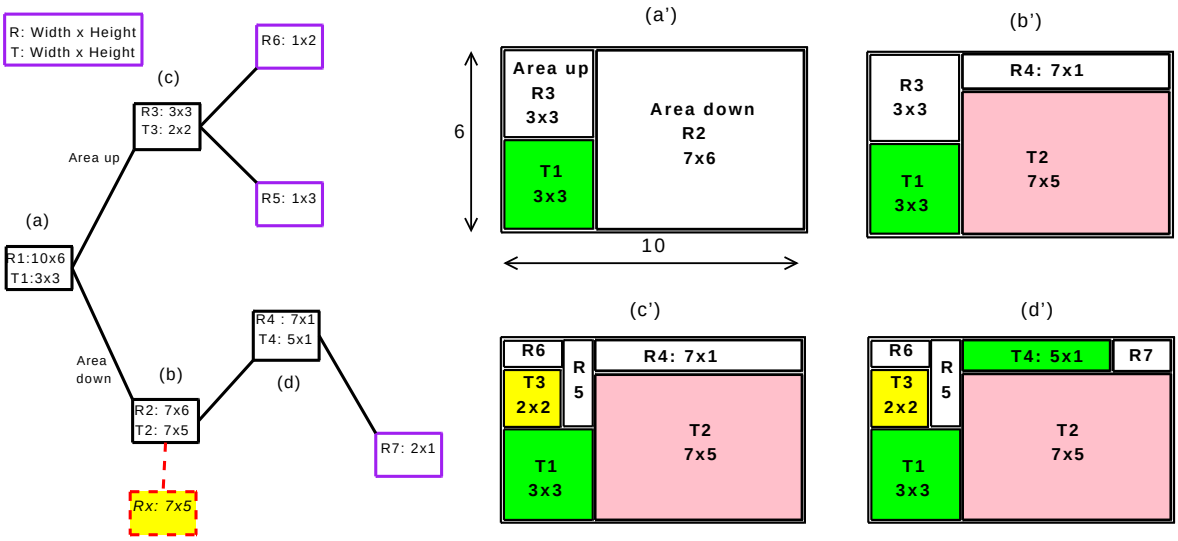


Figure 3.16: A binary tree used as a data structure that records the state of the FPGA

Storing the states of the reconfigurable array in such a binary tree structure makes merge and split operations more intuitive. Indeed, initial area of a node could be retrieved when all its children areas are free. For example, when task T_3, placed on node R_3 completes, the original size of R_3 is retrieved on the chip by moving back the tree to a former state. This is done by simply deleting children area nodes R_5 and R_6. However, a situation can arise when the task hosted in the node finishes while there is one or many tasks still running in the subtree. In this situation, an area node R_X is generated as pictured in figure 3.16, where the size of R_X is equal the size of the finished task. Indeed, if T_2 finishes before T_4, R_2 cannot be retrieved immediately. R_X is generated as an extra child node, in order to make it available for further placement.

The complexity of finding an internal node (that host a task) is bounded by $O(n)$ in the case of a binary tree, and by $O(log(n))$ in the case of a binary search tree, n being the number of tasks currently on the array; in addition if the binary search tree is balanced the complexity drops to $O(log_2(n))$. A binary search tree can be properly built based on the coordinates of the generated areas. For example, in the depicted case of figure 3.16, a vertical split is used. Hence, two children rectangles (e.g. *area down* R_2 and *area up* R_3) generated from the same father rectangle (R_1) will differ from each other by their X-coordinate. As shown on node (a) and (a'), rectangle R_3 along with its descendants will always be at the left of the X-coordinate of rectangle R_2. The X-coordinate may be a building criteria of the binary search tree. However in the case of overlapping rectangles, the criteria differentiating the *area up* from the *area down* while building the tree will be much more difficult.

A tree structure is usually used with an additional list that contains the available free areas (leaves of the tree). The reason is that checking such a shorter list to find a feasible placement is much more easier than scanning the tree.

To summarize, the complexity of managing a simple list is interwoven with the complexity of the partitioning strategy. A simple list provides a low complexity, but leads to a higher fragmentation especially in the case nonoverlapping partitioning. A tree structure eases the merging process and therefore limits the fragmentation.

List of MERs

Building a list of MERs at each task placement and withdrawal leads to high quality placement. The free area is partitioned in MERs as pictured in figure 3.15. By principle, finding all MERs assumes that the whole reconfigurable array is scanned. Hence, the worst case performance for finding all the MERs is $O(w \cdot h)$, w and h being respectively the number of columns and the number of lines of the reconfigurable array. However, many techniques have been proposed to identify and build all MERs. Most of them rely on the fact that each time a task is placed, the reconfigurable array is affected locally. Hence, only the MERs that overlap with the MER accomodating the task are updated.

Handa and Vemuri (2004c) described the *Staircase* algorithm as technique that finds MERs in a structure. The main advantage of the Staircase algorithm is that it scans an average of 15% of the whole array in order to update the list of MERs. Therefore, Staircase algorithm is of lower time complexity. This lower time complexity is obtained by keeping concurrently a table that contains

free and occupied cells in the array, and an encoding table that contains some coefficients. These data are used to build stairs of avalaible areas from which MERs are deduced.

Cui and Deng (2007) also proposed the ScanLine Algorithm (SLA), an algorithm that efficiently finds MERs in a reconfigurable array. It used the same principle as Staircase by concurrently maintaining a table that contained occupied and free cells in the array, and a table of coefficients. However the way of calculating the MERs were completely different. While looking for a MER, each column was checked in order to find its key element if one existed. If found, then its MKE (Maximal Key Element) was found and the MER deduced. The main advantage of SLA algorithm was its shorter updating process, at each task placement or withdrawal.
SLA and Staircase algorithms were quite similar as they only needed a partial scan of the array while refreshing the MERs. Therefore they had lower time complexity. The simulation results showed that Staircase achieved a speedup of 2.5 times in comparison with SLA, as discussed in section 4.2, Chapter 4.

Vertex-List

To the best of our knowledge, Tabero et al. (2004) first used a Vertex-list to manage free areas on a reconfigurable array. As stated previously, Tabero et al. (2006) implemented a one-level looking-ahead scheduling algorithm which used Vertex-list to record the state of the chip. Tabero's works emphasized the fact that Vertex-list based placement approach allowed the algorithm to take into account and therefore minimize chip fragmentation. More details are given below in section 3.9 which presents a few fragmentation-based scheduling algorithms, including Tabero's Vertex-list. It is shown that Vertex-list approach is slow because it is basically MERs-based.

3.8.4 Fitting Strategies

Packing rectangular-shaped hardware tasks on a reconfigurable chip is similar to the 2D bin-packing problem, which is an extension of the classical 1D bin-packing.

1D Placement

One-dimensional bin-packing consists of placing rectangular modules in rows. One example applied to hardware tasks placement on a reconfigurable array is mapped in figure 3.17. Table 3.1 shows parameters of 6 real-time tasks $T_1 ... T_6$ to be placed. As they arrive, the tasks are placed as shown in figure 3.17. A 1D placement assumes that each task spans the entire height of the

device, whatever the real height of the task may be. Consequently, task T_6 is rejected despite the fact that there are enough contiguous space that can accomodate it.

Tasks parameters	T_1	T_2	T_3	T_4	T_5	T_6
a_i : arrival time	1	1	3	3	8	10
e_i : execution time	14	17	18	16	14	12
d_i : deadline	20	20	23	23	23	23
w_i : width of the task	2	1	1	2	1	1
h_i : height of the task	4	2	5	2	2	2

Table 3.1: Tasks to schedule on an FPGA of size 7 X 6

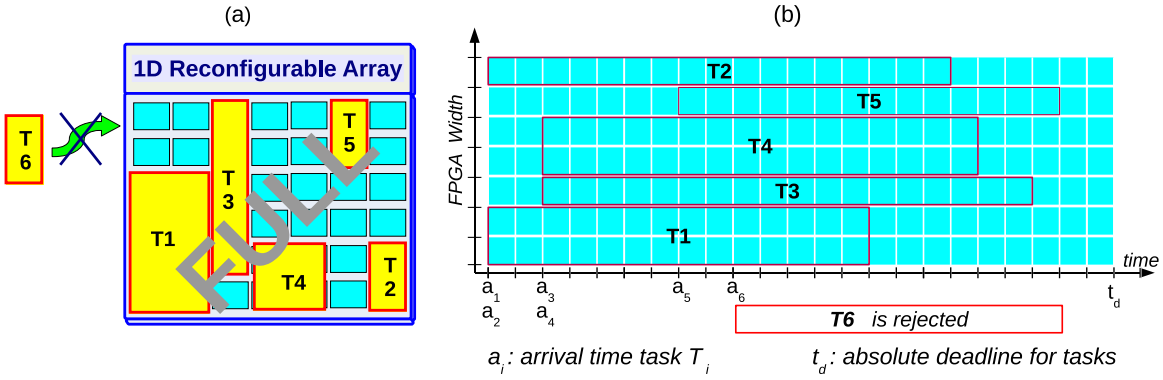


Figure 3.17: Scheduling tasks on a 7 X 6 reconfigurable array using a 1D placement model

1D bin-packing algorithms are used to perform 1D placement. In 1D placement, modules are placed and routed in a vertical manner as depicted in figure 3.17(a). Hence, modules cannot interfere on the X-axis. The placement problem is simplified to the allocation of one interval on the X-axis to each incoming module. Heuristics implementing 1D placement are more simple and hence more suited to online placement. Until recently, because of technological restrictions, FPGAs were hardly supporting 2D placement. Indeed, the devices were only reconfigurable column-wise. Hence, even if 1D and 2D heuristics were developed and simulated, mosts of prototyping works were limited to a 1D scenario. Fortunately, these restrictions are progressively overcome and random parts of FPGAs are independently reconfigurable.

Best Fit (BF) and *First Fit (FF)* are two well-known fitting strategies for bin-packing algo-

rithms (Coffman et al., 1997). The FF algorithm tends to be faster by putting the arriving module in the lowest indexed bin that may accommodate the module. The BF algorithm minimizes the wasted space by choosing the bin that has the smallest room to accommodate the arriving module. Both algorithms require $O(n)$ time for each insertion operation in the worst case, n being the number of bins.

2D Placement

In the 2D bin-packing problem, the rectangular module to be inserted can be placed anywhere on the reconfigurable area (figure 3.18). 2D bin-packing heuristics are used with some restrictions to perform 2D placement of rectangular modules on a reconfigurable chip. As depicted in figures 3.18 and 3.19, the modules to be inserted cannot be rotated and therefore must be positioned with a fixed orientation.

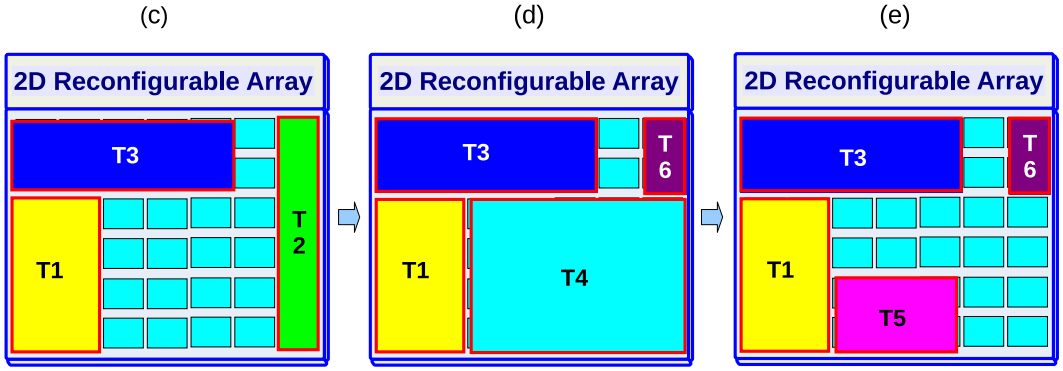


Figure 3.18: 2D placement model of tasks on a 7 X 6 reconfigurable array

3D Placement for scheduling

When considering scheduling, a third dimension has to be added for the time (figure 3.19). Hence the 3D placement is similar to packing rectangles into a container $W \cdot H \cdot D$ where W is the weight, H is the height and D is the depth. In the case of a 3D placement, the depth is replaced by a time axis t. Hence, contrary to container loading, the rectangular modules can only be placed in limited places because some places are lost to time.

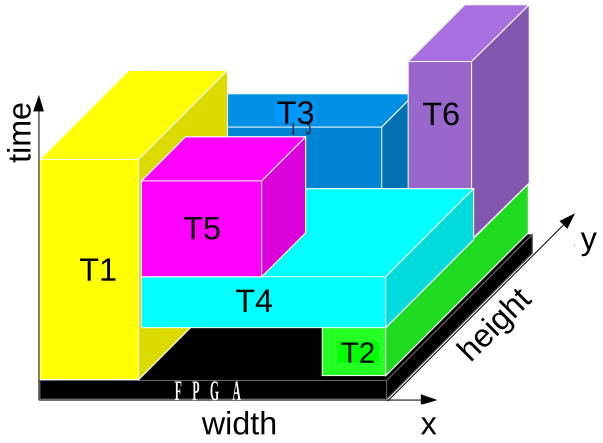


Figure 3.19: 3D view of the 2D placement model illustrated in figure 3.18

3.8.5 Related Work

Bazargan et al. (2000) have presented a fast online placement method for dynamically reconfigurable systems, as well as 3D placement algorithms for statically reconfigurable architectures. The reconfigurable fabric is homogeneous and consists of a 2D array of reconfigurable functional units (RFUs). This work provided a valuable framework for research on online and offline scheduling and placement of tasks on a reconfigurable computing system. Their placement algorithms are divided in the classical two main parts: an empty space manager for partitioning and a search engine and bin-packing rule for tasks insertion/deletion.

(i). The first part (*partitioning manager*) uses both overlapping and nonoverlapping approaches and a binary tree as described and depicted earlier in figure 3.16 page 111. The Bazargan et al. (2000)'s overlapping approach is MER-based and is proposed through the KAMER algorithm. KAMER stands for Keeping All Maximum Empty Rectangles. This first method (KAMERs) takes quadratic space in terms of number of modules on the reconfigurable array. It has to manage $O(n^2)$ empty rectangles for n placed tasks on the array. Indeed, inserting a new task splits empty rectangles into smaller ones while removing a task merges empty rectangles to form bigger ones. This increases the time needed to insert an arriving module on the chip. However, it provides a high quality placement (while combined with a bin-packing algorithm like Best Fit) in spite of the fact that it is slower. Consequently, the KAMERs approach is much more suitable for offline problems.

In the nonoverlapping approach, Bazargan et al. (2000) proposed the Keeping Nonoverlap-

ping Empty Rectangles method. This method induces $O(n)$ complexity algorithms as the number of empty rectangles to manage is linear in terms of the number of placed tasks. Therefore this approach which comes in a variety of forms is not optimal but faster. For example, by giving up slightly on the placement quality, a variant of this approach, combined with the Best Fit (BF) fitting strategy, has a speedup of 16 times compared to the KAMERs (Bazargan et al., 2000). Keeping Nonoverlapping Empty Rectangles approach is consequently more suitable for online problem, even if it increases fragmentation of the chip. The interesting part of this work is the one devoted to online placement. Although, thanks to its highest placement quality, the KAMERs method is used as the baseline for comparison against other placement algorithms, in terms of the quality of placement that is lost to the benefit of the amount of speedup that is gained.

Many nonoverlapping partitioning variants are assessed by Bazargan et al. (2000) in addition to static vertical and horizontal partition pictured in figure 3.13. Shorter Segment (SSEG), Longer Segment (LSEG), Square Empty (LSQR), these are examples of partitioning strategies that dynamically partition vertically or horizontally depending on the size and the shape of the resulting free rectangles. None of them significantly outperforms the other. However, SSEG-BF (SSEG partition strategy coupled with a Best Fit fitting strategy) shows some improvement over other nonoverlapping variants and therefore provides a good trade-off for online placement, when compared to KAMER algorithms.

(ii). The second part (*search engine*) of Bazargan et al. (2000)'s algorithms uses bin-packing fitting strategies as described above (Best Fit, Bottom Left, First Fit, etc.) to select an empty space among those that can accommodate the module whose insertion is requested.

Ahmadinia et al. (2004) proposed a new 1D online dynamic task scheduling algorithm using 1D FPGA partitioning in order to provide a better result than the KAMER and the Keeping Nonoverlapping Empty Rectangles methods presented by Bazargan et al. (2000). Their FPGA area model is homogeneous indeed and is divided into slots (or clusters). The arriving tasks are placed inside one of the slot depending on their completion time. The main idea behind this so-called cluster based algorithm is to avoid fragmentation occurring in the Keeping Nonoverlapping Empty Rectangles method, by freeing contiguous region (removing contiguous tasks) on the FPGA at nearly the same time. Indeed, if the tasks placed in the same slot (or neighborhood) have nearly the same end of execution time, the tasks will be removed at nearly the same time, and a large empty space will be created at a precise location. Hence, even larger arriving tasks could be easily

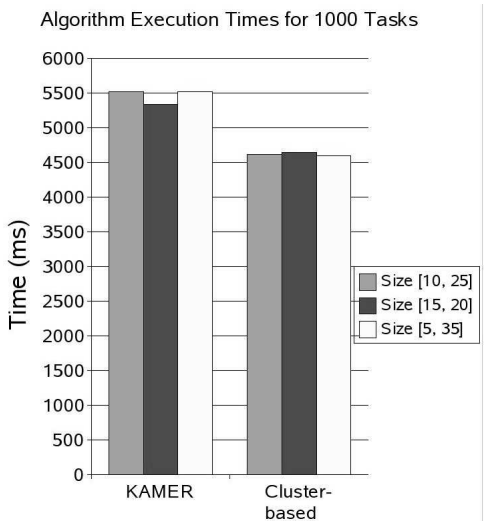


Figure 3.20: Algorithms execution time comparison between KAMER algorithm (Bazargan et al., 2000) and 1D Cluster-based algorithm (Ahmadinia et al., 2004)

placed in the newly created empty space. In addition, a new free space manager is proposed which requires linear memory ($O(n)$), and which runs faster since it doesn't need to divide or merge empty rectangles. In their example, the FPGA is divided into 3 slots.
The results of experiments compared with the KAMER algorithm in terms of how fast they execute on one hand, and of how many tasks get rejected on the other hand. Figure 3.20 shows the comparison result obtained by simulating 1000 tasks with width and height uniformly distributed in three intervals corresponding to the three slots. Their algorithm has an improvement of 15 to 20% compared with the execution time of the KAMER algorithm (Bazargan et al., 2000) while both algorithms have nearly the same percentage of rejected tasks (15.5% for KAMER vs 16.2% for cluster-based algorithm).

Walder et al. (2003) improved the Barzagan's nonoverlapping partitioner proposed in Bazargan et al. (2000). These improvements aimed at limiting the chip fragmentation and therefore decreasing tasks rejection. They are discussed later in this chapter in section 3.9.3, from a fragmentation perspective.
In addition, Walder et al. (2003) proposed a hash matrix as a data structure that represents the state of the reconfigurable array (figure 3.21-(b)). As depicted in figure 3.21-(c), (d) and (e), the matrix stores nonoverlapping rectangles that are available for placement. Each free rectangle

is referenced by a key. A hash function maps a key to the entry of the matrix that holds the information on the free rectangles referenced by the key (see figure 3.21-(b)). Each entry (a, b) in the table consists of a list of free rectangles (of $width \geq a$ and $height \geq b$) and a pointer to the elected rectangle in the list. Hence, the entry stores information about any rectangle capable of accomodating any task $T_{[W,H]}$ of size W x H . A rectangle is elected among others according to a fitting strategy (e.g. best fit, first fit, etc...).

With this structure, a feasible placement is found for an arriving task in a constant time. Hence, the area finder algorithm performs in time complexity $O(1)$. The latter features make the matrix particularly suitable for online placement.

However, the main drawback of a hash matrix is its size which is equal to the size of the reconfigurable array. Therefore, updating the matrix after each task placement or deletion is a time consuming operation. For example, in figure 3.21-(c), the reconfigurable array is empty and is assumed to be a free rectangle R_1. When a task of size 3X5 is placed (see figure 3.21-(d)), R_1 is deleted from the matrix and R_2 and R_3 are inserted. Therefore, the entries have to be updated in order to take into account newly inserted or deleted rectangle(s). After inserting or deleting a task T_i of width w_i and height h_i, the matrix update process could scan and update up to $w_i \cdot h_i$ cells. Despite this long update process, the main advantage of using a hash matrix is that as soon as an area has been found in $O(1)$ time for a task, the latter could be placed and started immediately, and the hash matrix updated later. This feature makes Walder et al. (2003)'s placement method valuable for online real-time systems that require short waiting time.

Using a hash matrix that stores nonoverlapping rectangles affects the scheduling/placement algorithm runtime overheads, but can not affect the chip utilization ratio and the tasks rejection ratio. Indeed, searching for the area that can accommodate a given task either in a list or in hash matrix provide the same result in terms of the selected area. However, the time complexity depends on the data structure.

Roman et al. (2006) also proposed a partition-based management of the empty space. The FPGA model is a homogeneous FPGA, made of a 2D grid of identical basic cells as depicted in Chapter 2, figure 2.6, page 34. The FPGA area is divided into four partitions with different sizes where the tasks will be executed. The size of the partitions is adjusted during run-time according to the profile of the tasks set being processed, in order to adapt it to the variations of tasks profile. Each task T_i is modelled by a rectangle and is expressed by a tuple: $Ti = w_i, h_i, tarr_i, tex_i, tmax_i$ where w_i is the task width, h_i is the height, $tarr_i$, the clock cycle at which it arrives, tex_i the

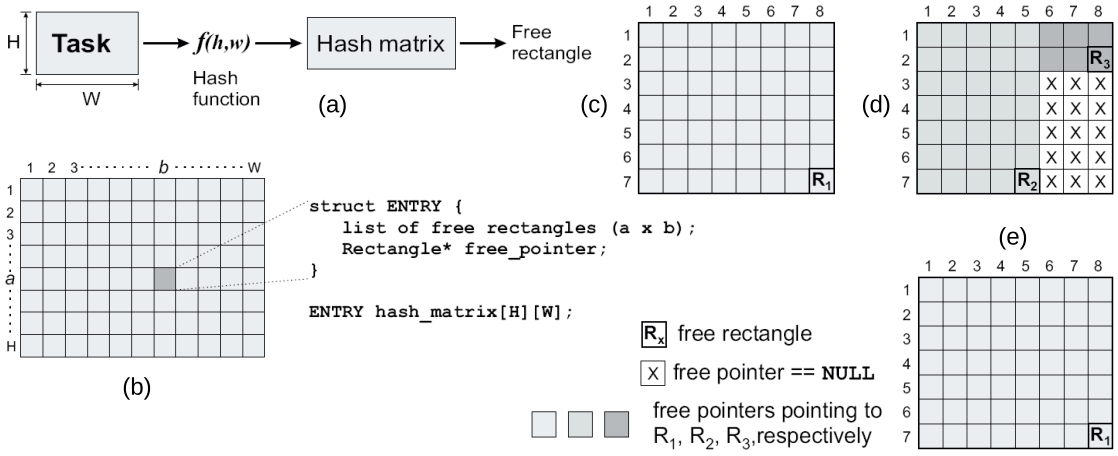


Figure 3.21: The hash matrix approach (a), the hash table (b) rectangle insertion/deletion in the hash matrix (c), (d) and (e) (Walder et al., 2003)

execution time of the task and $tmax_i$ the time needed by the tasks to run to completion. With such a fixed and reduced number of partitions, managing the data structures representing and organizing the FPGA space is far simpler. This leads to a fast algorithm (of constant complexity $O(1)$) particularly suitable for runtime scheduling and allocation of incoming hardware tasks on runtime reconfigurable FPGAs. External fragmentation is also avoided. Their algorithm is $O(1)$ complex and may compete in performance with other common algorithms like First Fit (FF) with exhaustive search, of $O(n^4)$ for the 2D allocation problem (where n is the dimension of square FPGA). Experiments carried out have shown that this is particularly true for task sets with heterogeneous size. Hence, the strength of this work is to have proved that a simple $O(1)$ algorithm is a feasible solution for heterogeneous size tasks scheduling / allocation problem in multitasking on FPGAs. But this approach is not suited to homogeneous size tasks set and its FPGA area model doesn't take into account the ever increasing heterogeneity of new FPGAs.

Considering the heterogeneity of the reconfigurable array

Homogeneous and heterogeneous reconfigurable arrays have been pictured respectively in figures 2.6, page 34 and 2.11, page 43. Many researchers (e.g. Koester et al., 2005, 2006) attempted to improve placement algorithms by taking into account this hardware heterogeneity in order to optimize the resource utilization. Consequently, depending on the logic it uses, a given module (hardware task) will have a few feasible placement positions on the reconfigurable area. For

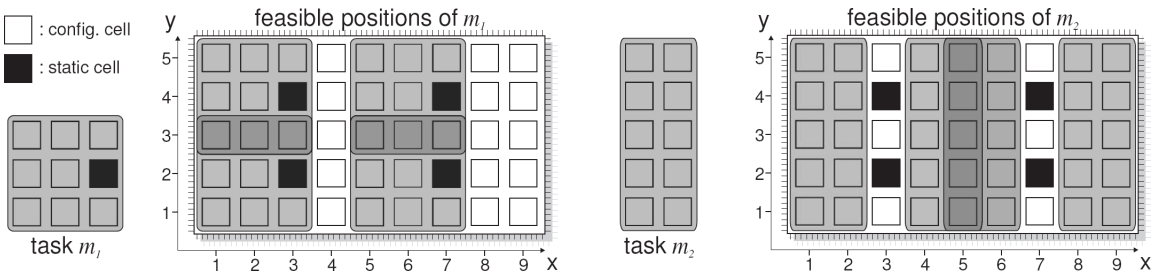


Figure 3.22: Placing tasks m_1 and m_2 on an heterogeneous reconfigurable architecture (Koester et al., 2005)

example, a module using only configurable cells (CLBs) would have a large amount of feasible positions while a module using a great amount of memory would have feasible positions restricted to the positions of the memory blocks.

Koester et al. (2005) proposed a heterogeneous model of reconfigurable hardware. In the model, the reconfigurable area consists of two types of components or cells as shown in figure 3.22 : Configurable cells (e.g. CLBs) which can implement any logic function, and embedded static cells (e.g. multipliers, embedded memories, processors, etc...) which are dedicated high performance IP blocks merged into the FPGA.

In their model pictured in figure 3.22, each task (e.g. tasks m_1 and m_2) is assigned a set of feasible positions on the reconfigurable architecture before run-time. The feasible positions of a hardware task that use static cells (e.g. m_1) depend on the location of the static cells on the reconfigurable architecture. A hardware task that only uses configurable cells (e.g. m_2) can only be placed in positions where static cells are located. According to the two rules, task m_1 has 4 feasible positions; same for task m_2. On one hand, each cell of the reconfigurable fabric has a utilization probability caused by the feasible position of all hardware tasks. This utilization probability is determined before run-time (at design time) since all hardware tasks and the reconfigurable fabric models are known beforehand. On the other hand, for each feasible position of a requested task m, the mean utilization probability of the corresponding cells give a position weight w_{pos} that is used to decide which feasible position to select for the hardware task placement. Depending on whether the position weights w_{pos} are generated before run-time or dynamically updated at run-time according to the current allocated tasks, two placement algorithms were respectively proposed : The *Static Utilization Probability Fit (SUP Fit)* and the *Runtime Utilization Probability Fit (RUP Fit)*. In the light of metrics such as the *average device utilization*, the *hardware task*

rejections and the *average priority* of the rejected tasks, simulation results of SUP Fit and RUP Fit algorithms for 1D and 2D placement have shown some improvements on the standard Best Fit (BF) placement algorithm, presented in Coffman et al. (1997).

Table 3.2 from Koester et al. (2005) summarizes the simulation results of the 1D-placement approach compared with the standard Best Fit (BF) placement algorithm. The SUP Fit algorithm has a low run-time complexity and generates less rejected tasks than the Best Fit algorithm. The RUP Fit algorithm produces the highest device utilization (less fragmentation) and less relative tasks rejection. But it has a high run-time complexity since the position weights have to be updated after each task removal or insertion. Because of technological restrictions of FPGAs at that time, Koester et al. (2005) prototyped an 1D placement approach in their work.

Metrics vs Algo	Best Fit	Sup Fit	Rup Fit	Homogeneous Best Fit
Av. device utilization (%)	49.21	47.94	**50.46**	56.28
Tasks rejection (%)	74	**66**	68	33
Relative tasks rejection (r) (%)	24.43	23.94	**22.92**	14.88
Av. priority rejected tasks	**1.68**	**1.68**	1.73	1.88

Table 3.2: Comparison of the simulation results with the 1D-placement approach (Koester et al., 2005)

3.9 Fragmentation and Related Work

Partial and runtime reconfiguration allows a single FPGA chip to concurrently executed many tasks in a space sharing basis. These tasks of arbitrary sizes are dynamically swapped in and out the chip over time, leading to an increasingly fragmented FPGA as shown in figure 3.25 page 126. Fragmentation arises when the available resources are spread over the FPGA, and consist of noncontiguous small areas. This could prevent tasks from being placed even if there is enough available resources to accommodate them. Indeed, a task could only fit in contiguous resources that cover an area at least as bigger as the task area.

In an offline placement on partially reconfigurable FPGAs, as the sequence of tasks to be executed are known in advance, greedy algorithms or heuristics could be found to determine

the optimal fitting strategy that minimizes the fragmentation. For example, highest complexity strategies (e.g. a MER based defragmentation combined with a cells scan based defragmentation strategy as illustrated in figure 3.23) could be afforded.

In an online context, since the sequence of modules to be loaded on the reconfigurable area is not known beforehand, insertion and deletion of modules leads to the fragmentation of the available space. In order to avoid task rejection because of this fragmentation, there is a need of low complexity algorithms or heuristics that defragment the FPGA at runtime. Depending on the way fragmentation is measured and on the way free areas are managed, there are different way of coping with the chip fragmentation problem, leading to different results and complexity as detailed below and mapped in figure 3.23:

(i). *The nonoverlapping rectangles approach*

As previously stated, available areas are kept as nonoverlapping rectangles. While placing a task, the placer chooses among all the rectangles which could fit the task, the rectangle which minimizes the fragmentation (e.g. Best Fit or more complicated heuristics). Hence, as for placement heuristics, fragmentation heuristics are of lower complexity thanks to the limited number of rectangles, but are less efficient.

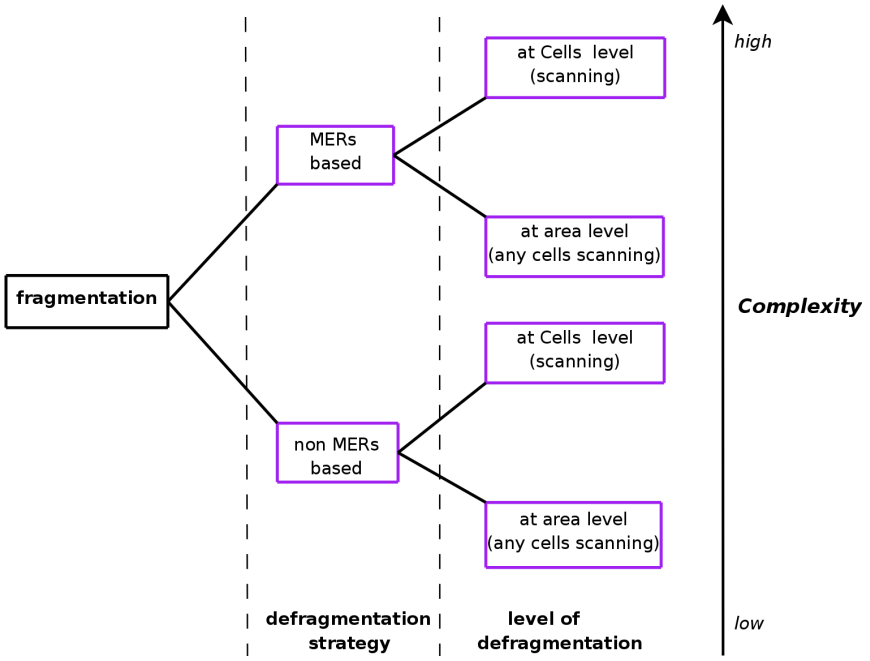


Figure 3.23: Defragmentation strategies: complexity grows with performance

(ii). *The MER (Maximum Empty Rectangle) approach*

Managing nonoverlapping rectangles leads to more fragmentation, as any single of such rectangles is included in a MER, if it is not a MER itself. As already stated, the advantage of managing MER (Maximum Empty Rectangles) is that placement algorithms or heuristics always find a placement solution if one exists. This means that any single candidate area is identified, the best candidate which optimizes a given criteria (e.g. minimizes the final fragmentation in the present case) could also be identified, leading to a better placement quality. A way of reducing the chip fragmentation is to deal with MERs while placing a task. Hence, one could choose the MER which minimizes the fragmentation. This generally means choosing among the MERs which could fit the task, the one which contributes the most to the final fragmentation of the chip. Consequently, optimal defragmentation heuristics are MER-based. However, the runtime overhead is very high because of greedy scanning processes preformed while updating the list of MERs at each task placement or withdrawal.

(iii). *The Cell level approach*

The two approaches (MER and nonoverlapping) presented above directly derive from the way free areas are managed. The trend of the resulting complexity of fragmentation heuristics are quite similar to the complexity of placement heuristics in the two cases. However, the final complexity of determining the fragmentation also depends on whether the contribution of each free area to the final fragmentation results in the contribution of each cell in the area or not. If the fragmentation is determined at cell level, there is a need to scan the complete cells in the area. A cell by cell scanning process remain the only way to obtain optimal results in terms of minimum fragmentation, but lead to greedy computing operations with high runtime overheads. A fragmentation metric combining the MER approach with a cell level fragmentation is definitely optimal (e.g. Cui and Deng, 2007), but the greediest solution in terms of complexity and runtime overheads.

There are two types of fragmentation, *internal* and *external*.

3.9.1 Internal and Intra-task Fragmentations

In 1D placement, an *internal fragmentation* occurs when a task does not utilize the full height of the reconfigurable device area (see pictures on the left of figures 3.24 and 3.25). This internal fragmentation leads to a lower FPGA utilization ratio in 1D placement compared to 2D. One way of minimizing internal fragmentation is to maximize the height of hardware tasks during its design.

By doing so, some resources could be gained (saved) as shown in the right of figure 3.24. Indeed, tasks T_1 and T_1' are similar with respect to total area and functionality but are of different shape. T_1' occupies the full height of the device for a smaller width, and therefore produces less internal fragmentation than T_1. Such an improvement is obtained by using appropriate shape constraints in place and route tools at design time.

In addition, because of the fact that modules are rectangular-shaped, there are *intra-task fragmentations*. Indeed, an intra-task fragmentation is a loss of resources that arises when the hardware task is assumed to be rectangular. As pictured in figure 3.24, in the rectangle bounding all the resources needed by task T_1 or T_2, there are some resources that remain unused and that are consequently lost. However, as rectangular tasks are much more easy to manipulate by placement algorithms, this assumption is made by almost all studies on hardware task scheduling.

3.9.2 External Fragmentation

In figure 3.25 (b), there are currently 4 modules spread on the reconfigurable array, but almost the half of the array is still free. However, because of its shape, the incoming module T_5 cannot be inserted even though its size is smaller than the total remaining free space. This fragmentation of the free space is known as external fragmentation. It increases the tasks rejection ratio and decreases the chip utilization ratio because of the waste of resources.

3.9.3 Related Work

The main challenge of placement heuristics is to manage the free space and schedule arriving tasks so that the fragmentation is avoided. This is quite difficult to achieve in an online scenario.

As stated earlier, offline algorithms or heuristics are very efficient but slow, while online placement heuristics are faster with less efficiency. In some systems with some recurring idle times (e.g. in an automotive electronic system during the night when the car is parked), an offline placement component can optimally relocate the modules on the FPGA, in order to defragment the available free space during the idle time. Unused or non-critical modules could be interrupted and relocated in order to free the maximum contiguous space. This offline defragmentation eases the job for the online placement component while the system is in use. Hence, one can overcome fragmentation problems in some specific application by combining online placement with an offline defragmentation. Veen et al. (2005) present such a strategy. Their online placer has an offline component called the defragmenter which performs an offline relocation of the currently placed

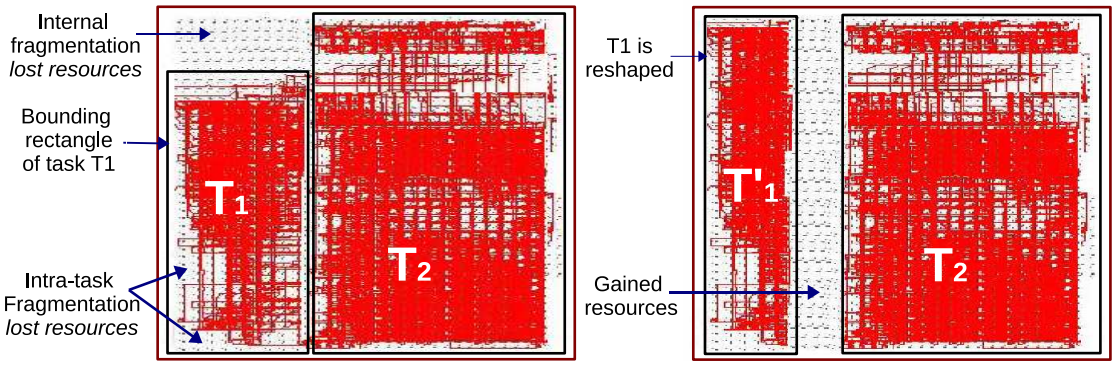


Figure 3.24: Intra-task and internal fragmentation

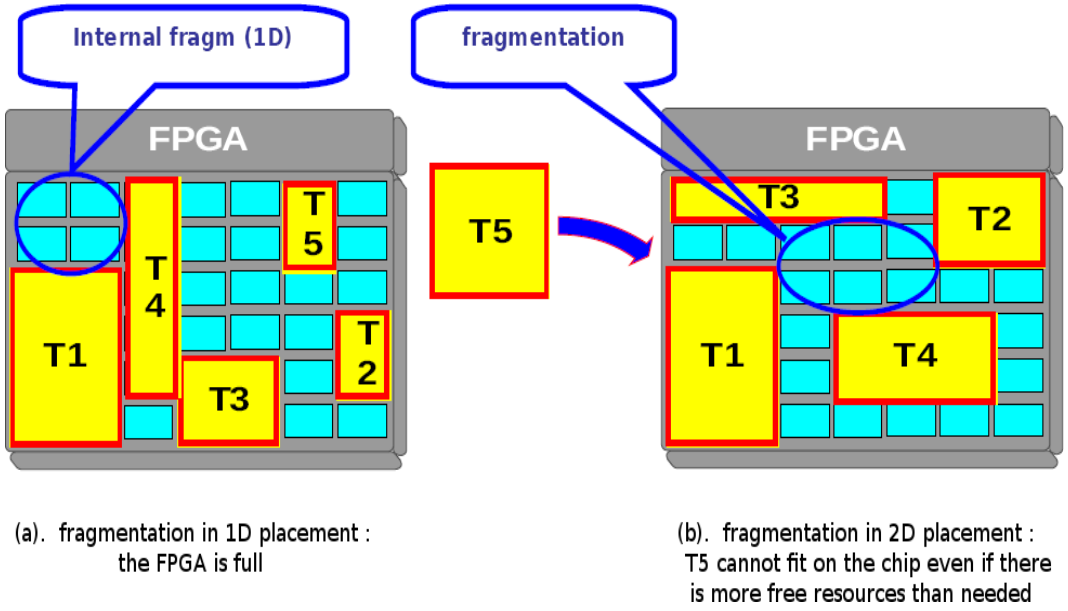


Figure 3.25: A fragmented FPGA: the free space available on the chip is sufficient to insert the arriving task, but its shape doesn't allow it.

tasks with the objective to maximize the areas of contiguous free resources. This defragmentation eases the online tasks placement process which is done at runtime. This approach is extremely useful for FPGA in which partial reconfiguration can only be performed columnwise (e.g. Xilinx Virtex II Pro FPGAs). Indeed, in such an FPGA, while performing a 2D placement, reconfiguring (or placing) a module on the FPGA affects all the modules interfering columnwise. If defragmented online, one has to reconfigure all the interfering modules, which is not desirable at runtime.

The KAMER algorithm used in Bazargan et al. (2000) and combined with various fitting strategies was also aiming at minimizing the fragmentation by providing the optimal fitting solution. Hence, if there are more than one solution, one can choose the fitting strategy that optimizes (minimizes) the fragmentation (e.g. best-fit). This optimality comes at the cost of algorithm complexity as explained earlier in this chapter. Therefore, such a MER-based areas management is difficult to be combined with defragmentation strategies such as partial rearrangement approach (O. Diessel and Schmidt, 2000) discussed below.

However as noticed earlier, the adjacency graph (or binary tree, Bazargan et al., 2000) avoids an unlimited fragmentation by retrieving previous states of the array, as shown in figure 3.26. Overlapping or nonoverlapping areas are inserted in or deleted from the binary tree at tasks placement or deletion. Figure 3.26 depicts the case of nonoverlapping rectangles. It shows how the area of reconfigurable array is fragmented during tasks placement and restored at tasks ending.

When an available area is selected for task placement, if the task is smaller than the area, the remaining area is divided in two nonoverlapping rectangles, bounding the time complexity of inserting a task to $O(n)$. As illustrated in figure 3.26-(a) rectangle A_0 is chosen to accommodate the task T_1. Two nonoverlapping children rectangles A_1 and A_2 are then generated, resulting from a vertical split of the remaining area (figure 3.26(b)). As tasks T_1 and T_2 are successively placed (corresponding to states $(a) \rightarrow (b) \rightarrow (c)$ of areas splitting process...), the tree is updated accordingly (corresponding to states $(a') \rightarrow (b') \rightarrow (c')$ of the tree update process). Assuming that tasks T_1 and T_2 successively finished (which is not always true), the original area A_0 is rebuilt along the reverse process (states $(c) \rightarrow (b) \rightarrow (a)$ of bigger areas recovering process, states $(c') \rightarrow (b') \rightarrow (a')$ of the tree update process). At any time, the currently free rectangles are the leaves of the tree. Storing successive states of the array and restoring them eases the process of rebuilding areas which have been split during a task placement.

However, no matter which split is used (vertical, horizontal, overlapping, etc.), a situation may arise where a task is rejected because the contiguous space which could fit it does not belong to the same rectangle or sub-graph. Indeed, the tree intrinsically produces a fragmentation problem that is lower for overlapping rectangles. The tree is non optimal as a task could be rejected because of this fragmentation problem. The main advantage of the tree approach is its automatic bigger areas restoration and its algorithmic complexity.

O. Diessel and Schmidt (2000) have proposed a quad-tree structure which store the information of the available resources on the FPGA. Three methods are studied (local repacking, ordered

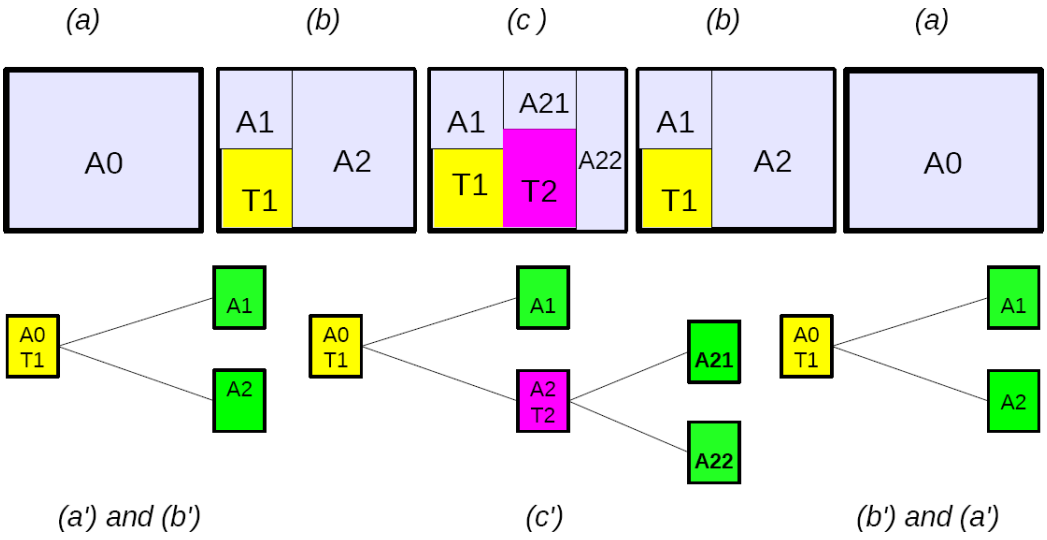


Figure 3.26: Bazargan's adjacency graph: bigger rectangles restoration process (Bazargan et al., 2000)

compaction and genetic algorithm) for placing tasks on a partially and dynamically reconfigurable FPGA by a so-called partial rearrangement process. Partial rearrangement aims at rearranging some of the already executing tasks in order to fit an arriving task which normally couldn't fit immediately. Partial rearrangement hence significantly reduces queuing delays as tasks are placed as soon as possible and are completed earlier, freeing the place for other incoming or waiting tasks. The proposed three defragmentation methods are of different but still high complexity. The genetic algorithm approach is more efficient in finding fitting areas at the cost of complex rearrangements, but is worthy only if the reconfiguration delay (time needed to configure the task) of the task is small compared to its execution time. Hence, there is more time for processing complex rearrangements without causing execution delays. The two other approaches (local repacking, ordered compaction) are less complex, and then suit to tasks with shorter execution time and longer reconfiguration delays. The computation time of the three different methods are not included in their study, as they assumed that these computations might be executed in the background during inter-tasks arrival period. In addition, these methods also assumed hardware tasks preemption, which is not that obvious in an online real time context.

Contrary to O. Diessel and Schmidt (2000) who coped with preemptive tasks, Walder and Platzner (2002) presented a transformation method (Footprint Transform, figure 3.27) for non-

preemptive multitasking on FPGAs. What is new in their case is that their task model also consider coarse granularity tasks of non rectangular shape. However, each task consist of a set of rectangular tasks denoted as subtasks (e.g. figure 3.27). Hierarchically, as shown in figure 3.27, S_1, S_2 and S_3 are rectangular subtasks constituting coarse grain task T. Relative positions of subtasks can be modified during allocation (footprint transforms), in order to find a suitable shape to task T (shape 1, 2, 3, etc. in figure 3.27) which fits in the available free area. Footprint transform is performed on a task if its placement fails at the first attempt. Footprint transform consequently reduces queuing delays as tasks are placed as soon as possible.

Also presented are *first-fit* and *best-fit* placement techniques. Among a list of positions in the free area that may accommodate the task to be placed, best-fit algorithm selects the best candidate according to a given criteria. The selection criteria in the present case is the fragmentation of the residual area. Indeed, this fragmentation is to be minimized. The residual area here is the remaining free area on the array after placing the current task. Accordingly, Walder and Platzner (2002) introduced a new fragmentation metric denoted as *fragmentation grade* and expressed by equation 3.14.

$$F = 1 - \frac{\sqrt{\sum_i (n_i \cdot {a_i}^2)}}{\sum_i (n_i \cdot a_i)} \tag{3.14}$$

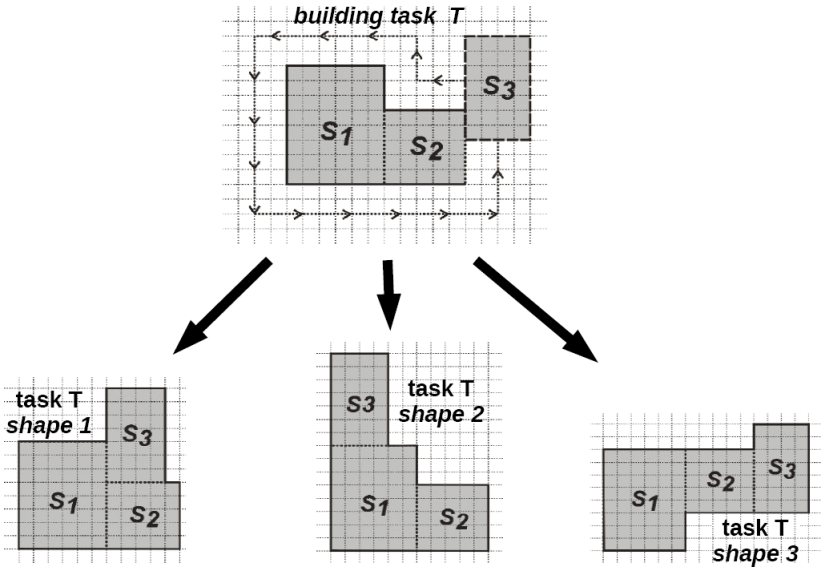


Figure 3.27: Footprint Transform (Walder and Platzner, 2002)

Best fit algorithm first determines as many fragmentation grade F (of the residual area) as possible rectangles or positions where the task could be placed. It then assigns to the task the

position with the minimum F. As F is calculated for any single possible fitting position of the task T on the free part of the FPGA array, the bigger T (and/or the smaller the available area on the array), the less possible fitting positions. Equally, the lower F, the more the future task likely to fit. However, determining fragmentation grade remains a computationally heavy iterative process as for a given allocation, all the possible rectangles of the residual area are identified and classified according to their size. The resulting histogram of free areas consists of i classes of n_i rectangles of size a_i, and the deriving fragmentation grade for one allocation is given by the equation 3.14. Their results are mitigated as footprint transform in combination with first-fit strategy sometimes outperforms best-fit in terms of tasks set total execution time (scheduler makespan), sometimes not. However these results are obtained with only 25% of the tasks set being footprint transformed. Such results are interesting as they show how different combinations could lead to different algorithm complexity *vs* scheduling quality trade-offs. However, complexity and runtime overhead of algorithms are not assessed in order to measure their suitability to online scheduling.

In a later work, Walder et al. (2003) proposed an improvement of the Bazargan's partitioner presented above (Bazargan et al., 2000) in order to limit the fragmentation and improve placement quality. They gave up the idea of dealing with non rectangular task and the novelty is to delay as much as possible the split decision while placing a new task in a bigger rectangle. Walder et al. (2003) relied on the same binary tree structure mentioned by Bazargan et al. (2000) and presented above in section 3.8.3 to manage the areas. Hence, no matter which heuristic is used by Bazargan et al. (2000) to decide how to split the hosting rectangle (vertical, horizontal, etc...), a situation could arise where the next task could not fit in any of the resulting rectangles because of the split previously done. In order to minimize such cases, one solution is to delay the split decision until the arrival of the next task to be placed, and then to perform the right split accordingly. For example, in their so-called on-the-fly partitioning (OTF), Walder et al. (2003) manage overlapping rectangles in innovative ways. Hence, an overlapping rectangle is resized only if it overlaps with the just placed task. Generated children rectangles are kept overlapping as much as necessary. Two other variants of partitioning algorithms are assessed. Walder et al. (2003)'s partitioning approaches outperform Bazargan et al. (2000)'s partitioner by up to 70% in terms of average waiting time of tasks.

In order to limit the fragmentation of the reconfigurable array, Ahmadinia et al. (2004) proposed an algorithm that uses two horizontal lines to manage free spaces on the array. Instead of maintaining a list of empty rectangles, they managed two horizontal lines, one above and one

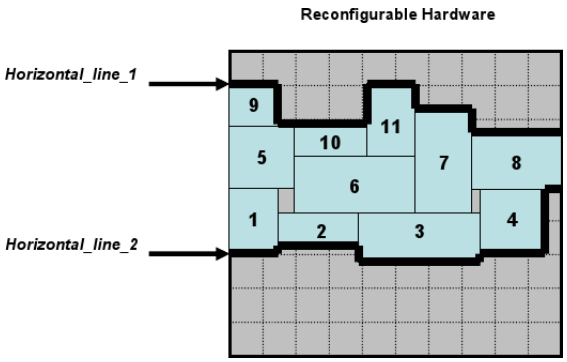


Figure 3.28: Using horizontal line to manage free space Ahmadinia et al. (2004)

under the already running tasks as depicted in figure 3.28. As any significant free area shouldn't be between the two horizontal lines, newly arriving tasks could therefore be placed either on the top of *Horizontal_line_1* or at the bottom of *Horizontal_line_2* and the corresponding line is modified accordingly. The overall idea behind this approach is to put beside each other tasks with nearly the same finishing time. Hence, contiguous spaces are free at nearly the same time, reducing the fragmentation. The resulting empty space is then more likely to fit future and even larger tasks. Furthermore the principle is extended to a clustered approach, where the reconfigurable array is split in a number of 1D clusters, each cluster accommodating tasks with similarities in terms of finishing time.
What is interesting in this algorithm (which doesn't manage MERs) is its comparison to MER approach. Indeed, as already stated, managing MER certainly lead to a better placement quality in terms of tasks rejection ratio, but at the cost of high complexity and algorithm runtime overhead. Results show that their algorithm takes slightly 18% less time to compute than the KAMER algorithm in Bazargan et al. (2000) for almost the same tasks rejection ratio (resp. 16.2% and 15.5%).

In Handa and Vemuri (2004a) and Handa and Vemuri (2004b) are proposed MER based approaches for avoiding fragmentation while scheduling non-preemptive tasks. In Handa and Vemuri (2004b), free areas are kept as list of MERs. In order to avoid areas fragmentation, scheduling decisions are deferred as much as possible to accommodate dynamically changing task priorities. Results show that delaying scheduling decisions leads to a reduced areas fragmentation (better area utilization) only when tasks are not data-dependent upon each other (*out of order processing*, unlike *in order processing*) and can be executed in any order. In Handa and Vemuri (2004a) frag-

mentation is calculated at cell level. In their approach, the total fragmentation contribution of each cell in the MER $TFCC_C$ is obtained by summing its contribution in the horizontal dimension (FCC_{Cx} on x axis) and in the vertical dimension (FCC_{Cy} on y axis) as shown in equation 3.15. The total fragmentation of a MER TF is the average value of $TFCC_C$ for all the cells in the MER, as given by equation 3.16

$$\text{horizontal dimension : } FCC_{Cx} = \begin{cases} 1 - \frac{v_x}{\overline{L_x}-1} & \text{if } v_x \leq \overline{L}_x \\ 0 & \text{otherwise} \end{cases}$$

$$\text{vertical dimension : } FCC_{Cy} = \begin{cases} 1 - \frac{v_y}{\overline{L_y}-1} & \text{if } v_y \leq \overline{L}_y \\ 0 & \text{otherwise} \end{cases}$$

$$\forall_{cell\ C}\ , \qquad TFCC_C = FCC_{Cx} + FCC_{Cy} \tag{3.15}$$

$$\forall_{MERs\ of\ k\ cells}\ , \qquad TF = \frac{1}{k} \cdot \sum_{i=1}^{k} TFCC_C(i) \tag{3.16}$$

L_x (resp. L_y) represents twice the average width (resp. height) of the tasks currently placed on the array, and v_x (resp. v_y) is the number of contiguous empty cells horizontally (vertically) aligned with the involved cell. FCC_{Cx} and FCC_{Cy} express the fact that the more are empty cells surrounding a cell in the MER and the smaller is the average size of the tasks currently placed, the lesser is $TFCC_C$.During a placement process, if there are many MERs which can fit the task, the MER with the maximum TF is chosen. Hence the MER with the highest fragmentation impact on the FPGA is preferably destroyed (by placing a task on it), preserving other MERs with lesser contribution to the fragmentation. The idea is to consequently reduce the overall FPGA fragmentation. Once the hosting MER is found, the task is placed in one of its four corners with the highest TF for a rectangle of the size of the task.
At all levels, this algorithm is based on a scanning approach as on one hand it deals with MERs (which is computationally greedy), and on the other hand all the cells of MERs are scanned and a huge amount of time is spendind computing the right corner where to place the task.

Tabero et al. (2004) use a Vertex List structure to keep the track of the available free area. They propose a fragmentation-based heuristics (among other criteria-based heuristics) that prevents the proliferation of holes in the FPGA. Hence, before assigning one position among possible candidates, the fragmentation produced by each candidate area is first evaluated. Hence, the area that minimizes the fragmentation is chosen. This approach is nearly similar to the one

used by Walder and Platzner (2002), even if they use different fragmentation metrics. Indeed, Tabero et al. (2004) first calculate the fragmentation produced by each MER, which is obtained by summing the contribution of each cell of the MER. Obviously, this slows down the placement process.

Koester et al. (2006) adopted a *defragmentation-by-modules-relocation* approach to deal with continuous fragmentation of the reconfigurable array over time. As hardware tasks could be placed and removed at runtime[14], an increasing fragmentation of the FPGA prevents next tasks from being placed. Their solution is to relocate at runtime the currently placed tasks for being able to place the requested task. Their runtime defragmentation algorithm aims at implementing such an approach. Prototyped on a dynamically and partially reconfigurable Xilinx Virtex-II FPGAs, their algorithm applied to the 1D placement shows some improvement of placement quality. For example, the total execution time of tasks set is reduced to 87.1% in the best case, compared to another scheduling algorithm without any runtime defragmentation. However, using this *deframentatio-by-relocation* approach is worthy only if tasks reconfiguration time is negligible compared to tasks execution time. In addition, as noticed earlier, their approach takes into account the heterogeneity of the FPGAs by identifying feasible positions for tasks, according to their types of resources (logic cells or embedded memory).

Cui and Deng (2007) proposed an online task placement algorithm which aimed at minimizing fragmentation on partially reconfigurable FPGAs. They introduced a 2D area fragmentation metric that takes into account the probability distribution of sizes (width and height) of future tasks arrival. Hence, in their assumption, dedicated embedded applications are targetted and consequently, task arrival patterns are predictable. Cui and Deng (2007) is also a MER based approach which uses a Scan Line Algorithm (denoted as SLA and studied in a previous work in Cui and Deng (2007)) to determine the set of MERs and maintain a fragmentation matrix (FM). As in Handa and Vemuri (2004a), the contribution to the fragmentation is calculated at cell level and then at MER level. However, apart from assuming that the probability distribution of sizes of arriving tasks are known, what is also new is that they introduced a Time-Averaged Area Fragmentation (TAAF) metric. TAAF metric is used to evaluate the fragmentation of a MER over present and future time. Hence, MERs contribution to fragmentation are not compared only at present time, but averaged over a time interval $[t_{present}, t_{future}]$. Indeed, in order to compute

[14] at restricted or feasible positions as presented in Koester et al. (2005)

the TAAF, one has to mimic future events (end of some tasks and beginning of planned task, modification on MERs along with cell fragmentations) which is computationally intense. As for almost all similar works coping with the fragmentation problem, there is no algorithm running time measurement. In Cui and Deng (2007), the running time of algorithms that update the list of MERs is measured on a Solaris workstation. Combining a MERs-based area management with a cell-based defragmentation strategy and a one-level looking-ahead-based scheduling approach leads to a time-consuming placement not suitable for an online context on an embedded processor.

3.10 Conclusion of the Chapter

This chapter reviewed the background literature relating to the research. The chapter discussed the real-time scheduling problem for uniprocessor and multiprocessor systems. From there, the chapter drew similarities between this well known scheduling problem and the problem of scheduling real-time hardware tasks on dynamically and partially reconfigurable hardware devices. The main advantage of this approach was to see how the models used in microprocessor scheduling could be transposed in reconfigurable hardware device scheduling. Afterwards, the chapter presented different works on hardware tasks scheduling through the two mains scheduling strategies denoted as *looking-ahead* and *without-looking-ahead* respectively. The review also discussed the underlying placement problem which is specific to reconfigurable hardware devices scheduling. The fragmentation problem was presented separately from the placement problem in order to point out its importance in tasks placement. This separation led sometimes to some redundancy as the same related work could be presented from a scheduling/placement perspective and from a fragmentation perspective.
A summary and a classification of related work on scheduling and placement for reconfigurable hardware devices is presented in table 7.1, page 246, Appendix A.

The next chapter will present the methodology along with the models and the metrics proposed in this thesis in order to cope with the problem of scheduling online real-time hardware tasks on dynamically and partially reconfigurable hardware devices.

In this thesis, most of the scheduling algorithms proposed in Chapter 5 and simulated in Chapter 6 use placement strategies that rely on the hash matrix (Walder et al., 2003) and the tree (Bazargan et al., 2000) discussed earlier in this chapter (respectively on pages 118 and 110). If

not so, it is clearly indicated. This does not affect the comparative study of scheduling algorithms, as two algorithms can be compared only if they all rely on the same placement strategy.

Chapter 4

Proposed Methodology, Models and Metrics

4.1 Introduction

This thesis relies on a formalism well-known in real-time scheduling modeling. According to this formalism, discrete time is measurable, and centric as it characterizes each element in the system. In the previous chapter, the formalism was customized whenever possible to take into account the specificity of systems that include dynamically reconfigurable parts. Some models and metrics for microprocessor scheduling have been already introduced in the previous chapter. However, the latter mainly discussed real-time scheduling problems in general, and exhaustively presented related work on scheduling and placement targetting reconfigurable hardware devices.

A part of this chapter deals with models and metrics that are meaningful for hardware tasks scheduling on reconfigurable hardware devices. Application and hardware tasks models are presented, followed by a resources model, then a scheduler model along with the underlying placer model. As stated previously, the thesis mainly considers the problem of scheduling a set of *aperiodic real-time tasks* on a reconfigurable array. The proposed models and the corresponding metrics are derived from multiprocessor platforms models described in the previous chapter.

Hereinafter is the proposed methodology that results from the understanding of the literature review done in the previous chapter, the primary objective of this thesis being: *"the online scheduling of real-time hardware tasks on partially and dynamically reconfigurable hardware devices"*.

4.2 Methodology

4.2.1 Introduction

The problem of scheduling on-line real-time tasks for multitasking or hardware virtualization on a reconfigurable hardware device is coupled with a placement problem. Real-time scheduling aims to define how to schedule elementary tasks of an application on a limited computing resource in order to complete the application within a given time frame. The placement aims to use area management algorithms and heuristics in order to efficiently allocate the reconfigurable array to tasks. Therefore, scheduling and placement are quite closely linked.
The proposed methodology first took into account the targetted architectures which are embedded reconfigurable systems submitted to many constraints, including online real-time issues. As a whole, online scheduling and placement heuristics should be light enough to run on an embedded processor and to take fast placement decisions as tasks arrive. Consequently, fast scheduling/placement heuristics may be preferred to those that provide high quality placement at the cost of high runtime overhead. Different scheduling and placement approaches were discussed in the previous chapter and summarized in tables 7.1 and 7.2, pages 246 and 247.
The following methodology will aim to provide a quick guidance that leads to acceptable trade-offs between runtime overhead and placement quality, and which are likely to enable online scheduling of real-time hardware tasks on a runtime and partially reconfigurable device.

4.2.2 Proposed Methodology

As scheduling tasks on a reconfigurable platform brings an additional placement problem, the methodology consists of assessing the runtime overhead of placement algorithms in order to see which are suitable for online real-time scheduling. Indeed in all previously cited work (Chapter 3) on reconfigurable hardware scheduling and placement, algorithm simulations are performed on desktop computers. Desktop computers are usually clocked at more than 1GHz, are power hungry, and host sophisticated cache organizations (e.g. 1MB L2 cache) and memory management units. Furthermore, unlike embedded processors, they concurrently run numerous services and applications, most devoted to ergonomics and human machine interaction. These factors make runtime overheads very difficult to measure accurately (e.g. at cycle level). The obtained results may not reflect what would have been the real runtime overhead on an embedded processor running these scheduling and placement algorithms. In general embedded processors are rarely clocked at

frequencies higher than 250 MHz.

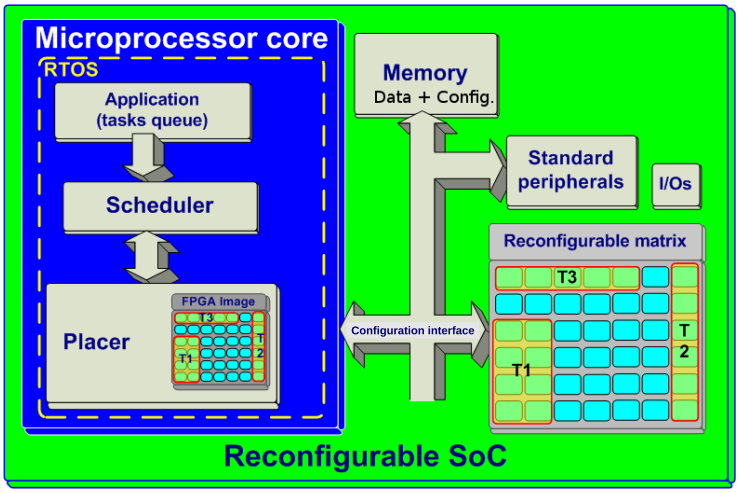


Figure 4.1: A simple architecture of a reconfigurable SoC

The first step in the methodology was to implement MERs-based placement algorithms on an embedded platform in order to see the real runtime overhead of such placement algorithms. This step relied on a simple model of an embedded reconfigurable system-on-chip, depicted in figure 4.1. This simple model consists of a CPU-based processor and a reconfigurable fabric. The CPU-based processor runs scheduling and placement algorithms that manage the reconfigurable part of the chip. Therefore, scheduling and placement algorithms were mapped as software programs written in C language. As illustrated in figure 4.1, the placer maintains a data structure that reflects the state of the reconfigurable matrix. When the placer finds a place to fit a task T_i, a loader (also run by the CPU and not explicitly represented here) loads the corresponding bitstream from the memory and partially configures the reconfigurable matrix, through a configuration interface. However, configuration time overhead is technology dependent as shown later in figure 4.6. Here, the focus was on assessing the runtime overheads of scheduling and placement strategies.
The next step was to perform timing measurements on the embedded CPU-based processor when running placement algorithms. The aim of the experiments was to:

1. see what the timing limitations are for systems that could be online real-time scheduled by an embedded processor running a MERs-based placer, and
2. identify which combinations of scheduling and placement algorithms are likely to be used in an online real-time context.

The next section presents these experiments, and the obtained results, along with subsequent conclusions.

4.2.3 Running two MERs-based Algorithms on an Embedded Processor

One important factor is to know the time taken by the scheduler to place or remove a task on the FPGA structure. In a MERs-based algorithm, what is time consuming is the process of scanning the array in order to find all the MERs that are within it. The update is done at each placement or removal.

1. ***A basic Scheduler/Placer***

   A basic scheduler that manages two lists of tasks has been implemented : a running tasks list and a waiting tasks list. The scheduler is invoked by two events : either when a task has just arrived (figure 4.2-a) or when a task has just completed (figure 4.3-c).

   - In the event of task(s) release ; as shown in figure 4.2-a, when a task arrives, the scheduler calls the placer (figure 4.2-b) that checks whether there is an MER to accommodate the task. If there is one, it is assigned to the task. The placement is successful. The placer updates the list of MERs (figure 4.2-b) and the task is added to the running tasks list (figure 4.2-a). However, if there is no MER that accommodate the task and if the task can still meet its deadline, the task is added to the waiting tasks list for further attempts. Otherwise the task is rejected.
   - In the event of task(s) termination ; as depicted in figure 4.3-c, when a task ends, the scheduler calls the placer (figure 4.3-d) that removes the task from the array and that updates the list of MERs. Afterwards, the scheduler (figure 4.3-c) tries to place as many tasks as possible from the waiting list, and updates the list of MERs accordingly after each successful placement.

   This scheduling algorithm is quite simple but sufficient to assess the efficiency of MERs-based free areas management. The scheduler maintains two lists of tasks: the running list and the waiting list. The tasks in the waiting list are sorted according to their arrival time. If an area is freed, the algorithm attempts to place the tasks in the waiting list, beginning from the heading task. Hence, a free area is assigned to a task in the list only if the area cannot fit any of its preceding tasks (that normally arrived earlier). The scheduling algorithm is detailed in the next chapter where it is denoted as *basic scheduling*. Following

the same principle, other priority policies can be applied (e.g. EDF, LLF, etc.).

2. ***Two MERs-based algorithms***

   The *SLA - Scan Line Algorithm* (Cui and Deng, 2007) and *Staircase* algorithm (Handa and Vemuri, 2004c) have been implemented. SLA and Staircase algorithms are improved versions of Bazargan et al. (2000)'s MERs-based algorithms. They are slightly similar in algorithmic complexity. The process of placing or removing a task on the array induces a time consuming operation: the update of the list MERs as highlighted respectively in figures 4.2-b and 4.3-d. The average time taken by the two algorithms to perform the update operation were measured. Staircase scans an average of 15% of the reconfigurable array in order to update the list of MERs on the chip after a task placement or removal. Therefore, it runs faster than SLA, as confirmed in the final results shown in figure 4.4.

3. ***Design environment and tools***

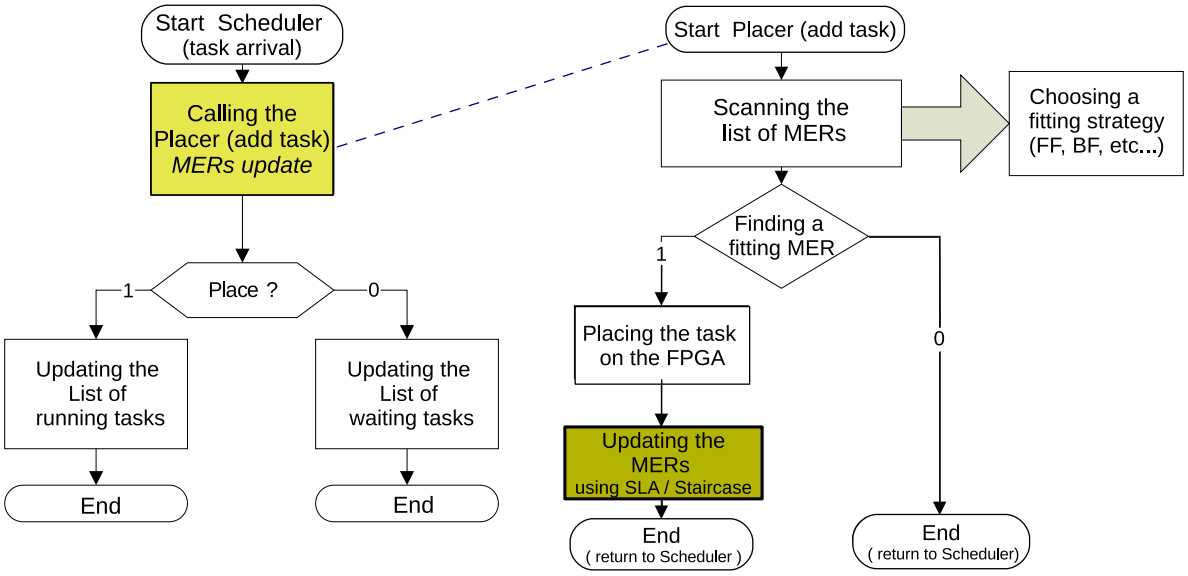
   In order to carry out accurate timing measurements on an embedded platform, the two algorithms were implemented in C language, and then cross-compiled to target an embedded processor, the Xilinx MicroBlaze soft core. The soft core along with a timer (for timing measurement purposes) were instantiated on a Xilinx Spartan 3E FPGA hosted by the Spartan 3E Starter board.

   Both hardware and software parts of the design were designed using the Xilinx EDK development design environment. 1M bytes of external memory storing the program, data and stack, was attached to the embedded processor. The heap and the stack were sized to 12k bytes each. These sizes were chosen according to the size of the reconfigurable array and the number of tasks to be placed.

4. ***Simulation parameters and results***

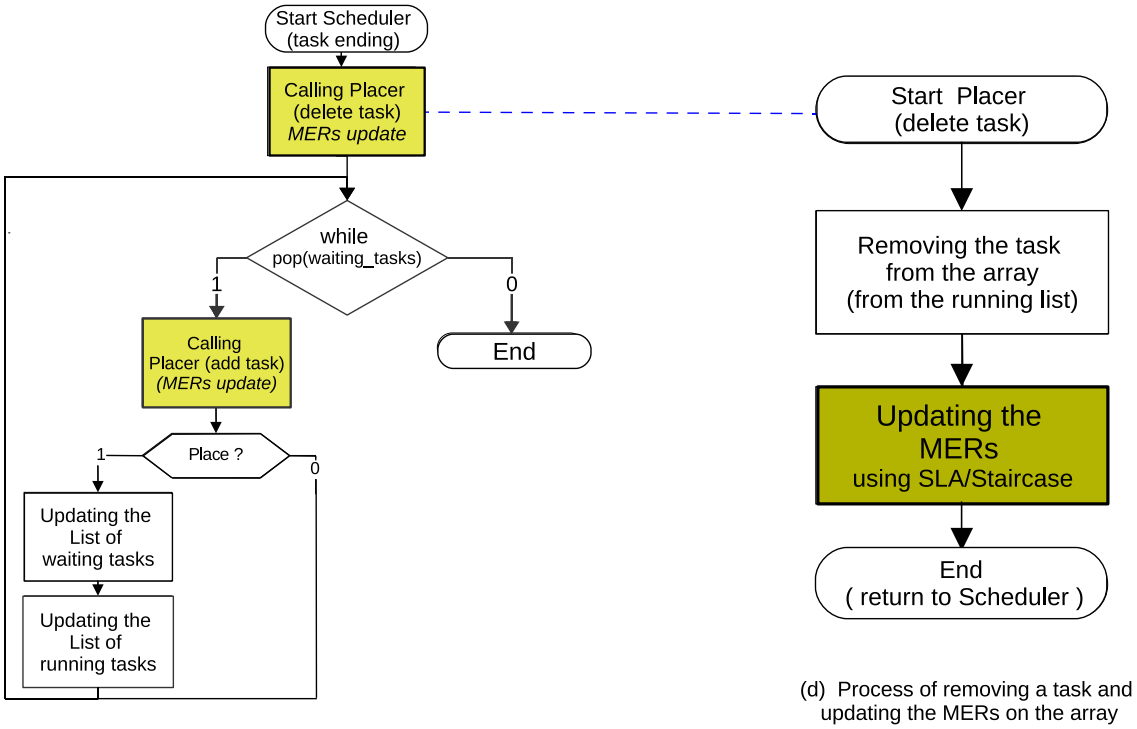
   Parameters of generated tasks are detailed in table 4.1-(a). These parameters are uniformly distributed in the interval $[min, max]$. Table 4.1-(b) shows the results of the tasks rejection ratio and the number of calls to MERs update function. The array is scanned and the list of MERs updated every time a task is added to or removed from the array. For measuring tasks rejection ratio and the number of calls to function that updates the list MERs, the simulations were conducted using a set of 2000 tasks. They were run on a desktop computer clocked at 1.8 Ghz. Therefore, these results are platform independent and are easier to implement and run on a desktop PC.

   The tasks rejection ratio is almost identical for *best fit* and *first fit* fitting strategies. In fact



(a) Process of scheduling an arriving task on the FPGA (b) Placing a task on the FPGA and updating the MERs

Figure 4.2: Scheduling one task on the reconfigurable array using a MERs-based placement algorithm.



(c) Scheduling steps at a task ending; attempt to place waiting tasks

(d) Process of removing a task and updating the MERs on the array

Figure 4.3: Scheduling the end of a task using a MERs-based scheduling algorithm.

Tasks parameters	*min*	*max*
Arrival time	1	2000
Width	1	19
Height	1	19
Execution time	5	35
Deadline	2	8

(a) Tasks parameters for simulation

SLA/Staircase	*First Fit*	*Best Fit*
$R_j(\%)$	13	12
N_{MERs}	3470	3502

R_j : *Tasks rejection ratio (%)*

N_{MERs} : *Number of calls to MERs update function*

FPGA : width = 50 ; height = 40

(b) Partial simulation results

Table 4.1: Simulation paremeters for tasks and the reconfigurable array (FPGA)

in most cases best fit rejects only 1 to 2% fewer tasks than first fit. However, the global runtime overhead is greater for the best fit algorithm because the greater the number of tasks that are placed then removed, the greater the number of calls to function that updates the MERs. This improvement of up to 2% on the tasks rejection ratio has been confirmed through identical experiments conducted with different sets of 2000 tasks.

Conversely, measuring the runtime overhead is more sensitive and platform dependent. The embedded FPGA platform presented above were used in order to get accurate results. The MicroBlaze soft core embedded processor was clocking at 50 MHz. The application scenario consisted of a set of 80 tasks with parameters similar to those in table 4.1-(a). Figure 4.4 details the obtained results. As MERs-based algorithms use a scan approach, the time taken to update the MERs is proportional to the size of the array. The size of the FPGA is assumed to be *50 X 40*, which is relatively small compared to the ever increasing size of current FPGAs. According to the results, *Staircase* algorithm outperforms *SLA* in terms of runtime overhead as it runs more than twice as fast. It takes an average of $6ms$ to update the list of MERs and can take an average of $15ms$ in the worst cases. While looking in depth at the scheduling process described in figure 4.2, It can be observed that *updating the MERs* (performed by the placer and highlighted in figures 4.2-(b) and 4.3-(d)) is only a part of the process, albeit the most time consuming. The time taken to reconfigure the reconfigurable array is not shown here and cannot be neglected.

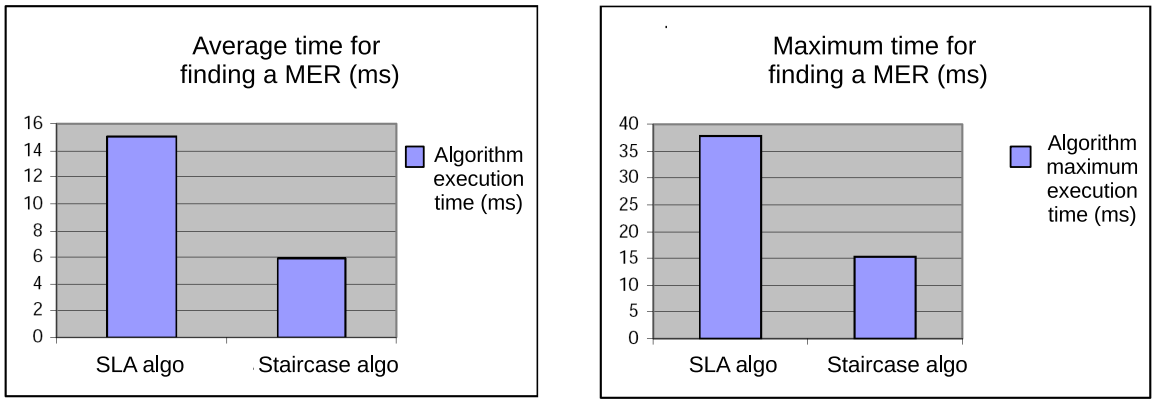


Figure 4.4: Time for finding a MER (Maximum Empty Rectangle)

4.2.4 Lessons Learnt from Preliminary Results and Conclusion

The above preliminary results clearly show one thing. MERs-based algorithms can not be used in real-time systems that have to respond in a timeframe below a given threshold. For example, let's assume a hard real-time where the tick of the scheduler is $20ms$, it would not be practically possible to respect the real-time constraint. As shown on the left of figure 4.4, the average time for updating the list of the MERs is $6ms$ for Staircase algorithm and $15ms$ for SLA algorithm. However if considering the worst case (figure 4.4-right), the update time is $15ms$ for Staircase and $37ms$ for SLA algorithm.

Figure 4.5 depicts a timing detail of a scheduling that uses Staircase algorithm for area management. It shows that each task placement is preceded by a MER detection that takes at least $15ms$ in order to update the list of MERs, in addition to the configuration time that does not explicitly appear here. Indeed, if considering the trend of configuration time of the FPGA over the last decade (shown in figure 4.6) as well as other scheduler runtime overheads, then the time needed to schedule one hardware task could easily reach hundred plus milliseconds. For example, Xilinx Virtex-6 FPGA may be fully reconfigurable in about $50ms$ at the fastest possible configuration speed. Let's recall that the reconfiguration time of a hardware task on the array is proportional to the size of the task.

However, configuration overhead may rapidly become too long for some real-time systems e.g. where scheduling intervals are $\leq 100ms$.
Furthermore the sizes of the reconfigurable device (50X40) along with tasks (19X19 maximum) used in the simulations and reported in table 4.1 are relatively small. Hence, resulting timing

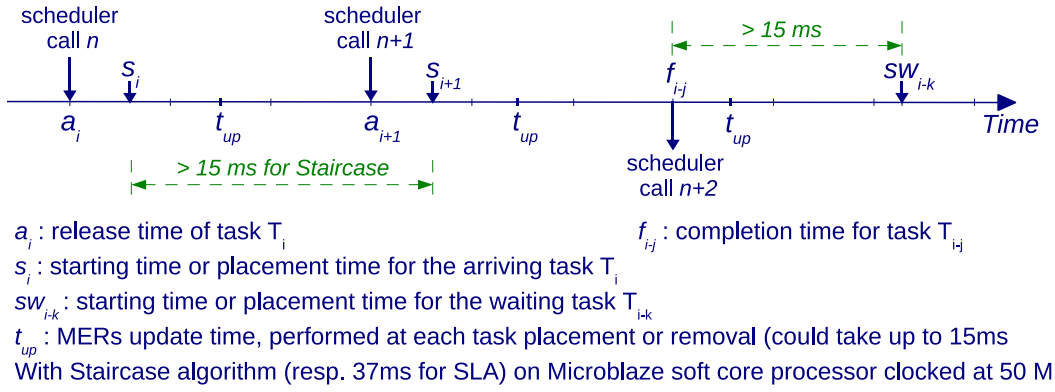


Figure 4.5: Scheduling timing and overheads (staircase)

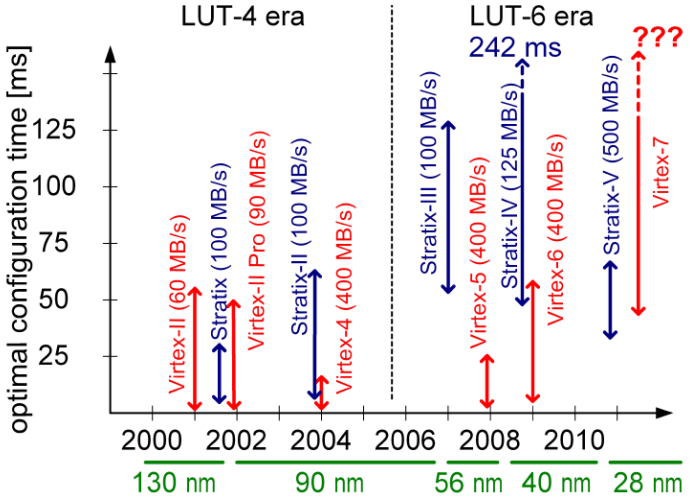


Figure 4.6: Evolution (over one decade) of the configuration time of a full FPGA when considering the fastest possible configuration speed (Koch and Torresen, 2010).

measurements of figure 4.4 above are quite optimistic in the sense that the scanning process for updating the list of MERs depends on the sizes of the reconfigurable array and tasks.
Consequently it is better to focus on non-optimal but faster placement algorithms. MERs-based algorithms will be essentially used to assess how bad behave non-optimal algorithms when compared to the optimal solution. Different trade-offs between scheduling/placement algorithms and placement quality were made, in order to reach reasonable algorithms complexities and runtime

overheads that enable online real-time scheduling. Therefore three main options were identified:

(i). ***Non-optimal placement strategies***

As much as possible, it is preferable to use non-optimal placement algorithms which are not based on maximum empty rectangles (MERs). Detecting the latter requires a scan of the array that may be of complexity $O(w \cdot h)$ in the worst case, where w and h are the width and the height of the reconfigurable array. However, the placement quality can also be improved by using overlapping rectangles (overlapping rectangles are not necessary the MERs), and which are of quadratic complexity with respect to the number of placed tasks.

(ii). ***Looking-ahead scheduling for online clairvoyant paradigm***

As much as possible, *looking-ahead scheduling* algorithms (presented in section 3.7.2, Chapter 3) must be used in order to derive benefit from the online clairvoyant paradigm. Indeed, the online clairvoyant paradigm allows a *looking-ahead scheduling* algorithm to prospect future states of the processing resources, and therefore improves the scheduling by taking rapid scheduling decisions. However *looking-ahead scheduling* algorithms are more complicated as prospecting future states of the reconfigurable array implies mimicking many tasks starting and/or tasks completion on the array. Tasks starting and ending induce placement operations that should better not be of high complexity or runtime overhead. Therefore, MERs-based areas management is not advised.

(iii). ***Multi-shape tasks***

Another way of improving the scheduling/placement quality without increasing the algorithm complexity and runtime overhead is to generated several variants per task. Such approach is also discussed in Danne and Platzner (2006a) for offline scheduling. Mahr et al. (2011) also considered more than one variants per task while studying online scheduling on a reconfigurable array. Their online module selection evaluates various scheduling approaches that mainly rely on the size of the modules. However, they do not correlate the modules selection with the underlying placement strategy. Conversely, this thesis evaluates the impact of the number of variants per task along with their aspect ratio, with respect to the underlying placement strategy. In this thesis, this approach is denoted as *multi-shape-based* placement. It increases the probability of fitting more tasks on the reconfigurable array, which can be very valuable in a real-time online context. Variants of the same task differ from each other by their size and shape and the corresponding execution time. *Multi-shape-based* placement therefore requires an extra effort at design time to generate and store many

variants (bitstream) per task, but fortunately eases the runtime placement process.

For online real-time scheduling, (i) and (ii) above suggest to combine scheduling and placement is such a way that both are not of high complexity and runtime overhead. For example, if the placer relies on a MERs-based algorithm, then the scheduler must be of low complexity, which is *without-looking-ahead* assignment policy (section 3.7.1, Chapter 3).
However, when using looking-ahead scheduling, a simple placer is advised (e.g. using nonoverlapping rectangles managed by a binary or ternary tree). As *multi-shape-based* placement does not really increase placement algorithm complexity, it could be combined with *looking-ahead* scheduling to achieve a better scheduling.

Few priority-driven scheduling algorithms are proposed, where tasks parameters that calculate the priority are either geometric (width, height, size, shape ratio) or temporal (deadline, laxity, etc.), or a combination of both. Furthermore, priority-driven scheduling strategies are combined with *looking-ahead* and *without-looking-ahead* scheduling approaches.

4.3 Models

An overview of scheduling problems targetting uniprocessor and multiprocessor systems has been presented in Chapter 3. The chapter gave some basic processor and tasks models that suit to these problems. It then drew some similarities with reconfigurable hardware systems scheduling. This section focuses on the problem of scheduling an application on a partially and dynamically reconfigurable array. The application consists of a set of independent aperiodic real-time hardware tasks that have to meet their deadline. The section presents the models which reflect as much as possible an application, the processing resources, the scheduler and the placer.

4.3.1 Real-Time Tasks and Applications Modeling

In this thesis, an *application* is a sequence of tasks. Each *task* T_i fulfils a specific and identifiable function in the application. A task corresponds to a sequence of operations to be run on the computing resources. The same task could be run several times. A *job* J is a running instance of a task. $J_{i,j}$ denotes the j^{th} instance of task T_i. Sometimes, it will be simply referred to as task. From a modular perspective, an application is made of one or several modules, and each module itself consists of one or several tasks. Hence, a task is the smallest identifiable block of an application which fulfils a clear specific function. This task could be a home-made IP or a

manufacturer's IP provided as a pre-synthesized and technologically independent netlist. It may also be an HDL (e.g. VHDL or Verilog) description of an electronic functionality.

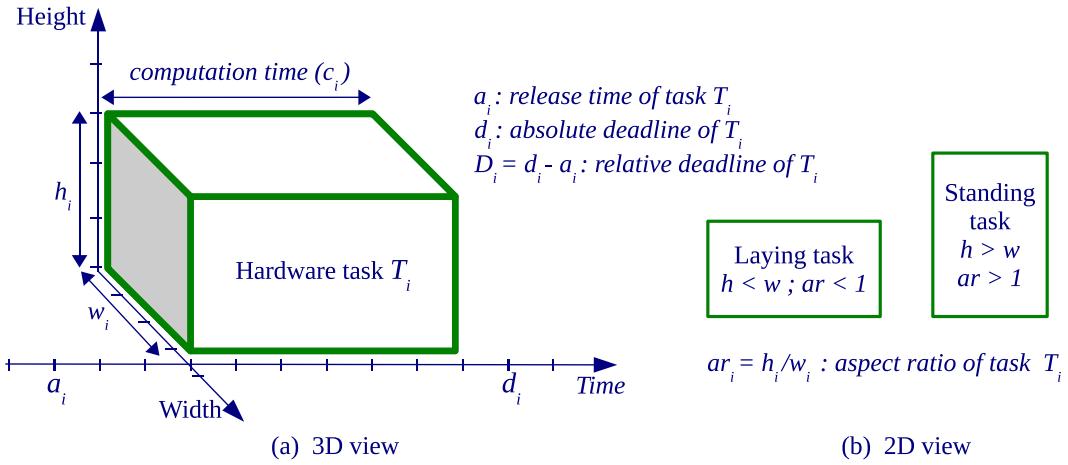


Figure 4.7: A hardware task model: 2D view (b) and 3D view (a)

Basic Hardware Tasks Model

In this thesis, a hardware task T_i is an electronic functionality to implement in a reconfigurable device. Hence, it is a bitstream ready to be downloaded in the reconfigurable hardware device. A bitstream is generated after synthesis, placement and routing of a digital circuit. It contains information about the position of the circuit (placement) on the chip.
Hereinafter are some basic assumptions widely accepted in related work, and that make the study simpler.

- ***Structural and temporal characteristics***
  A hardware task T_i shares with a software task similar temporal characteristics that are:
  - the *arrival* or *release time* a_i,
  - the *computation time* c_i or *execution time* e_i ,
  - the *finishing* or *completion time* f_i,
  - the *absolute* (resp. *relative*) *deadline* d_i (resp. D_i),
  - the *period* (resp. minimum inter-release time) P_i for a periodic task (resp. for a sporadic task), etc. In aperiodic tasks model, as each task only arrives once, making

the definition of a period is meaningless. However it is assumed that aperiodic tasks share the same period $P_i = t_D$ where t_D is equal to the biggest absolute deadline in the system as expressed in equation 4.4, page 151.

In addition, a hardware task has functional and structural characteristics. Functional characteristics reflect the behaviour or the functionality of the task. In the present model, functional characteristics of a task are not taken into account for its placement. Structural characteristics provide geometric information on the task (e.g. area size, shape, etc.). Hence, this information is the most valuable for finding a place that may fit the task while timing characteristics provide information for scheduling.

In general, as shown in figure 4.7-(b), a hardware task T_i is a rectangular module to be implemented on the reconfigurable device.

Hence, the most common geometric characteristics are: the *width* w_i and the *height* h_i.

A rectangular shape simplifies task placement, but at the cost of *intra-task fragmentation* as shown in figure 3.24, page 126 and discussed in section 3.9.1. Indeed, each task is represented by the rectangle that encompasses all the resources actually used by the task on the reconfigurable array (figure 3.24). Therefore what is denoted as *intra-task fragmentation* is the resulting lost of resources within the encompassing rectangle.

By mixing geometric and temporal characteristics, hardware tasks may be seen as 3D cubic boxes as mapped in figures 4.7-(a) and 4.9.

- ***States of a hardware task***

  As pictured in figure 4.5 (page 144), a task T_i is *active* within the interval that spans from it release time a_i to its finishing time f_i. Within this time interval, a task may be *ready-to-run*, *running* or *waiting*. Figure 4.8 depicts different states of a task. Once tasks are created, they are in the *idle* state. They are then released depending on the task model (periodic, aperiodic, sporadic). Once released a task becomes active and available (ready) for scheduling. A task in the *ready* state is ready to run and just waits for the scheduler to give it access to the computing resources. This means either other tasks with higher (or equal) priority are running in the reconfigurable array, or there is not enough contiguous free area to accommodate the ready task. Tasks in the *running* states are those that are currently running on the reconfigurable array. A running task may move to the *waiting* state if it waits for an event or resources before continuing its execution. Events may be temporal (delay) or external (interaction with environment). A task which is in *waiting*

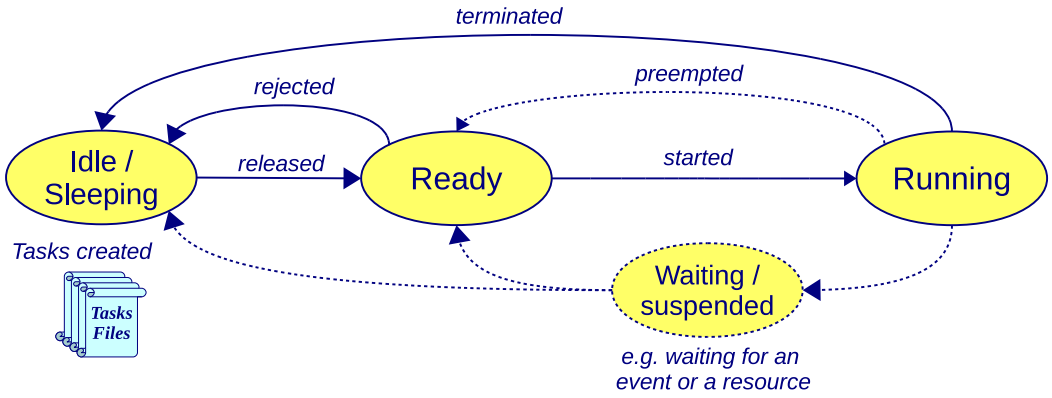


Figure 4.8: Different states of a hardware task

state can move to the *ready* state after the expected event occurs, or can move to the *idle* state if the event is out of time (e.g. if the waiting task cannot still meet its deadline).

In this thesis, the waiting state along with states transitions in a dotted line are not taken into account. Which means that there is no preemption and there is no tasks moving to the waiting state.

- ***Aspect ratio, standing vs laying task***

  The *aspect ratio* of a task T_i is given by equation 4.1 and shown in figure 4.7-(b). Hence the task is square-shaped for a ratio equals to 1.

  $$ar_i = \frac{h_i}{w_i} \tag{4.1}$$

  On figure 4.7-(b) is also depicted a standing and a laying task. A task T_i is denoted as *standing task* if $ar_i > 1$ (resp. *laying task* if $ar_i < 1$). Synthesis of a hardware task provides some flexibility in choosing the desired aspect ratio.

- ***Multi-shape same size task***

  As previously stated in section 4.2.4 above, there may be many variants of the same task. On figure 4.7-(b) both standing and laying tasks may be similar with respect to everything (functionality and timing) but their shape (width and height). This is more detailed later in section 5.5, Chapter 5 while presenting *multi-shape-based* scheduling algorithms.

- ***Relocatability, rotatability***

  A hardware task is a synthesized and pre-routed electronic circuit, which is assumed to be *relocatable* wherever on the chip, as far as there is enough contiguous free resources to accommodate it. This full relocatability assumes that the reconfigurable hardware device is homogeneous. However if there is some heterogeneity on the device area, the task may be relocatable only on limited parts of the chip where the resources are similar to resources used by the task. Such a case is depicted later in figure 4.10, page 154.

  However, hardware tasks are not rotatable, meaning that an area accommodates a task only if there is no need to rotate the task. For example, albeit the *laying task* and the *standing task* of figure 4.7-(b) are similar in terms of area size, an area which width and height are similar to the width and the height of the *laying task* cannot fit the *standing task*.

- ***Online clairvoyant paradigm, preemption and precedence constraints***

  Unlike offline models which are deterministic models, this thesis copes with more realistic application patterns or scenarios where it is not always possible to know the complete application flow beforehand. To reflect this situation, unknown information are expressed by randomly generating tasks parameters (arrival time, processing time, width, height, etc.) with known probability distribution. Hence, arrival time of each task is arbitrary and is unknown beforehand. As long as a task is not released, its parameters are kept unknown to the scheduler. This behaviour corresponds to the online clairvoyant paradigm.

- ***Preemption and precedence constraints***

  This thesis considers the case of independent and non preemptable real-time tasks that are subjected to deadline constraints. Therefore, execution time and deadline of real-time tasks are required.

- ***Software vs hardware versions of tasks***

  It is assumed that for any hardware task there exists a software version. Hence, a task that cannot be run in hardware can be run in software with the corresponding execution time. In general, hardware tasks run faster, making hardware implementation suitable for computing acceleration. Nevertheless, this work does not deal with the scheduling of software versions of tasks.

Online Application Model

In many applications, tasks arrive aperiodically. For example, nowadays, embedded systems are becoming more interactive. Tasks may arrive because an event occurred or a sensor reading is available. The system may then accurately estimate on the fly the resources and the time required to process newly acquired input information. Such an application model where the information on tasks are known as they arrive corresponds to the *online clairvoyant* paradigm. In addition, if the tasks in the application are submitted to deadline, then it is denoted as *online real-time application.*

This thesis mainly consider online real-time applications featuring an online clairvoyant paradigm (more detailed in Chapter 3, section 3.4.2).

There are various kind of online applications, including semi-online ones. Indeed, in most cases, there are some partial information available that could help improving the scheduling. For example some information may be available on tasks (e.g. their maximum and/or minimum size, their maximum and/or minimum execution time, the total number of tasks, etc.) or on the application (e.g the total number of resources needed by the application, etc.).

An *application* Ap_k or Γ_k is a set of k tasks $\Gamma_k = [T_1, T_2,...T_k]$. Γ_k may be viewed either as a random task graph (without precedence constraints) or as a data flow graph with precedence constraints as mapped in figure 4.9. It is assumed that the tasks arrive as they are ordered in Γ_k. This is expressed in equation 4.2 below where a_i is the arrival time of task T_i.

$$\forall T_i \ , \ T_j \in \Gamma_k \ , \ i < j \Rightarrow 0 < a_i \leq a_j \tag{4.2}$$

Implicitely the release time of the application is denoted as t_a and corresponds to the release time of the first task or job in the application Γ_k. Let T_i be the first task released in Γ_k,

$$t_a = a_i \Rightarrow \forall T_j \in \Gamma_k, T_j \neq T_i \ , \ a_i \leq a_j \tag{4.3}$$

Identically, the absolute deadline of the application Γ_k is denoted as t_d and corresponds to the latest absolute deadline of tasks in the application. Let T_i be the latest deadline d_i of a task in Γ_k,

$$t_d = d_i \Rightarrow \forall T_j \in \Gamma_k, T_j \neq T_i \ , \ d_i \geq d_j \tag{4.4}$$

$t_D = t_d - a_i$ is the relative deadline of the application. Therefore Γ_k may be expressed either as $\Gamma_k = [T_1, T_2,...T_k, t_d]$ or as $\Gamma_k = [T_1, T_2,...T_k, t_D]$.

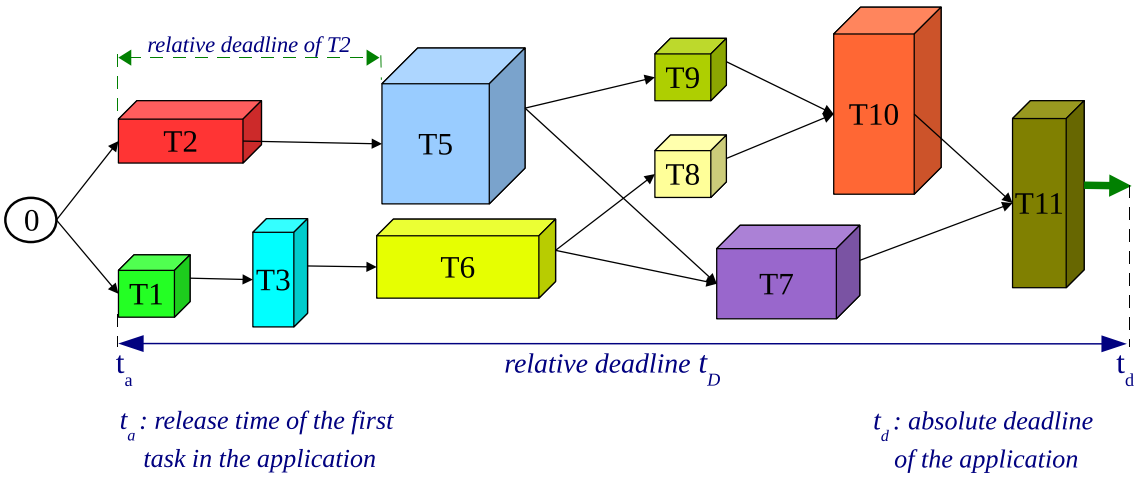


Figure 4.9: An application as a set of boxes (taskgraph).

4.3.2 Reconfigurable Devices Area Models

Area models of reconfigurable hardware devices drastically impact on the complexity of proposed tasks placement solutions. Therefore these models cannot be studied separately. Depending on how the diversity of reconfigurable resources are taken into account while placing a task, one distinguish mainly *homogeneous* and *heterogeneous* area models. Furthermore, the *hierarchical* model discussed in Chapter 2 section 2.5.11 is derived from the latter model. However, depending on whether the reconfigurable technology enables columnwise partial reconfiguration or random partial reconfiguration, one distinguish 1D placement and 2D placement.

Homogeneous model *vs* heterogeneous model

In Chapter 2 section 2.5.2 is given a clear definition of homogeneous and heterogeneous reconfigurable hardware devices along with some commercial examples. This section will propose a simple model of theses architectures. It was previously said that a DPRHW may be viewed as a 2-Dimensional array of Combinational Logic Blocks (CLBs) surrounded by vertical and horizontal programmable routing channels. In the basic model proposed here, the routing channels will not be explicitly mentioned , as they do not have any influence on the proposed placement model. Indeed, the latter model assumes that a task can fit in a rectangular area on the array as far as there is enough contiguous free space to include the task.

A simplified model of an homogeneous reconfigurable hardware device is mapped in figure

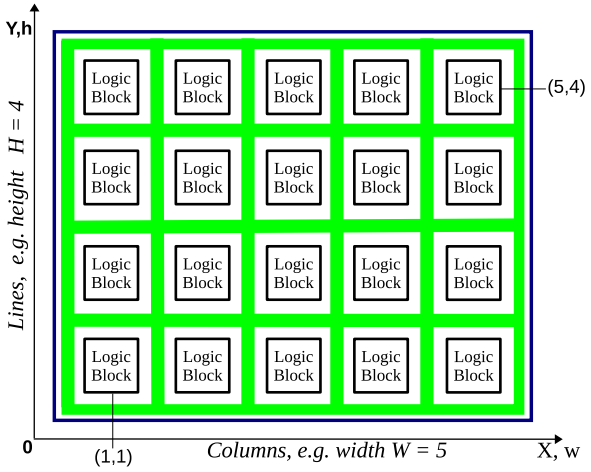
4.10-(a). Even if DPRHW like FPGAs are becoming more heterogeneous today, this simple model is still widely used to outline scheduling and placement problems. In general, Programmable Logic Blocks are organized in W columns and H rows where W and H are respectively the width and the height of the reconfigurable device. As it is assumed that the reconfigurable hardware device enables partial reconfiguration, $A_{fpga} = W \cdot H$ expresses the number of independently reconfigurable units (CLBs) on the reconfigurable array. A_{fpga} also reflects the size or the area of the array.
As shown in figure 4.10-(a) and (b), an $X - Y$ coordinate axis is used to physically localize each CLB or any other resource on the reconfigurable array.

A simplified model of an heterogeneous reconfigurable hardware device is mapped in figure 4.10-(b). Unlike figure 4.10-(a) there are some pre-built or pre-instantiated dedicated blocks that are already at some fixed positions on the chip. As discussed in Chapter 2, heterogeneous reconfigurable array is the current dominating trend in FPGA architecture. However, the above presented homogeneous model could be easily extended to represent an heterogeneous array. Indeed, as it is always assumed that a hardware task to be implemented is relocatable wherever on the homogeneous reconfigurable array as far as there is a rectangular portion that accommodate it, the following two assumptions can be made:

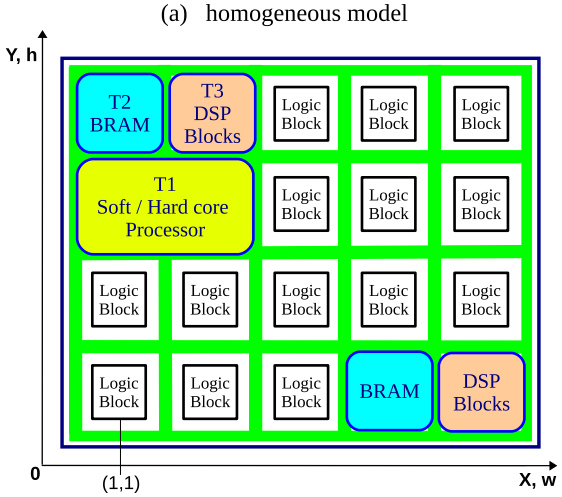
(i) an heterogeneous array is an homogeneous array that has already accommodated some static and position-constrained tasks. For example, the heterogeneous array on 4.10-(b) embeds some hardwired blocks (softcore/hardcore processor, BRAM memory, DSP blocks) that are considered as permanent and position-constrained tasks (T_1, T_2 and T_3, top left of the array). Only the remaining areas are still available for placing other tasks.

(ii) an heterogeneous array is an array where each task has one or few possible placement positions on the array depending on the matching between the kind of resources needed by the task and the position of such resources on the array. The case is shown in figure 4.10-(b) where dedicated resources are in the bottom right of the array. The BRAM and the DSP blocks may be assigned to specific tasks that mainly need these resources to be efficiently implemented.

Chapter 3 section 3.8.5 presented related work (e.g. Koester et al., 2005, 2006) that improve the placement quality by taking into account the hardware heterogeneity in order to optimize the resources utilization. In their model, each task has a few possible placement positions on the array, and the heterogeneity of the array consists of two kind of resources: configurable logic blocks and



(a) homogeneous model



(b) heterogeneous model

Figure 4.10: Simple models of homogeneous and heterogeneous reconfigurable array

memory blocks.

In this thesis, time overhead due to configuration of a hardware task on the reconfigurable hardware device is assumed to be negligible. Example of full reconfiguration of FPGAs are pictured in figure 4.6.

4.3.3 Scheduler Model

As this thesis considers the online clairvoyant paradigm, the scheduling algorithm is not allowed to use information about the future. Consequently, at time t, the scheduler is not aware of any information (a_i, e_i, d_i, w_i, h_i, etc.) on any task arriving at time $a_i \geq t$.

As depicted in figure 4.11, once a task $T_i = (a_i, e_i, d_i, w_i, h_i)$ of the application Γ_k is released in the system, the scheduler takes it into account and schedules it in conjunction with the *placer*. Therefore, using a given assignment policy, the scheduler decides which tasks have to be run on the reconfigurable array. The scheduler acts as a tasks manager and might manage certain lists of tasks. It requests the placer to find a position (x_i, y_i) free at current time or in the future depending on whether the scheduling policy is *looking-ahead-based* or not. Tasks that cannot meet their deadline are rejected. A position (x_i, y_i) and a starting time s_i is then assigned to each arriving task $T_i \in \Gamma_k$ if possible, in such a way that task T_i does not overlap concurrently spatially and temporally with any other task in the system. The placer acts as a resources manager by keeping the state of the reconfigurable area, by optimizing its use thanks to management heuristics, and by providing the scheduler with best available areas.

Equations 4.5 and 4.6 express the conditions for scheduling tasks on the reconfigurable array. These conditions allows many tasks to run either on a space-sharing basis or on a time-sharing basis. The placer is responsible for verifying the space-sharing conditions, which is $(x_i + w_i) \leq (x_j)$ for the 1D placement. The first equation 4.5 is for the 1D placement while the second 4.6 is for the 2D placement.

$$\text{With 1D placement: } \forall T_j \in \Gamma_k, T_j \neq T_i \ , \ \begin{cases} [(x_i + w_i) \leq (x_j)] \vee [x_i \geq (x_j + w_j)] \vee \\ [(s_i + e_i) \leq (s_j)] \vee [s_i \geq (s_j + e_j)] \end{cases} \tag{4.5}$$

$$\text{With 2D placement: } \forall T_j \in \Gamma_k, T_j \neq T_i \ , \ \begin{cases} [(x_i + w_i) \leq (x_j)] \vee [x_i \geq (x_j + w_j)] \vee \\ [(y_i + h_i) \leq (y_j)] \vee [y_i \geq (y_j + h_j)] \vee \\ [(s_i + e_i) \leq (s_j)] \vee [s_i \geq (s_j + e_j)] \end{cases} \tag{4.6}$$

where a_i, e_i, d_i, w_i, h_i are respectively the arrival time, execution time, deadline, width and height of task T_i.

As discussed earlier in Chapter 2, section 2.7.3 and illustrated in figure 2.18 and figure 2.19, this thesis considers a scalable and distributed multi-RTOS architecture with respect to application requirements and system constraints. Hence, as it focuses on the management of the reconfigurable part of the system, the scheduler is devoted to that part, beside other schedulers managing other

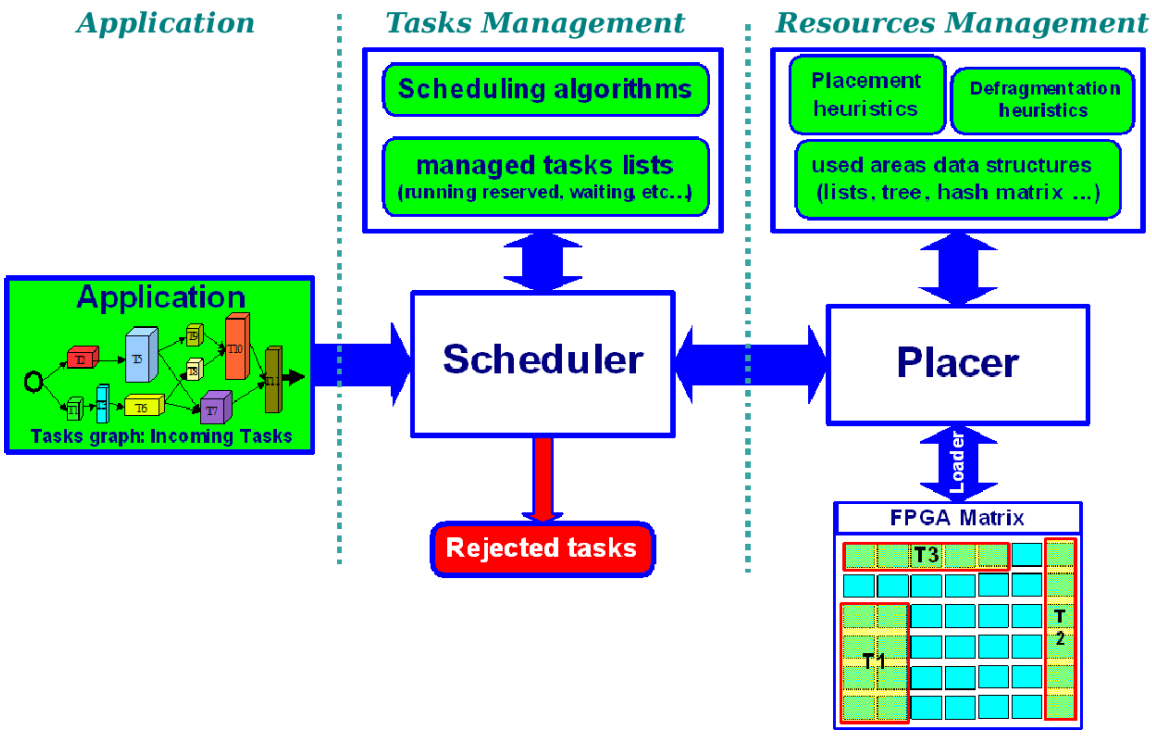


Figure 4.11: The global simulation model

PEs, under the supervision of a global OS. Hence, the scheduler responds (successfully or not) to the requests of the global scheduler. This assumes that tasks in the system may have numerous alternative implementations (software, hardware, etc.) with various costs, performances and QoS.

In this thesis, the scheduler implements a non pre-emptive scheduling policy that does not enable tasks migration.

4.3.4 Placer Model

As stated above, the *placer* responds to placement requests sent by the scheduler. As depicted in figure 4.11 the placer and the scheduler interact constantly. The former acts as the resources manager for the latter.

The placer model is more detailed in figure 4.12. On one hand as shown in the left branch of the figure, the placer partitions and manages the reconfigurable array through a data structure which keeps the state of the array. For this, it uses various splitting, merging and defragmentation

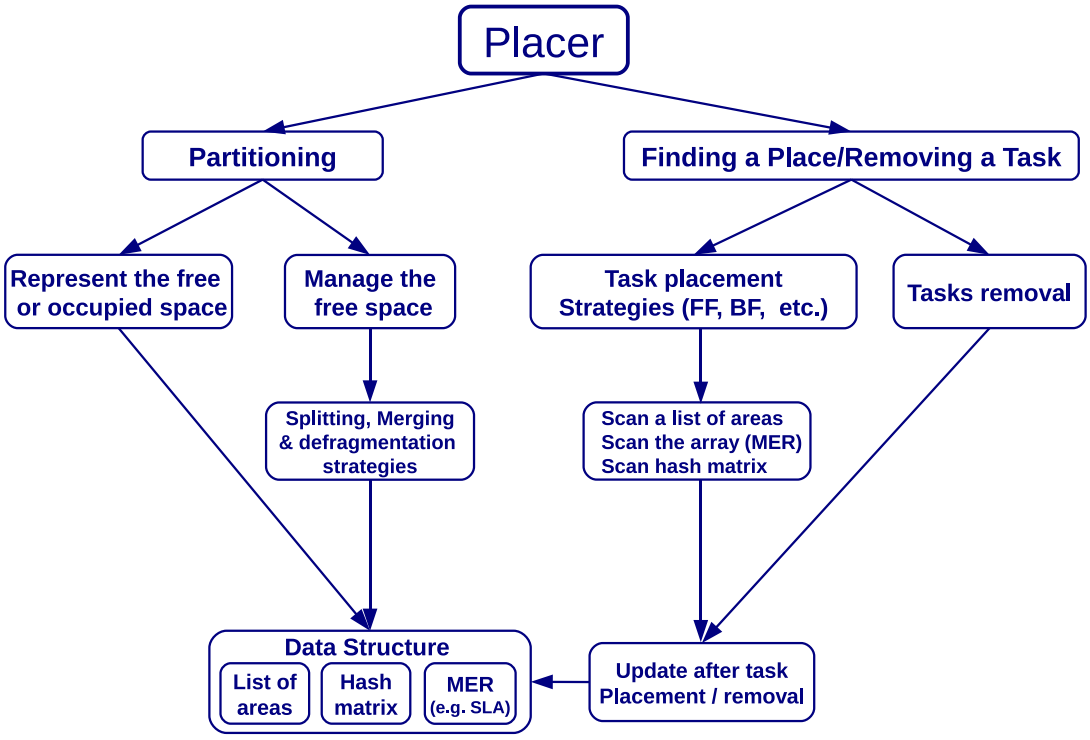


Figure 4.12: The placer model and its different functional parts.

strategies. On the other hand as depicted by the right branch of figure 4.12, the placer allocates and deallocates areas to tasks and updates the data structure accordingly. To do this, the placer searches for available areas. It then uses different fitting strategies (1D, 2D, BF, FF, etc.) that have been presented in Chapter 3 section 3.8.4. Figures 3.17 page 114 and 3.18 page 115 depict examples of fitting strategies.

The data structure may be a simple list of areas (nonoverlapping or overlapping), a binary tree (e.g. Bazargan et al., 2000) or a hash matrix (e.g. Walder et al., 2003).

The quest for a quality placement especially in an online real-time context may be to:

(i). find the best data structure that stores information of free spaces available on the reconfigurable array and that:

- eases the search for places to fit new tasks.
- eases the update of the structure after adding or deleting a task.

(ii). find meaningful metrics that assess the placement quality and the fragmentation of the

reconfigurable array and that are easy to calculate. In addition, if possible, the placer may take into account future tasks parameters distribution (if known, e.g. width, height, etc.) in order to improve tasks placement.

Achieving such a quest is rarely possible and requires a trade-off between the two points mentioned above. For example, the hash matrix proposed by Walder et al. (2003) stores free areas and allows the placer to quickly index them with a constant time complexity. But such an easy search comes at the cost of heavy matrix update process at each task placement and removal.

4.4 Metrics

The metrics rate the performance of scheduling and placement of an application which consists of k tasks. They are almost similar to scheduling metrics in a microprocessor, with some specificity due to the underlaying placement.

4.4.1 Reconfigurable Hardware Resources Metrics

The total amount of resources available on a reconfigurable array corresponds to its area

$$A_{fpga} = W \cdot H$$

where W and H are respectively the width and the height of the reconfigurable array.
The total amount of resources provided by the device within a given time interval that spans from time t_1 to time t_2 is given by equation 4.7.

$$A_{fpga(\delta t)} = W \cdot H \cdot \delta t \tag{4.7}$$

where $\delta t = t_2 - t_1$

4.4.2 Tasks Metrics

This thesis mainly consider the problem of scheduling a set of aperiodic real-time tasks on the reconfigurable array. Hardware tasks metrics are derived from periodic tasks metrics. They are defined as follows:

i). **Time utilization factor** $(U^{(t)}{}_{Ti})$

The time utilization factor of a periodic task T_i is the ratio between the task execution time

and its period as show in equation 4.8. It reflects the fraction of time spent by the task on the reconfigurable array within the period of the task.

$$U^{(t)}{}_{Ti} = \frac{e_i}{P_i} \tag{4.8}$$

where e_i and P_i are respectively the computation time and the period of T_i.

In the case of an aperiodic task, the *time utilization factor* of T_i is defined as

$$U^{(t)}{}_{Ti} = \frac{e_i}{D_i} \tag{4.9}$$

where D_i is the relative deadline of T_i. Therefore, $res_i = A_i \cdot U^{(t)}{}_{Ti}$. However the time utilization factor metrics does not inform on the amount of resources really needed by the task.

ii). **Absolute computational load** (Res_{Ti})

The absolute computational load of a hardware task T_i as the total amount of resources required to complete an instance of the task. It is given by equation 4.10 where w_i , h_i, e_i and A_i denote respectively the width, the height, the execution time and the area requirement of T_i.

$$Res_{Ti} = w_i \cdot h_i \cdot e_i = A_i \cdot e_i \tag{4.10}$$

iii). **Relative computational load** (res_{Ti})

The relative computational load of a task T_i is its *computational load* relative to its relative deadline D_i for an aperiodic task (period or minimal inter-release time P_i for a periodic task and a sporadic task respectively). It reflects the fraction of time within the active state of the task that has been really spent by the task on the reconfigurable array, and the area requirement of the task.

The *relative computational load* res_i corresponds to the ***system utilization*** ($U^{(s)}{}_{Ti}$) of a task T_i in monoprocessor scheduling that is given by:

$$U^{(s)}{}_{Ti} = res_{Ti} = U^{(t)}{}_{Ti} \cdot A_i = \begin{cases} \frac{Res_{Ti}}{D_i} = \frac{e_i}{D_i} \cdot w_i \cdot h_i & \text{for aperiodic tasks} \\ \frac{Res_{Ti}}{P_i} = \frac{e_i}{P_i} \cdot w_i \cdot h_i & \text{for periodic and sporadic tasks} \end{cases}$$

4.4.3 Application Metrics

Let Ap_k or $\Gamma_k = [T_1, T_2, ...T_k, t_d]$ be an *application* or a set of k tasks that must be run to completion before its absolute deadline t_d. Let's define:

i). *Time utilization factor* ${U^{(t)}}_{\Gamma}$

The *Time utilization factor* of the complete set of aperiodic tasks Γ_k is expressed as follows :

$$ {U^{(t)}}_{\Gamma} = \sum_{i=1}^{k} \frac{e_i}{D_i} \tag{4.11} $$

Obviously, the equation becomes ${U^{(t)}}_{\Gamma} = \sum_{i=1}^{k} \frac{e_i}{P_i}$ if tasks are periodic.

ii). **System utilization** ${U^{(t)}}_{\Gamma}$

The *system utilization* of the complete set of k aperiodic tasks Γ_k takes into account the amount of resources used by the set as expressed in equation 4.12.

$$ {U^{(s)}}_{\Gamma} = \sum_{i=1}^{k} \frac{1}{D_i} \cdot w_i \cdot h_i \cdot e_i = \sum_{i=1}^{k} {U^{(t)}}_{Ti} \cdot A_i \tag{4.12} $$

iii). **Current time utilization factor**

Let $\Gamma_{(t)}$ be a partial set of Γ_k that have already been released at current time t and that respects the following condition.

$$ \Gamma_{(t)} = \{T_i \in \Gamma_k : a_i \leq t < d_i\} \text{ where } d_i = a_i + D_i \tag{4.13} $$

The *current time utilization factor* of the set at current time t is then

$$ {U^{(t)}}_{\Gamma_{(t)}} = \sum_{i=1}^{t} {U^{(t)}}_{Ti} = \sum_{i=1}^{t} \frac{e_i}{D_i} \tag{4.14} $$

The *current time utilization factor* only characterizes the tasks set but does not give any indication on the load of the system and the number of processing resources. Therefore, a multiprocessor system defines a *system utilization factor* ${U^{(s)}}_{\Gamma_{(t)}}$ at a given time t that respects condition 4.13 as follows :

$$ {U^{(s)}}_{\Gamma_{(t)}} = \frac{{U^{(t)}}_{\Gamma_{(t)}}}{m} \tag{4.15} $$

where m is the number of microprocessors. This definition may be transposed in the case of an m slots partitioned reconfigurable array where any slot may fit any task as discussed in 3.6.4 and depicted in figure 3.8, page 92.

iv). **Absolute application (computational) load** Res_{Γ}

Res_{Γ} expresses the total amount of resources required to complete Γ_k and is given by the following equation 4.16

$$ Res_{\Gamma} = \sum_{i=1}^{k} w_i \cdot h_i \cdot e_i = \sum_{i=1}^{k} A_i \cdot e_i = \sum_{i=1}^{k} Res_i \tag{4.16} $$

where w_i , h_i, e_i, A_i and Res_i denote respectively the width, the height, the execution time, the area requirement and the total amount of resources of task T_i.

v). **Relative application (computational) load** (res_Γ)

Also referred to as *relative amount of resources* required to complete an Γ_k on a given reconfigurable array of size $A_{fpga} = W \cdot H$, is given by the following equation 4.17

$$res_\Gamma = \frac{1}{W \cdot H \cdot t_D} \cdot \sum_{i=1}^{k} w_i \cdot h_i \cdot e_i = \frac{1}{A_{fpga} \cdot t_D} \cdot Res_\Gamma \tag{4.17}$$

where $Res_\Gamma = \sum_{i=1}^{k} Res_{Ti}$ is the above-mentioned absolute application load of application Γ_k, and t_D the relative deadline of application Γ_k.

4.4.4 Scheduling Metrics

These metrics are related to the application and the processing resources. These metrics are available after a scheduling and reflect its quality. Some of them have been described on section 3.3.4 as example of objective functions that are very common in scheduling problems.

i). **Utilization ratio of the reconfigurable array** ($U_{fpga(\%)}$)

The average utilization ratio of a reconfigurable array (FPGA) of size $A_{fpga} = W \cdot H$ on which an application $\Gamma_k = [T_1, T_2, ...T_k, t_D]$ has been scheduled is given by the following equation 4.18:

$$U_{fpga(\%)} = \frac{\sum_{i=1}^{k} w_i \cdot h_i \cdot e_i}{W \cdot H \cdot mak} = \frac{\sum_{i=1}^{k} Res_{Ti}}{W \cdot H \cdot mak} \tag{4.18}$$

where mk is the *makespan*.

ii). **Tasks rejection ratio** $Rj_{\Gamma_k(\%)}$

$Rj_{\Gamma_k(\%)}$ is the ratio of instances of tasks $T_i \in \Gamma_k$ that have failed to be placed on the reconfigurable array.

$$Rj_{\Gamma_k(\%)} = \frac{n_j}{k} \cdot 100\% \tag{4.19}$$

where $n_j \leq k$ is the number of jobs rejected among the k jobs in the application Γ_k.

iii). **Makespan** mk

The makespan spans from the release time of the first task (of the application) to the time the last task ends. It is also denoted as *the length of the scheduling* and corresponds to the real duration of the application. The makespan is one of the most commun objective function. The smaller the makespan, the better the scheduling.

iv). **Flow time** ft **, average flow time** ft_{av} *and* **total flow time** ft_{tot}

Also known as the **response time** and expressed in equation 4.20, the flow time of a job J_i is defined as the difference between its completion time f_i and its arrival time a_i.

$$ft_i = f_i - a_i \tag{4.20}$$

The total flow time ft_{tot} (resp. the average flow time ft_{av}) is given by equation 4.21

$$ft_{tot} = \sum_{i=1}^{k} ft_i \qquad \text{and} \qquad ft_{av} = \frac{ft_{tot}}{k} \tag{4.21}$$

where ft_{tot} is the sum of the flow times of the k jobs in job sequence Γ_k, and ft_{av} the latter sum averaged by k.

v). **Waiting time** wt_i

It spans from the task T_i release time a_i to its starting time s_i assigned by the scheduling algorithm.

$$wt_i = s_i - a_i \tag{4.22}$$

The *average waiting time* of a sequence of n_a jobs, where n_a is the number of jobs accepted among the k jobs in Γ_k, is given by the total waiting time averaged over n_a as follows :

$$wt_{av} = \tfrac{1}{n_a} \cdot \textstyle\sum_{i=1}^{n_a} wt_i$$

vi). **Rejection delay** Rd_i

This metrics represents the time that flows from the release of a task T_i to its rejection. This metrics is meaningful in online real-time scheduling where one may consider more than one implementation alternatives. Hence, rejecting a reconfigurable hardware task too late may prevent the system from running it on a different computing resource (e.g. sequential processor). The latter situation may be prejudicial. This metric is proposed as a quality metric that emphasizes scheduling strategies that minimize the rejection delay. The rejection delay is expressed by as follows for a rejected task T_i :

$$Rd_i = t_{rej} - a_i \tag{4.23}$$

where t_{rej} is the rejection time of task T_i as given in equation 3.11, page 94, and a_i its release time.

vii). **Differential quality metric** UR_{qm}

this metric is proposed as a new quality metrics that emphasizes a good behaviour of the

scheduling/placement algorithm both in terms of chip utilization ratio and tasks rejection ratio. The metrics is especially meaningful in reconfigurable hardware scheduling. It is meant to ease the comparison between two algorithms that do not significantly differ either in chip utilization ratio or in tasks rejection ratio. The reason is that no matter which area management strategy is used, the reconfigurable array is submitted to a fragmentation problem that bounds its average utilization ratio.

UR_{qm} is expressed in the following equation :

$$UR_{qm} = 2\alpha \cdot [U_{fpga} - (\frac{1}{\alpha} - 1) \cdot Rj_{\Gamma_k}] \quad , \quad \alpha \in]0;1] \tag{4.24}$$

where U_{fpga} is the utilization ratio, Rj_{Γ_k} the tasks rejection ratio and α a weighting coefficient that reflects which of the two metrics is predominant. As the chip utilization ratio is to maximize and the tasks rejection ratio to minimize, the higher the difference between these two metrics the better the scheduling.

For the sake of simplicity, $\alpha = 0.5$ in this thesis, meaning that the utilization ratio and the tasks rejection ratio are given the same important. The metric UR_{qm} is meant to be positive otherwise the scheduler behaves very poorly.

For example, for $\alpha = 0.5$ the equation above becomes :

$$UR_{qm} = U_{fpga} - Rj_{\Gamma_k}$$

However even if U_{fpga} and Rj_{Γ_k} highly depends on the computational load of the application or set of tasks to schedule, $\alpha = 0.5$ is not really a fair value because $Rj_{\Gamma_k} \equiv 0\%$ is more likely to be achieved than $U_{fpga} \equiv 100\%$, because of the reconfigurable array fragmentation. $\alpha \in]\frac{1}{2};1]$ makes the utilization ratio predominant in the UR_{qm}. A negative value of UR_{qm} indicates that the scheduler behave poorly, as it achieves a high tasks rejection ratio combined with a low chip utilization ratio.

viii). **Scheduling runtime overhead**

The ***Cumulative algorithm execution time*** or *cumulative scheduling algorithm runtime overhead* represents the total amount of time spent by the scheduling algorithm to schedule and place all the tasks in an application or a tasks set. It is given by equation 4.25 below

$$Cumul_{overhead} = \frac{1}{m} \cdot \sum_{i=1}^{m} \sum_{j=1}^{n} RuntimeOverhead(i,j) \tag{4.25}$$

where m is the number of tasks sets, n the number of invocation of the scheduler while scheduling each tasks set, and $RuntimeOverhead(i,j)$ the scheduling algorithm runtime

overhead on the i^{th} tasks set during the j^{th} call of the scheduler. As runtime overhead during a single call of the scheduler is quite small, the cumulative value is more meaningful. The value is averaged on the number of tasks sets in order to reflects the portion of time devoted to the scheduling algorithm along a task set. Let's recall that the placement algorithm may be called one or many times at each scheduler invocation.

4.4.5 Feasible Schedule

A task sets Γ_k is feasibly scheduled on a reconfigurable array if the following condition is met

$$\forall T_i \in \Gamma_k \ , \ f_i \leq a_i + D_i$$

An application cannot be successfully scheduled on an FPGA if its relative computation load res_{Γ_k} exceeds 1. Hence, a *necessary but not sufficient* condition for scheduling an application $\Gamma_k = [T_1, T_2, ...T_k, t_D]$ on the reconfigurable device A_{fpga} is that the total amount of resources needed by Γ_k (which is $\sum_{i=1}^{k} w_i \cdot h_i \cdot e_i$) must not exceed the total amount of resources available on A_{fpga} during the relative deadline t_D of Γ_k (which is $A_{fpga(t_D)} = A_{fpga} \cdot t_D$). This is clearly expressed by the following condition

$$res_{\Gamma_k} \leq 1 \quad \Rightarrow \quad \sum_{i=1}^{k} w_i \cdot h_i \cdot e_i \leq W \cdot H \cdot t_D$$

deduced from equation 4.17 above. Because of fragmentation, the relative application load of a schedulable application is in general much more closer to 0.5 than 1.

Let Γ_k be a set of k aperiodically-arriving real-time tasks; tasks in Γ_k could be assumed $t_D - periodic$, t_D being the absolute deadline of Γ_k. This assumption relies on the fact that within the time interval that spans from 0 to t_D, each task is supposedly released once.

4.5 Global Simulation Model and Compatibility with the OVeRSoC Design Methodology

A functional representation of scheduling and placement simulation model is presented in figure 4.11. This simulation model encompasses basic components of a basic reconfigurable system. The simulation model may be used at two stages :

- at runtime, which means that the simulation model may be used to assess scheduling and placement algorithms through the aforementioned metrics.

- at design time, the model may be introduced in a design methodology in order to refine hardware/software partitioning. To achieve such a purpose, the global simulation model is C++ based to insure full compatibility both with the OVeRSoC design methodology discussed in Miramond et al. (2009a) and its enabling environment presented in Miramond et al. (2009b).

4.5.1 An UML Overview of the Global Simulation Model

The global simulation model of hardware tasks scheduling on the reconfigurable part of an RSoC has been depicted earlier in figure 4.11, page 156. An UML representation of this simulation model is proposed in figure 4.13.

The *basic system* consists of the *scheduler*, the *placer* and the set of *tasks*. Hence, the *basic system* instantiates each of these elements. The *basic system* first creates the *application* as a list of n tasks. It then creates the *scheduler* and the *placer*. It acts as the simulation engine. It generates the time basis for the whole system. Tasks are then released in the system through a FIFO list denoted as *arriving_tasks_fifo* and according to their release time.

Tasks are hosted in the list at their release time a_i and poped out before time $a_i + 1$. As other elements in the system learn piece by piece about tasks, this corresponds to the online clairvoyant paradigm. The *finishing_tasks_fifo* achieves the same functionality by hosting finishing tasks instead at their completion time f_i. The tasks in the *finishing_tasks_fifo* are poped out before time $f_i + 1$

The *basic system* invokes the *scheduler* everytime at least one task pops in the *arriving_tasks_fifo* or the *finishing_tasks_fifo*. As this thesis does not consider tasks preemption, these two events are the only that trigger off a rescheduling. Once the 3 main entities are built, the simulation is run to completion. The *basic system* assesses various scheduling metrics either directly or through the *scheduler* and the *placer*. The *scheduler* manages the tasks through as many lists of tasks (*tasks_queue*) as necessary, depending on its policy. A hardware task (*task_mod*) may feature many variants of different size and the corresponding computation time.

The *placer* generates different data structures needed to reflect the state of the reconfigurable array. Hence, it may instantiate an *FPGA_matrix* (as an array of *cells*), and a list of free rectangles. Indeed, the *scheduler* measures the metrics that are related to tasks (e.g. tasks rejection ratio) while the *placer* evaluates those related to the reconfigurable array (e.g. chip utilization ratio, fragmentation, etc.).

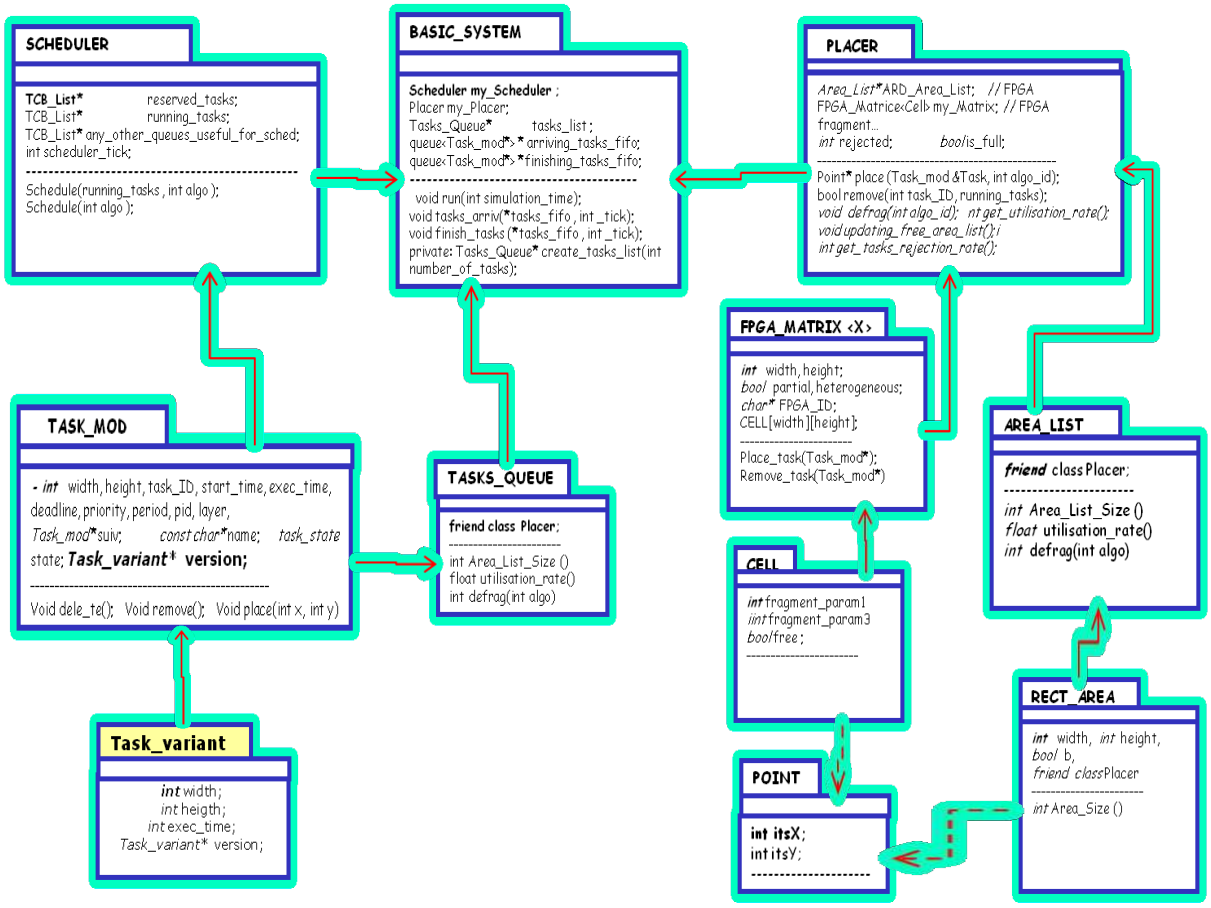


Figure 4.13: An UML overview of the global simulation model of the DPRHW-OS for a reconfigurable platform.

4.5.2 The Importance of Using a C++ Based Simulation Model

As stated in the two first chapters of the thesis, the primary objective of this work was to propose scheduling and placement strategies that suited to online real-time scheduling of hardware tasks on dynamically reconfigurable hardware devices. However this work is likely to feed any Reconfigurable SoC design methodology with information, models and metrics that were learned from the primary objective. This second aspect of this work comes within the scope of the classical design philosophy that aims to overcome as much design challenges as possible at compilation time instead of running time.

An RSoC design methodology denoted as *OveRSoC methodology* has been briefly introduced

in Chapter 2, section 2.7.3. The methodology was described as *OS-centric* or *OS-based*. This methodology is depicted in figure 2.19 page 60. The design flow is described in detail in Miramond et al. (2009a) and uses the DOGME tool (Miramond et al., 2009b), its dedicated front end for designers. The methodology relies on a system level simulation that is performed prior to and alongside the design steps and that is fed by the application and the system constraints.
The second aspect of this thesis was to insure a full compatibility with the OVeRSoC design methodology along with tools. Thus, an object oriented approach that relied on the C++ programming language was adopted in this thesis. Therefore, all the components of the UML diagram in figure 4.13 are C++ objects, fully compatible with SystemC language, OveRSoC being a SystemC-based methodology.
The main advantage of using a SystemC-based approach is its ability to adapt to almost all abstraction levels (from system level to implementation level) while providing hardware/software co-simulation and verification. This allows the designer to rely on the same models to validate different parts of his system throughout the design process. In order to deal with scheduling and placement issues, this work relied on the model pictured in figure 2.18, page 59. In the model, an application is to be mapped on a multiprocessor platform. The system consists of resources consumers (tasks or applications) and processing elements (processors, memories, etc..), both separated by an intermediate OS layer that acts as a resources manager. The OS layer hides the details of the platforms to the application layer and assigns different resources to different tasks of the application in such a way that the application runs to completion and achieves its goals.

Compatibility between any design methodology and C++ based UML model proposed in this thesis may allow any designer to tune, through simulations, both its architecture and the RTOS that suits to its management.
As illustrated by the UML diagram in figure 4.13, various models of the reconfigurable array that may be accurate at cell level were provided. These models do not express any communication, but may easily express neighbourhood between cells and therefore rectangular-shaped modules. The refined model on the array depends on the placement strategy used. The model may dynamically map the DRA[1] of figure 2.19. The same goes for the *basic system* in figure 4.13 that features the *scheduler*, the *placer* and hardware tasks, and that may map the DPRHW-OS[2] devoted to the DRA part.

[1] Dynamically Reconfigurable Architecture, also denoted as DPRHW in this thesis.
[2] Dynamically and Partially Reconfigurable Hardware Device - Operating System

4.6 Conclusion of the Chapter

This chapter has started with the proposed methodology which results from the literature review presented in Chapter 3. The methodology first consisted of doing some accurate measurements on some existing scheduling and placement algorithms. In doing so, a framework for finding the algorithms suitable for scheduling online real-time hardware tasks on DPRHWs has been established. The chapter also defined models of real-time applications that consist, in this thesis, of a set of aperiodic real-time tasks. The *Scheduler* and the *Placer* models were also presented. Afterwards, different scheduling metrics have been defined, and some of them customized to make them meaningful for DPRHW scheduling. The end of the chapter presented the global simulation model and its UML representation. The compatibility of this global simulation model with C++ and SystemC based methodology for RSoC design was emphasized, the OveRSoC methodology being taken as an example.

The next chapter will propose scheduling and placement algorithms that results from the above proposed methodology, and that suit to online real-time scheduling of hardware tasks on dynamically and partially reconfigurable hardware devices (DPRHWs).

Chapter 5

Proposed Algorithms for Online Real-Time Scheduling & Placement

5.1 Introduction

This chapter presents and discusses different scheduling and placement algorithms that rely on related work discussed in Chapter 3 and on the models detailed in the previous chapter. Regarding the scheduling algorithms, they are presented according to their two main families: the *looking-ahead* scheduling and the *without-looking-ahead* scheduling.

In *without-looking-ahead* scheduling approach, the algorithms are essentially priority-driven, where the priority of each task is based on its geometric and temporal parameters. Hence, in addition to various temporal parameters based algorithms such as EDF, LLF etc., other priority-driven algorithms that are either based only on geometric parameters of hardware tasks, or based on their geometric and temporal parameters are proposed.

As online real-time applications are targeted, scheduling algorithms are combined with appropriate placement strategies in order to provide low runtime overheads. Hence, *multi-shape scheduling* algorithms that improve tasks placement opportunities without significantly influencing the runtime overheads of the algorithms are also proposed. The idea behind the *multi-shape* approach is to provide, at design time, more than one task parameter combination for each task in the system. The *multi-shape* scheduling algorithms are also priority-driven, the main policy being to give the highest priority to the normal version of the task.

In *looking-ahead* scheduling approach, as runtime overheads tend to be higher, the latter

approach is combined with low complexity placement strategies as justified in the previous chapter. Examples of such placement strategies are : 1D placement, partitioned placement and multi-shape tasks placement.

5.2 Tasks Parameters Based Global Scheduling

The *tasks parameters-based scheduling* algorithms are *priority-driven scheduling* algorithms. A priority-driven scheduling assigns a priority to each job (or task) in the application. The priority is based on the parameters of tasks. The scheduler keeps a list of ready tasks sorted by their priorities. Therefore, the available processors resource is allocated to the highest priority tasks. In priority-driven real-time scheduling for microprocessors, a task priority is usually based on timing constraints. Hence, it is calculated using temporal parameters of the tasks (e.g. deadline for EDF, laxity for LLF, etc.).
However, in the case of a hardware task that features both temporal and geometric parameters, there are more opportunities for combining these parameters in order to derive more priority-driven algorithms. Hence this section presents other priority-driven algorithms that are based either on geometric parameters of tasks, or on a combination of geometric and temporal parameters. In this thesis, these scheduling scheme are denoted as *parameters-based scheduling*.
The priority $\wp_i$ of each task T_i is calculated using its parameters. Different tasks parameters-based scheduling algorithms differ from one another in the way of calculating $\wp_i$. At each scheduling time t_{sch}, the algorithm requests the placer to place the task with the highest priority, the task that heads the waiting (ready) queue. If the placer fails to place, it then tries to place the next task in the list and so on, as far as the reconfigurable array is not full. Obviously, this may lead to a situation where a task with a lower priority may be running on the reconfigurable array while a higher priority task is waiting. By the way, the tasks that can no longer respect their deadline constraints are removed from the waiting list. The scheduling algorithms is said *work-conserving* as any area in the array may be kept idle if it could fit a ready task, no matter what its priority is.

A generic pseudo code of parameters-based scheduling algorithm is shown in table 5.1 page 171. There is any details of the placement strategies used. The scheduling algorithms may be combined with placement strategies of various complexity in order to achieve a given overall complexity.

As tasks are released, they are inserted in a waiting or ready list (W) and sorted according to

A pseudo code of the *parameters-based* algorithm

Initialization: $t_{sch} \leftarrow CurrentTime$; $F \leftarrow FinishingTasks(R, t_{sch})$; $F \subset R$;

$A \leftarrow ArrivingTasks(t_{sch})$; $A_{free} \leftarrow$ *free areas*; Area $P \leftarrow 0$; $flag \leftarrow 0$;

$W \leftarrow WaitingTasks(t_{sch})$, the list is sorted according to the considered parameter $\wp$;

$R \leftarrow RunningTasks(t_{sch})$, the list is sorted in increasing finishing time.

Schedule_Param_Based (W, R, A_{free});

```
1.  ∀T_i ∈ F , do                          % dealing with finishing tasks
2.      F ← F − T_i                        % if there is any at time t_sch
3.      update (R, A_free)
4.      flag ← 1
5.  end ∀T_i ∈ F
6.  if (flag)                              % if any task has ended at time t_sch
7.      ∀T_i ∈ A                           % updating the waiting list W
8.          W ← W ∪ T_i                    % W is kept sorted ℘−wise
9.      end ∀T_i ∈ A
10.     ∀T_i ∈ W , A_free ≠ 0              % attempt to place waiting tasks
11.         if (P = place (T_i, A_free))   % if placement successful
12.             R ← R ∪ T_i                % updating the running list R
13.             W ← W − T_i                % updating the waiting list W
14.         end if
15.     end ∀T_i ∈ W
16. else                                   % if placement fails
17.     A ← sort(A, ℘)
18.     ∀T_i ∈ A
19.         if (P = place (T_i, A_free))   % attempt to place just arrived tasks
20.             R ← R ∪ T_i                % if successful, updating R
21.         else
22.             W ← W ∪ T_i                % otherwise, add in W if worthy
23.     end ∀T_i ∈ A
24. end if (flag)
```

Table 5.1: A pseudo code of the *tasks parameters-based* scheduling algorithm

their priority $\wp_i$ in such a way that :

$$W = [T_1, T_2, T_3, ..., T_{n-1}, T_n] \Rightarrow \wp_1 > \wp_2 > \wp_3 > ... > \wp_{n-1} > \wp_n \tag{5.1}$$

for the n tasks currently in the list. As this is a without-looking-ahead scheduling approach, a task that fails to be placed is kept in the waiting queue W for further attempts, as long as it can still meet its deadline. Hence, any rejection occurs at time t_{rej} as expressed earlier in equation 3.11 page 94. The rejected task is removed from the waiting list during its update (table 5.1, line 7). The tasks in the running list (R) are sorted according to their increasing finishing time.

In a uniprocessor system, the microprocessor is assigned to the task with the highest priority which is T_1 in the example above. However, as scheduling hardware tasks on a reconfigurable hardware devices is much more similar to multiprocessor scheduling, several tasks may concurrently run on the device. Hence, the reconfigurable array is allocated to the m tasks that it may accommodate concurrently. A task of a lower priority may be running while another task of higher priority is waiting because there is not enough contiguous free space to place it.

The scheduler is invoked each time t_{sch} a new task is released or a running task completes. In the latter case, the algorithm takes into account the area(s) just freed by the terminated task(s). Consequently, it updates the running tasks list R and the list of free areas A_{free} accordingly (line 1 to 5).

The completion of one or several tasks (marked by a tasks completion flag set to 1) frees new areas on the array. Hence, if there are newly free areas, the newly released tasks are first inserted in the waiting list (line 6 to 9) with respect to their priority. The scheduler then tries to place as many tasks as possible from the waiting list (line 10 to 15).

However, if any task completion has occurred at scheduling time t_{sch} (marked by a tasks completion $flag$ that remains to 0), there is no newly areas freed on the array. Therefore, as the array hasn't changed and cannot fit any task from the waiting list, the scheduler attempts to place only the newly arrived tasks (line 16 to 23). In case of failure, the tasks are inserted in the waiting list for a further attempt (line 22).

As the only difference among priority-driven scheduling algorithms is the method for calculating the tasks priorities, the following sections present some of these algorithms with the corresponding formulas for the priority.

5.2.1 Temporal parameters based scheduling (Basic, EDF, LLF, etc.)

1. **Basic scheduling**

   In *basic scheduling* algorithm, the scheduler maintains two lists of tasks : the waiting list (W) and the running list (R). In the latter, tasks are sorted according to increasing completion times while the waiting list is sorted according to increasing arrival times, or decreasing priority. The priority of each task T_i is given by

   $$\wp_i = \frac{1}{a_i}$$

   a_i being the release time of T_i. Therefore, the n ready tasks currently in the waiting list W are sorted on a first come first served basis. Equation 5.1 above becomes :

   $$W = [T_1, T_2, T_3, ..., T_{n-1}, T_n] \Rightarrow a_1 < a_2 < a_3 < .... < a_{n-1} < a_n$$

   where a_i is the release time of task T_i. The equation suggests that as tasks are released, basic scheduling algorithm tends to start them as soon as possible. Consequently, the waiting time of tasks is minimized.

   The pseudo code of basic scheduling algorithm is detailed in table 5.2 page 174. The tasks lists W and R are maintained sorted as previously described. There are two events that invoke the scheduler : task(s) termination and task(s) release.

   When one or many task terminations arise at a given time t_{sch} (without any task released), one or many areas are consequently freed (line 1 to 5). Tasks in the waiting list are then prioritized. The scheduler through the placer attempts to place them, beginning from the task heading the list. The whole list is then scanned and all the tasks that may fit on the array are placed, as far as the array is not full (line 6 to 13).

   Everytime t_{sch} one or many tasks are released, if there is any task finishing at t_{sch}, the scheduler directly attempts to place the newly arrived task(s) on the array. In case of failure, the task(s) is added at the rear of the waiting list.

   When both events arise simultaneously (task release and task finished), the newly released task(s) is first inserted in the waiting list before any placement attempt.
   The simulation results of this simple scheduling scheme is presented in chapter 6 section 6.3.1 and compared with other scheduling algorithms.

2. **Earliest Deadline First (EDF)**

   As formerly defined in section 3.5.3 page 81, EDF scheduling assigns the highest priority to

A pseudo code of the *basic scheduling* algorithm

Initialization: $t_{sch} \leftarrow CurrentTime$; $F \leftarrow FinishingTasks(R, t_{sch})$; $F \subset R$;

$A \leftarrow ArrivingTasks(t_{sch})$; $A_{free} \leftarrow$ *free areas*; Area $P \leftarrow 0$; $flag \leftarrow 0$;

$W \leftarrow WaitingTasks(t_{sch})$, the list is sorted on a first come first served basis.

$R \leftarrow RunningTasks(t_{sch})$, the list is sorted in increasing finishing time.

Schedule_Basic (W, R, A_{free});

1. $\forall T_i \in F$, do % dealing with finishing tasks
2. $F \leftarrow F - T_i$
3. update (R, A_{free})
4. $flag \leftarrow 1$
5. end $\forall T_i \in F$

6. if (*flag*) % at least one task has ended at time t_{sch}
7. $\forall T_i \in W$, $A_{free} \neq 0$ % dealing with waiting tasks first
8. if (P = place (T_i, A_{free})) % if placement successful
9. $R \leftarrow R \cup T_i$ % R is updated and kept sorted
10. $W \leftarrow W - T_i$
11. end if
12. end $\forall T_i \in W$
13. end if (*flag*)

14. $\forall T_i \in A$, $A_{free} \neq 0$ % dealing with arriving tasks
15. if (P = place (T_i, A_{free})
16. $R \leftarrow R \cup T_i$
17. else $W \leftarrow W \cup T_i$ % W is updated and kept sorted $\wp$-wise
18. end if
19. end $\forall T_i \in A$...

Table 5.2: A pseudo code of the *basic scheduling* algorithm

the task with the closest absolute deadline. Therefore, the n tasks currently in the waiting list W are sorted according to their absolute deadline. Equation 5.1 becomes :

$$W = [T_1, T_2, T_3, ..., T_{n-1}, T_n] \Rightarrow D_1 < D_2 < D_3 < .... < D_{n-1} < D_n$$

where T_1 is the heading task and D_i the absolute deadline of task T_i.
As discussed earlier in section 3.7.1, two variants of global EDF scheduling (denoted as *EDF-First-k-Fit* and *EDF-Next-Fit* respectively) for reconfigurable hardware devices along with their schedulability analysis were proposed by Danne (2006).

In this thesis, the global EDF scheduling is similar to the aforementioned EDF-Next-Fit but applied to online aperiodic and nonpreemptive tasks, and using a 2D placement strategy.

The simulation results are shown and discussed in Chapter 6, section 6.3.1.

3. **Least Laxity First (LLF)**
   LLF scheduling is quite similar to EDF scheduling. LLF scheduling assigns the highest priority to the task with the smallest laxity. Therefore, the n tasks that are currently ready are sorted in a waiting list W according to their laxity. Therefore, equation 5.1 becomes :

$$W = [T_1, T_2, T_3, ..., T_{n-1}, T_n] \Rightarrow l_1 < l_2 < l_3 < .... < l_{n-1} < l_n$$

   where T_1 is the heading task and l_i the laxity of task T_i.
   LLF tends to prioritize the tasks that are closer to miss their deadline.
   The simulation results of the LLF scheduling is shown and discussed in Chapter 6, section 6.3.1

5.2.2 Geometric parameters based scheduling (BSF, SSF, etc.)

1. **Biggest Size First (BSF)**
   In BSF scheduling policy, the bigger the size of a task, the higher its priority. Priority of each task T_i is given by :

$$\wp_i = w_i \cdot h_i$$

In the case of equal size tasks, the task with the higher aspect ratio is assigned a higher priority. Hence, let T_i and T_j be two tasks:

$$w_i \cdot h_i = w_j \cdot h_j \Rightarrow \begin{cases} \wp_i > \wp_j \text{ if } \frac{h_i}{w_i} \geq \frac{h_j}{w_j} \Rightarrow \wp_i = \wp_j + \epsilon \\ \wp_i < \wp_j \text{ if } \frac{h_i}{w_i} < \frac{h_j}{w_j} \Rightarrow \wp_i = \wp_j - \epsilon \end{cases} \tag{5.2}$$

2. **Smallest Size First (SSF)**

   In opposition to BSF, SSF gives priority to smaller size tasks. Priority of each task T_i is then given by

   $$\wp_i = \frac{1}{w_i \cdot h_i}$$

   In the case of two tasks T_i and T_j with equal size the task with a higher aspect ratio is assigned a higher priority, as expressed in equation 5.2 above.

Intuitively, BSF will provide a better reconfigurable array utilization ratio compared to SSF, as it places bigger tasks first. Contrary to BSF, SSF increases the array fragmentation by placing smaller tasks on available areas that may fit bigger tasks.
The simulation results of the BSF and SSF scheduling are shown and discussed in Chapter 6, section 6.3.1.

5.2.3 Combining Geometric and Temporal parameters for scheduling

In this scheduling policy, the priority of each task is calculated using both geometric and temporal parameters of the task. *Classified Stuffing* (Chen and Hsiung, 2005) is one example of such scheduling algorithms. In Classified Stuffing (CS), a task is placed on either the leftmost or the rightmost of the reconfigurable array depending on its space utilization rate. The latter being the ratio between the width and the execution time of the task. CS is an improvement of normal 1D stuffing algorithms (Steiger et al., 2004). Both algorithms use a 1D placement, which means that the height of the task is meaningless and is not taken into account in the priority assignment. Section 3.7.2 in Chapter 3 provides more details of the CS algorithm and the 1D normal stuffing algorithm.

The coming section presents the *computational load based* scheduling algorithm. The algorithm combines geometric and temporal parameters and uses a 2D placement.

Computational load based scheduling

The algorithm is based on the absolute computational load of the task. The priority of each

task is given by :

$$\wp_i = w_i \cdot h_i \cdot e_i$$

which corresponding to the total amount of resources needed to complete the task. Tasks are sorted according to decreasing computational loads in a ready tasks list W, as shown in the following equation.

$$W = [T_1, T_2, T_3, ..., T_{n-1}, T_n] \Rightarrow w_1 \cdot h_1 \cdot e_1 > w_2 \cdot h_2 \cdot e_2 > .... > w_n \cdot h_n \cdot e_n$$

Hence, the algorithm schedules resources greedy tasks first. As in other priority-driven scheduling policies presented above, when many tasks share the same priority, the task with the highest aspect ratio is assigned the highest priority. This algorithm is likely to achieve a higher reconfigurable array utilization ratio, as tasks that use more resources are placed first.

5.3 Slots-based Scheduling

Slots-based scheduling relies on the partitioned scheduling presented in section 3.6.4. The reconfigurable array is partitioned into slots as shown in Chapter 3, figure 3.8. The number and the size of each slot may be either predetermined or dynamically changed depending on the size of arriving tasks. This resulted in many slots-based scheduling algorithms that are presented below.

5.3.1 *n X 1D* variable size slots scheduling

In *n X 1D* variable size slots scheduling, the reconfigurable array is 1D-partitioned, n being the maximum number of partitions. The size of the partitions depends on the size of the arriving tasks. Hence, the size of the first partition is fixed by the size of the first task. A simple 1D placer is used in each slot. The placer keeps a list of partition that are sorted according to a given criteria. For example, in *load-balanced 1D variable slots scheduling*, the slots are sorted according to the increasing loads. Hence, any new job is scheduled on least loaded slot. This thesis refers to this scheduling as *1D variable size slots scheduling*.

Figure 5.1 depicts three variants of a 1D variable size slots scheduling. This scheduling approach relies on the simplicity of the 1D placement used in each slot. The resulting internal fragmentation is drastically reduced compared to a traditional (non partitioned) 1D placement. Let Γ_6 be a set of six tasks to online schedule on the reconfigurable array using 1D variable size slots scheduling, as mapped in figure 5.1. Tasks in Γ_5 are sorted according to their increasing release times a_i as expressed below, and are placed on the reconfigurable array as they are released.

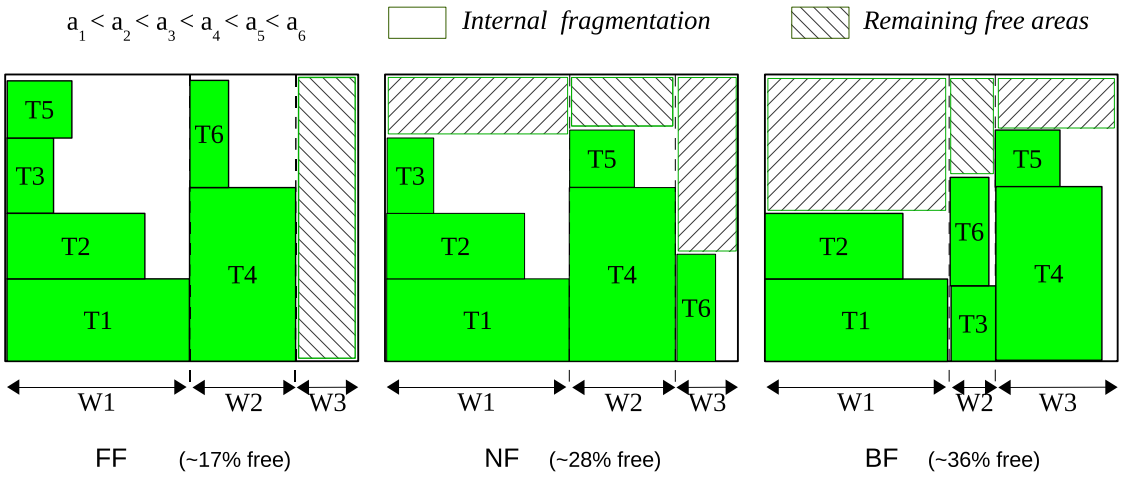


Figure 5.1: 1D-like partitioned scheduling

$$\Gamma_6 = [T_1, T_2, T_3, T_4, T_5, T_6] \Rightarrow a_1 < a_2 < a_3 < a_4 < a_5 < a_6$$

The size of the reconfigurable array is $A_{fpga} = W \cdot H$ where W and H are respectively the width and the height of the device. T_1 is the first task released and is placed on the bottom left of the reconfigurable array. A first slot denoted as W_1 is created, which width is equal to the width of T_1. At this stage, there are two slots on the device, W_1 and $W_2 = W - W_1$, not detailed on the figure. Therefore, different fitting strategies may be used : First Fit, Next Fit and Best Fit.

1. **First Fit (FF)** strategy places the next task in the first available slot that can fit the task. As shown on the left of figure 5.1, when tasks T_2 and then T_3 are arrive, they are placed in the first available slot, W_1. T_4 is released and cannot fit in W_1. Therefore, a second slot W_2 is created, which width is equal to the width of T_4. There are currently 3 slots on the reconfigurable array, W_1, W_2 and $W_3 = W - W_1 - W_2$. The next task T_5 is released and placed on the first slot that can fit it, which is slot W_1. Task T_6 arrives at last and is placed in the first slot that can accommodate it, the slot W_2. The 3^{rd} slot (hatched) is free.

2. **Next Fit (NF)** strategy places the current task on the next available area that may fit the task, as mapped in the middle of figure 5.1. Next Fit schedules tasks T_1, T_2 and then T_3 similarly to FF described above. The three first tasks are placed in the first slot W_1, as the latter is wide enough to fit them. When T_4 arrives, the scheduler checks the next available area that can accommodate the task. As the first slot cannot accommodate it, a slot of the same width as T_4 is generated, T_4 is placed. At this stage, there are 3 slots on the array

and fitting solutions for FF and NF are similar. Contrary to FF strategy, when T_5 arrives, NF assigns the second slot W_2 to it, instead of the first slot. Indeed, NF checks the area that is directly next to the previous area where the last task (T_4 here) has been placed. T_6 is released and placed in the next available area, which is in the 3^{rd} slot.

3. **Best Fit (BF)** always selects the area that fit the best and that reduces internal fragmentation. The algorithm selects among all possible fitting areas, the area which size is closer to the task size. This chosen area may be either in an existing slot, or in a newly generated slot which width will be equal to the width of the task. An example of a BF placement is pictured on the far right of figure 5.1. T_1 and T_2 are placed similarly to FF and NF placements described above. However, contrary to FF and NF, BF places T_3 in a newly generated slot W_2 that fits it the best and that minimizes the internal fragmentation. Indeed, FF and NF placements depicted respectively on the far left and the centre of figure 5.1 fit T_3 and T_5 in the first slot, inducing a lost of space.

After placing the six tasks using each of the 3 fitting strategies, there are two types of remaining areas :

- the remaining free areas (hatched areas) that are available and can accommodate new tasks.
- the internally fragmented areas (white areas) that are wasted as long as the neighbouring tasks that brought them out have not completed (e.g. the areas next to T_3, T_4 or T_5 in FF placement are lost as these tasks do not occupy the entire width of the slot accommodating them).

Figure 5.1 shows that the BF fitting strategy produces more free areas and less internal fragmentation compared to FF and NF. The remaining available free area represents $\sim$ 17% for FF strategy, $\sim$ 28% for NF strategy and $\sim$ 36% for BF strategy. The latter strategy increases the reconfigurable device utilization ratio accordingly.
A pseudo code of the 1D variable slots scheduling algorithm is presented in table 5.3, page 180. Lines 1 to 5 deal with tasks termination. If any has occurred, lines 6 to 14 update the slots and attempt to place the waiting tasks if there is any. Finally, lines 15 to 21 deal with arriving tasks. The placement (lines 8 and 16) are done using one or another of the fitting strategies above.
In the coming section, *n X 1D variable size slots scheduling* is combined with the *looking-ahead* scheduling approach.

A pseudo code of the *1D variable slots* scheduling algorithm

Initialization: $t_{sch} \leftarrow CurrentTime$; $F \leftarrow FinishingTasks(R, t_{sch})$; $F \subset R$;

$A \leftarrow ArrivingTasks(t_{sch})$; $A_{free} \leftarrow$ *free areas*; Area $P \leftarrow 0$; $flag \leftarrow 0$;

$W \leftarrow WaitingTasks(t_{sch})$, the list is sorted on a first come first served basis.

$R \leftarrow RunningTasks(t_{sch})$, the list is sorted in increasing finishing time.

Schedule_1D_Var_Slots_Based (W, R, A_{free});

1. $\forall\ T_i \in F$, do % dealing with finishing tasks
2. $F \leftarrow F - T_i$
3. $update(T_i, A_{free} \rightarrow slot_i)$
4. $flag \leftarrow 1$
5. end $\forall\ T_i \in F$

6. if (*flag*) % if at least one task has ended at time t_{sch}
7. $\forall\ slot_i \in A_{free}$, do % merging empty slots if exist
8. if$(slot_i \rightarrow empty)$ $merge\_slots\_if\_worthy(\ )$
9. end $\forall\ slot_i \in A_{free}$
7. $\forall\ T_i \in W$, $A_{free} \neq 0$ % dealing with waiting tasks first
8. $if(P = place(T_i,\ A_{free}))$ % if placement successful
9. $R \leftarrow R \cup T_i$ % R is updated and kept sorted
10. $W \leftarrow W - T_i$
11. $update(T_i, A_{free} \rightarrow slot_i)$ % updating the slot
12. end if
13. end $\forall\ T_i \in W$
14. end $if(flag)$

15. $\forall\ T_i \in A$, $A_{free} \neq 0$ % dealing with arriving tasks
16. $if(P = place(T_i, A_{free})$
17. $R \leftarrow R \cup T_i$
18. $update(T_i, A_{free} \rightarrow slot_i)$
19. else $W \leftarrow W \cup T_i$ % W is updated and kept sorted $\wp$-wise
20. end if
21. end $\forall\ T_i \in A$...

Table 5.3: A pseudo code of the *1D variable slots scheduling* algorithm

5.3.2 1D variable slots looking-ahead scheduling

Chapter 3 has presented the *looking-ahead* scheduling along with its advantages and drawbacks. The main advantage was the ability for the scheduler to know as soon as a task arrives, whether it may fit in the reconfigurable array at current time or later on. However, this rapid decision came at the cost of numerous placement and area management operations required to mimic future states of the reconfigurable array. This cost may be even higher if optimal area management strategies (e.g. MERs-based) are used.
The methodology presented in Chapter 4 first made accurate measurements of runtime overheads of MERS-based area management on a real embedded processor. Based on these measurements, different trade-offs between the scheduling scheme and the underlying placement strategy were suggested . Hereinafter is an algorithm that results from a combination of a looking ahead scheduling algorithm and an 1D slots based placement strategy. Any other possible combination hasn't been discussed.

1D variable size slots horizon (1D-VSSH)

1D variable size slots horizon (1D-VSSH) scheduling combines the advantages of a *looking-ahead scheduling* (especially the *horizon scheduling*, Steiger et al., 2004) with the simplicity of a *1D variable size slots scheduling/placement* presented in the former section. The 1D horizon scheduling (Steiger et al., 2004) is used in each slot. Following the same principle, a *stuffing* scheduling (Steiger et al., 2004) or an improved stuffing (e.g. Chen and Hsiung, 2005) may also be used.

Figure 5.2 illustrates a case where a set of 6 tasks $\Gamma_6 = [T_1, T_2, T_3, T_4, T_5, T_6]$ are scheduled on a 10X6 reconfigurable array using *1D variable size slots horizon scheduling*. Parameters of the tasks are detailed in table 5.3.2. Each task is submitted to a deadline constraint. The number and the size of slots are adjusted at runtime depending on the size of released tasks.

Table 5.4: Tasks parameters for *1D variable size slots looking-ahead* scheduling $t_D = max(d) = 18$

Tasks parameters	T_1	T_2	T_3	T_4	T_5	T_6
a : arrival time	1	1	3	3	5	8
e : execution time	8	7	10	6	5	4
d : deadline	10	10	17	17	17	18
latest starting time	2	3	7	11	12	14
w : width of the task	3	5	6	3	2	7
h : height of the task	5	4	2	3	3	3

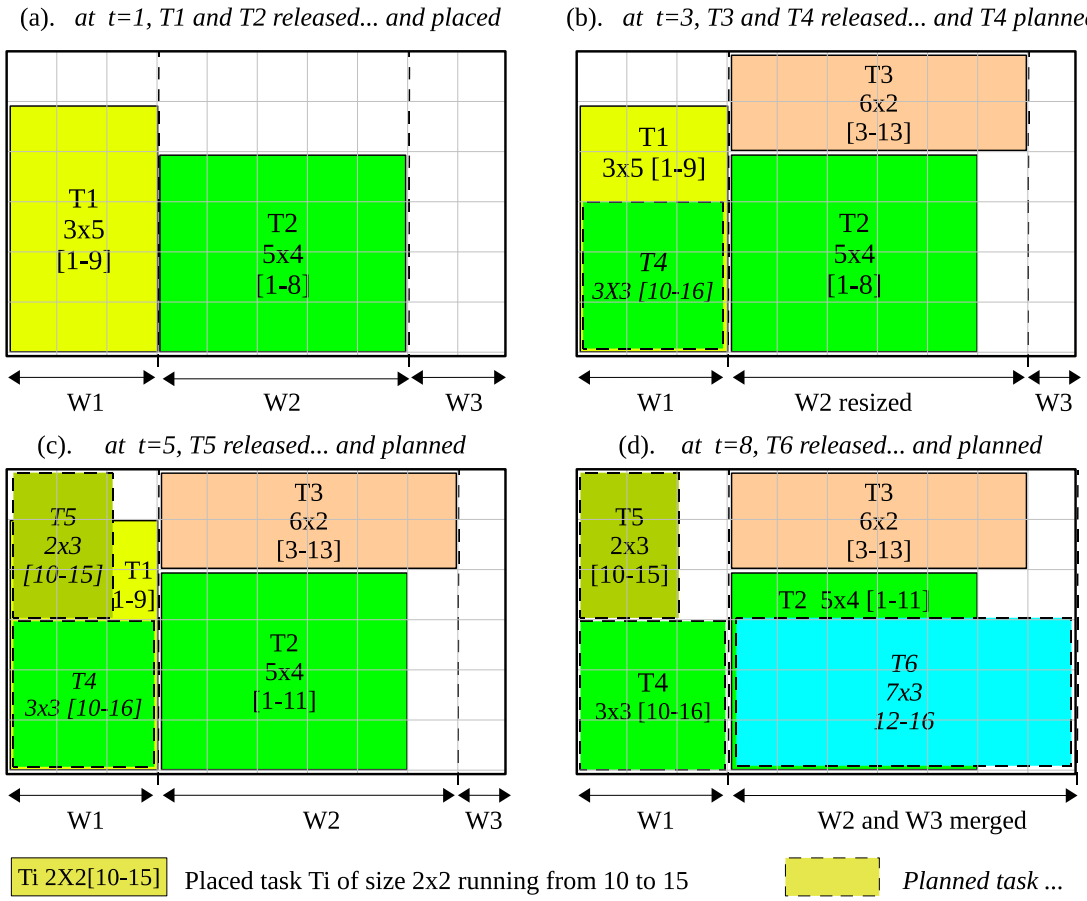


Figure 5.2: 1D improved horizon scheduling algorithm, also denoted as *1D variable size slots horizon* (1D-VSSH)

Horizon scheduling algorithm schedules a task either on the currently available areas, or prospects future states of the array in order to see if there is any area that may accommodate the task in a the future and run it to completion without missing its deadline.

At time $t = 1$, T_1 and T_2 are released and placed in the reconfigurable array (figure 5.2(a)). The width of the first slot W_1 is fixed by the width of T_1. T_2 is placed in the second slot W_2. W_2 and W_3 are floating size slots, as they may be resized. However, the size of W_1 is bounded as far as W_2 hosts at least one task.

At time $t = 3$, T_3 and T_4 are released (figure 5.2(b)). W_2 is resized in order to fit T_3. At current time, there is no place to fit task T_4. However, T_4 may start at time $t = 11$ latest without missing its deadline. As task T_1 will complete at $t = 9$, the area that will be freed may fit T_4. T_4 is

planned to start at time $t = 10$ at the position currently occupied by T_1. Therefore, T_1 and T_4 overlap on the figure.

At time $t = 5$, T_5 arrives and cannot fit in any currently available area. The latest starting time for it is $t = 12$ otherwise it will violate its time constraint. Consequently, slot W_1 may accommodate T_5 in addition to T_4 at time $t = 10$ after the completion of T_1. T_5 is planned as shown in figure 5.2(c).

At time $t = 8$, T_6. Following the same principle, T_6 is planned to start at time $t = 12$ after T_2 completes. However, slots W_2 and W_3 are resized in order to fit T_6.

The *relative application (computational) load* of the tasks set Γ_6 is given by

$$res_{\Gamma_5} = \frac{\sum_{i=1}^{k} w_i \cdot h_i \cdot e_i}{W \cdot H \cdot t_D} = \frac{3 \cdot 5 \cdot 8 + 5 \cdot 4 \cdot 7 + 6 \cdot 2 \cdot 10 + 3 \cdot 3 \cdot 6 + 2 \cdot 3 \cdot 5 + 7 \cdot 3 \cdot 4}{10 \cdot 6 \cdot 18} = 50.7\%$$

It reflects the amount of resources required by Γ_6 relative to the total amount of resources available on the reconfigurable array during t_D time units, t_D being the absolute deadline of Γ_6.

The utilization ratio of the reconfigurable array resulting from the current scheduling policy (1D variable slots horizon-looking-ahead) on Γ_6 is given by :

$$U_{fpga(\%)} = \frac{\sum_{i=1}^{6} w_i \cdot h_i \cdot e_i}{W \cdot H \cdot mk} = \frac{3 \cdot 5 \cdot 8 + 5 \cdot 4 \cdot 7 + 6 \cdot 2 \cdot 10 + 3 \cdot 3 \cdot 6 + 2 \cdot 3 \cdot 5 + 7 \cdot 3 \cdot 4}{10 \cdot 6 \cdot 16} = 57.1\%$$

mk is the makespan or the schedule length. It corresponds to the latest finishing time which is equal to 16 in the above case. W and H are respectively the width and the height of the reconfigurable array.

Simulation results for *1D variable size slots horizon looking-ahead* scheduling is shown and explained in Chapter 6, section 6.5.2.

5.3.3 1D variable slots scheduling with minimum makespan

The makespan is one of the most used objective function. Indeed, most of the scheduling costs are highly tight to the amount of time spent by a given user (or application) on the computing resources. Consequently, the makespan must be reduced. However, as shown in Chapter 3 section 3.4.3, there is no optimal solution in online scheduling, and performance analysis is done through *average-case analysis* or *worst-case analysis* (e.g. competitive analysis).

The flow time of each job may affect the makespan. Hence, the reduction of the total flow time may be considered. As expressed in equation 4.20 page 162, the flow time of a job is the time spanning from its release to its completion.

Minimizing the total flow time of a job sequence Γ_n is equivalent to minimizing the average flow time which has been also expressed in equation 4.21, page 162. In multiprocessor scheduling, one way of minimizing the total flow time is to give higher priority to jobs with longest execution time. The reason is that the sooner the jobs with long processing time start, the sooner they complete. This is even more important in online scheduling. Jobs with longer execution time should not have to wait too long before starting, as they would likely increase the flow time of the task, and therefore the makespan. An example is shown on the left of figure 5.3 where a late start of task T_{21} drastically lengthens the makespan. From a multiprocessor scheduling perspective, figure 5.3 depicts the worst case and the optimal case of a list scheduling of n tasks on m identical processors. The case is transposed in reconfigurable hardware scheduling domain.

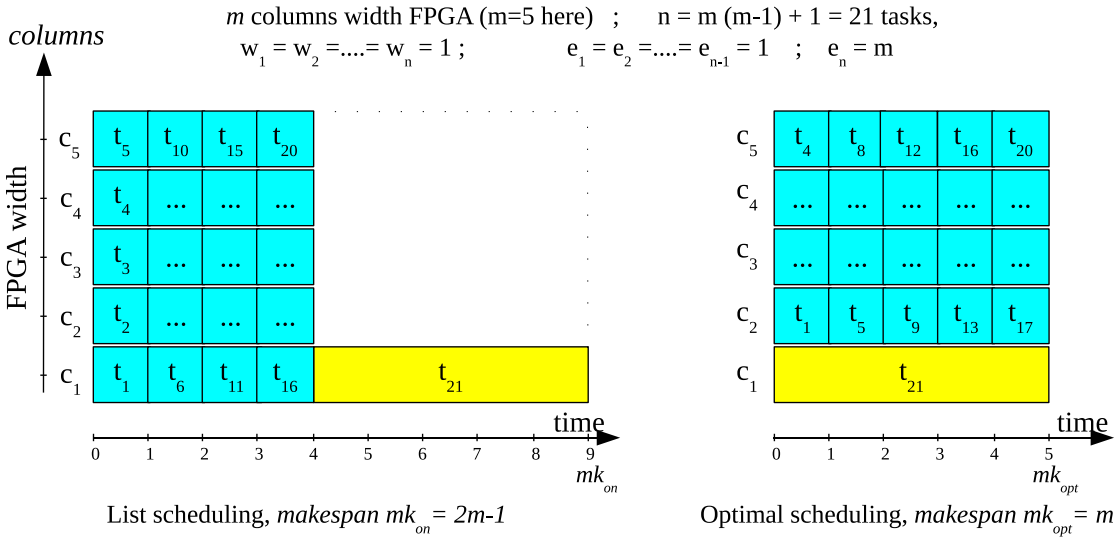


Figure 5.3: List scheduling *vs* optimal scheduling of n tasks on m identical processors; e_i is the execution time of task T_i

Let $\Gamma_n = [T_1, T_2, ..., T_n]$ be a list of n hardware tasks of width $w_1 = w_2 = ... = w_{n-1} = w_n = 1$ and processing times $e_1 = e_2 = ... = e_{n-2} = e_{n-1}$ and $e_n = m$. Let us assume that jobs in Γ_n are presented one by one to the scheduling algorithm. As soon as a job t_i is scheduled, the next job t_{i+1} in the list is available for scheduling. This may correspond to *jobs arrive over list* or *jobs arrive over time* online paradigm.

Let $FPGA_m$ be an $m-columns$ reconfigurable hardware device where m is the width of the device and $n = m \cdot (m-1)$. In order to schedule Γ_n on $FPGA_m$, let us assume that either the scheduler

uses a 1D placer or the height of each hardware task in Γ_n spans the entire height of $FPGA_m$. Therefore, $FPGA_m$ can concurrently fit m jobs. The list scheduling algorithm schedules any new job on least loaded processor (here the less loaded column). As depicted on the left of figure 5.3, the first job t_1 is placed in the first column c_1 at time 0, the second job t_2 on the second column c_2 and so on. The resulting makespan is $mk_{on} = 2m - 1$. Indeed, as the scheduler learns about tasks one by one, it cannot properly plan the arrival of tasks with longer execution time like T_{21}. Consequently, at the end, the final load of each column of the reconfigurable array is very unbalanced, leading to a higher makespan and a lower average utilization ratio. This special case of the left of figure 5.3 has been chosen to be a worst case scenario.

The right of figure 5.3 shows how optimal would have been an offline scheduling of the same tasks set Γ_n on the same reconfigurable device $FPGA_m$. Indeed, if we have a priori knowledge of the tasks in Γ_n, they can be optimally dispatched in different columns of the device, as shown by the right of figure 5.3. The resulting makespan is optimal and lowered to $mk_{opt} = m$.

As stated in Chapter 3 and expressed by equation 5.3, an online scheduling algorithm (e.g. list scheduling here) is said $c-competitive$ if for any input instance Γ_n, the objective function (e.g. the makespan mk_{on} in figure 5.3-left) produced by the algorithm on Γ_n is at least c times better than that obtained with the optimal offline scheduling, as shown in figure 5.3-right. This is expressed by

$$mk_{on} \leq c \cdot mk_{opt} \tag{5.3}$$

c is known as the competitive ratio of the online scheduling algorithm, and is expressed as follows:

$$c = \frac{mk_{on}}{mk_{opt}} = 2 - \frac{1}{m} \tag{5.4}$$

In the special case of figure 5.3 where $w_i = 1$, if we assume that each hardware task in Γ_n spans the entire height of the reconfigurable device, the utilization ratio in online scheduling and in offline optimal scheduling are respectively

$$U_{on(\%)} = \frac{\sum_{i=1}^{n} e_i}{m \cdot mk_{on}} \qquad \text{and} \qquad U_{opt(\%)} = \frac{\sum_{i=1}^{n} e_i}{m \cdot mk_{opt}}$$

where $U_{on(\%)} \leq U_{opt(\%)}$. Therefore, equation 5.3 becomes:

$$mk_{on} \leq (2 - \frac{1}{m}) \cdot mk_{opt} \tag{5.5}$$

As emphasized in equation 5.5, equation 5.4 guarantees the fact that the makespan of the online algorithm will never be beyond $(2-\frac{1}{m}) \cdot mk_{opt}$. The latter equation is also known as the *competitive ratio* of the *Graham's online list scheduling problem* (Graham et al., 1979) where a sequence of jobs has to be scheduled on m identical parallel processors in a way that the makespan is minimized.

A pseudo code of the *1D variable slots with minimum makespan* scheduling algorithm

Initialization: $A_{free} \leftarrow$ *free areas*; Area $P \leftarrow 0$; $flag \leftarrow 0$; $t_{sch} \leftarrow CurrentTime$;
$F \leftarrow FinishingTasks(R, t_{sch})$; $F \subset R$; $A \leftarrow ArrivingTasks(t_{sch})$; $R \leftarrow RunningTasks(t_{sch})$;
$W \leftarrow WaitingTasks(t_{sch})$, the list is sorted in decreasing processing time.
$A_{free} = \{C_1, C_{long}\}$, $C_1 = 1^{st}$ cluster, $C_{long} = 2^{nd}$ cluster for tasks with long processing time.
$Temp \leftarrow empty$, temporal list of tasks to be placed, sorted in decreasing processing time;
$e_{long} \leftarrow \alpha \cdot e_{max}$, long processing time threshold, e.g. $\alpha = 0.5$;
Schedule_1D_slots (W, R, A_{free}) ;

1. $\forall\ T_i \in F$, do % dealing with finishing tasks
2. $F \leftarrow F - T_i$
3. $update(R, A_{free} \rightarrow C_i)$ % updating the corresponding cluster
4. $flag \leftarrow 1$
5. end $\forall\ T_i \in F$
6. $if\ (flag)$ % if at least one task has ended at time t_{sch}
7. $Temp \leftarrow W \cup A$ % $Temp$ contains W and A.
8. $\forall\ C_i \in A_{free}, C_i \neq C_{long}$, do % merging empty slots if exist
9. if $(C_i \rightarrow empty)$ $merge\_clusters\_if\_worthy(\ )$
10. end $\forall\ C_i \in A_{free}$
11. $else$ $Temp \leftarrow W$ % $Temp$ contains only W
12. end $if(flag)$
13. $\forall\ T_i \in Temp$, $A_{free} \neq 0$ % dealing with waiting tasks
14. if $(T_i \rightarrow exec\_time \geq e_{long})$ $P = place(T_i\ , C_{long})$
15. if $(notP)$ $P = place(T_i\ , C_{i \neq long})$
16. if (P) % if placement successful
17. $R \leftarrow R \cup T_i$ % R is updated and kept sorted
18. $Temp \leftarrow Temp - T_i$
19. $update\_clusters(R, A_{free} \rightarrow C_i)$
20. end if (P)
21. end $\forall\ T_i \in Temp$
22. $W \leftarrow W \cup Temp$ % W is updated and kept sorted, $Temp$ emptied

Table 5.5: A pseudo code of the *1D variable slots with minimum makespan* algorithm

5.4 Placement Strategies for 2D Looking-Ahead Scheduling

In Chapter 4, MERs-based optimal area management strategies have been used to refine the methodology, and to refer to as a comparison reference. According to the same methodology, non optimal area management and placement strategies, combined on one hand with *looking-ahead* scheduling algorithms, and on the other hand with multi-shape tasks scheduling were adopted.
In most of designed scheduling algorithms, the placer manages the areas mainly through a binary search tree as described in Bazargan et al. (2000) and detailed in Chapter 3, section 3.8.3. The time complexity for finding a given node in the tree is $O(n)$ in the worst case, n being the number of tasks running on the FPGA. The complexity drops to $O(log_2(n))$ if the tree is balanced. By default, a 2D placer is used, unless a 1D placer is clearly mentioned in the name of the scheduling algorithm.
In some cases, the hash matrix presented by Walder et al. (2003) was used. The tree and the hash matrix (if used) are updated at each task insertion or deletion. The matrix stores free areas and allows the placer to find a feasible placement in constant time complexity $O(1)$. However, updating the matrix requires a potential scan of $w \cdot h$ entries, where w and h are respectively the width and the height of the area inserted in or deleted from the matrix. Walder et al. (2003) show that the number of real scans is one order of magnitude lower than that.

No matter if the matrix is used or not, a scheduling algorithm is comparable to another only if both are using the same placement strategy. This ensures that any improvement will be to scheduler's credit. In this chapter, there is no more details of the placement strategies, as they were formerly described in Chapter 3 and detailed in the papers of the authors cited above.
In the coming section is proposed a ternary tree structure suitable for *looking-ahead scheduling*.

5.4.1 A Ternary Tree structure for Looking-Ahead Scheduling

This section presents a ternary search tree structure enabling *looking-ahead* scheduling of real-time tasks on partially reconfigurable FPGAs. The structure suits to this scheduling approach, as it provides a good overview of present and future states of the reconfigurable array.
As previously stated, *looking-ahead* scheduling schedules each task once at its arrival and immediately rejects or accepts it. The accepted task may start instantaneously or later and still meet its deadline. This rapid scheduling decision makes *looking-ahead* scheduling of vital interest for online real-time systems. Indeed, an immediate task rejection gives to the OS the opportunity for finding alternative resources implementation other than the reconfigurable array.

Through the literature review in Chapter 3 and the methodology in Chapter 4, it was also stated that the reconfigurable array utilization ratio, the task rejection ratio and the scheduling algorithm complexity highly depend on the underlying placement strategies used. Consequently, using a *looking-ahead* scheduling is worthy in an online real-time context only when combined with low complexity free area management strategies.
In this section, the *looking-ahead* scheduling is combined with a low complexity 2D placement strategies in order to lower the overall scheduling/placement runtime overheads. This placement strategy relies on the binary search described in Chapter 3 section 3.8.3. The tree has been introduced by Bazargan et al. (2000) and improved by Walder et al. (2003). It stores the state of the reconfigurable array along the scheduling process. In this thesis, a ternary tree suitable for *looking-ahead* scheduling was derived from the binary tree.

Managing the tree : example of two horizon scheduling algorithms

The ternary tree structure is a variants of the binary tree. Storing the states of the array in the latter structure makes merging and splitting operations more intuitive, as detailed earlier in figure 3.26 page 128. The same principle is applied to the ternary tree. The time complexity for finding a node in the structure is bounded by $O(n)$, n being the number of tasks currently placed in the array.

Figure 5.4 pictures splitting and merging processes that result from the scheduling of a set of tasks on the reconfigurable array. Let $\Gamma_6 = [T_1, T_2, T_3, T_4, T_5, T_6]$ be a set of online real-time hardware tasks to schedule. Parameters of Γ_6 are shown in table 5.4.1.

As shown in figure 5.4 (i), (ii), and (iii), when a task is placed in an area, the area is divided in three rectangles according to vertical, horizontal or overlapping split . In the binary tree presented in Bazargan et al. (2000) and Walder et al. (2003), each node represents an area which can generate up to two children nodes when a task is placed. A third node is generated only in some special cases. In the ternary tree proposed in this thesis and depicted in figure 5.4, a third area node of the size of the task placed ((a) and (a')) is systematically generated. Each node stores the size and the time availability of an area. One example of a set of tasks to be scheduled is given in table 5.4.1. The scheduling algorithm learns about tasks as they are released. According to figure 5.4, the entire resources of the FPGA of size 10x6 is available at the beginning. Therefore,

- At time $t = 1$, tasks T_1 and T_2 are released in the system. They are immediately placed, as there is enough place to fit them (see a and b). Children nodes are generated accordingly

Table 5.6: Example of tasks parameters for *horizon-SFAF* and *horizon-EAAF* scheduling algorithms

$t_D = max(d) = 18$

Tasks parameters	T_1	T_2	T_3	T_4	T_5	T_6
a : arrival time	1	1	3	3	5	8
e : execution time	8	7	10	6	5	4
d : deadline	10	10	17	17	17	18
latest starting time	2	3	7	11	12	14
w : width of the task	3	5	6	3	2	7
h : height of the task	5	4	2	3	3	3

(see a' and b'). The information in the root node (b') shows that the whole FPGA will be available at time 18 while the child node 7x6 hosts the area 7x6 occupied during the time interval $[1, 8]$.

- At time $t = 3$, T_3 arrives and cannot fit in any currently available area (nodes 3x1, 5x2 and 2x6 on (b')). However, T_3 can fit in node 7x6 and still meet its deadline. Thus, T_3 is planned to start at time $t = 8$.
- At time $t = 5$, T_4 arrives. There are two scheduling options :
  1. the ***horizon-EAAF***
     this fitting approach assigns the *Earliest Available Area First* to T_4. The node that is available sooner than any other node capable of accommodating task T_4 is the node 6x4, available at time $t = 8$. This fitting strategy may correspond to first fit, from a temporal point of view.
  2. the ***horizon-SFAF***
     this approach assigns the *Smallest Fitting Area First* to T_4. The smallest node among all the node that can fit the task is the node 3x5, available at time $t = 9$). The node corresponds to the area that fits the best, similar to best fit fitting strategy.

Intuitively, *horizon-SFAF* should outperform *horizon-EAAF* in terms of reconfigurable chip utilization ratio, as it relies on a best fit approach. For the sake of simplicity, these two variants of horizon scheduling algorithms are sometimes denoted as *SFAF* and *EAAF* respectively.

The simulations results of the *horizon-SFAF* and the *horizon-EAAF* horizon scheduling using the ternary tree scheduling are presented and discussed in Chapter 6, section 6.5.1. The two variants are compared with two tasks parameters based scheduling algorithms (EDF scheduling and the Basic scheduling).

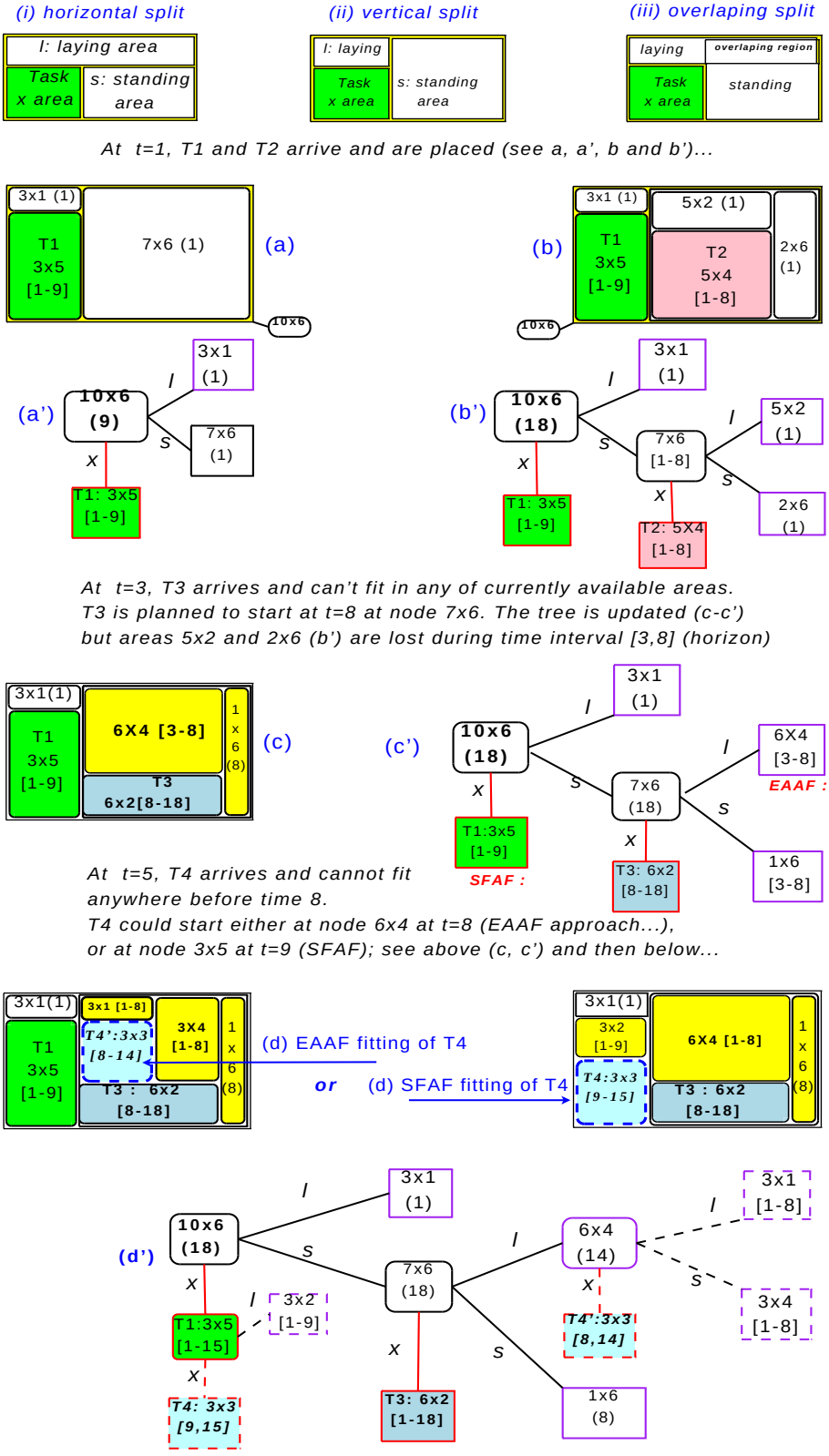


Figure 5.4: Ternary tree structure : splitting and updating processes

5.5 Multi-shape based Tasks Scheduling

Multi-shape based scheduling is a scheduling algorithm that assumes that each hardware task may have more than one versions that differ from each other by their shape, size and/or processing time. Such tasks is denoted as *multi-shape tasks*, and the corresponding scheduling/placement algorithms as *multi-shape based*.
When designing a hardware task, there are mainly two stages at which the designer can influence its shape and its size :

1. ***Prior to the synthesis phase***
   by using various arithmetic implementation techniques. The techniques range from bit-serial to fully parallel and therefore provide a range of options and compromises between the size of each task and its temporal characteristics. *Distributed Arithmetic* is a well-known example of such a technique. A task may have several execution times (or throughput) depending on whether it has been implemented serial, semi-parallel or fully parallel. Variants of the same task differ from each other by their size and the corresponding execution time. One example is depicted in figure 5.5 where task T_3 is a multi-shape task with a normal variant and some smaller variants (*smaller_standing* and *smaller_laying*) with a longer execution time.

2. ***After the synthesis phase***
   where Place & Route tools may be used to constrain a designed module in a rectangular region. This relies on the module-based design for partial reconfiguration described earlier in Chapter 2 section 2.5.8. Such a region-constrained Place & Route may slightly change temporal characteristics of the module or task (e.g. the highest operation frequency deduced from the longest path in the module layout). However, it is assumed in this thesis that execution time of a task remains slightly the same for slightly the same amount of configurable resources. Figure 5.5 depicts an example where task T_3 is a multi-shape task with a normal variant and some variants denoted as *same_standing* and *same_laying* that use the same amount of configurable resources for the same execution time, but are of different width and height.

Therefore, designing *multi-shape* tasks comes at the cost of an extra effort at design time. In addition, memory requirements for storing modules bitstreams increase linearly with the number of variants per task. As additional versions of hardware tasks tend to be *smaller* versions that

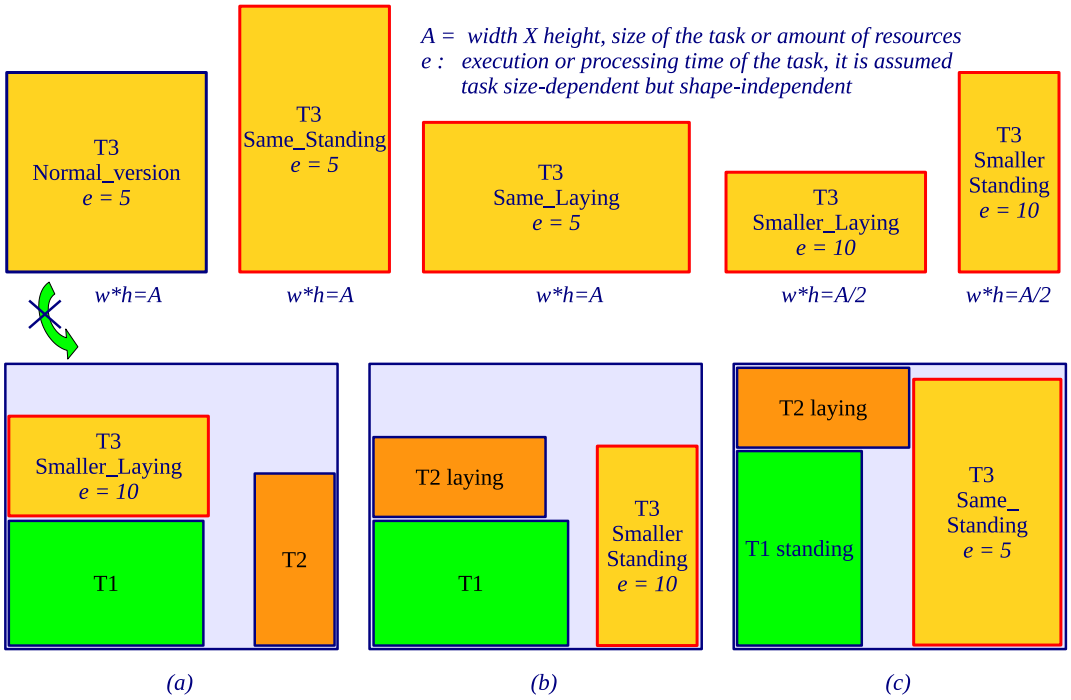


Figure 5.5: Multi-shape tasks provides more fitting opportunities (e.g. T_3 provides 5 variants).

minimize the total amount of configurable resources, memory requirement is bounded by $O(n)$ where n is the number of variants per task.

5.5.1 Raison d'être for multi-shape tasks

As stated in the previous chapter, a way of improving the scheduling/placement quality without increasing the algorithm complexity and runtime overhead is to generated hardware tasks that feature more than one rectangular shape. *Multi-shape based scheduling* is suitable for applications with hardware tasks that have more than one implementation version on the reconfigurable device, as depicted in figure 5.5. The top of the figure depicts an example of a multi-shape task with 5 versions. The bottom of the figure depicts a placement scenario of such tasks. The *normal* version of task T_3 couldn't fit on the reconfigurable array in any of the three cases illustrated in the bottom of figure 5.5. Indeed, after placing any of the two versions of tasks T_1 and T_2, the remaining area is not big enough to fit the normal version of T_3. (a), (b) and (c) map three placement alternatives. In the example, T_3 provides versions that use less resources (smaller versions) in addition to version that use the same amount of resources (same size versions).

Let $T_i = \{T_{i1}, T_{i2}, ..., T_{in}\}$ be an *n-versions* multi-shape hardware task where T_{ij} is the j^{th} version of task T_i . The ratio in equation 5.6 gives a simplified relation between size and execution time of the n different versions of task T_i.

$$\forall T_{ij}, T_{ik} \in T_i, \qquad \frac{A_{ij}}{A_{ik}} = \frac{w_{ij} \cdot h_{ij}}{w_{ik} \cdot h_{ik}} = \frac{e_{ik}}{e_{ij}} \tag{5.6}$$

where $A_{ij} = w_{ij} \cdot h_{ij}$ is the size of the j^{th} version of task T_i (resp. e_{ij} is the execution time of the j^{th} version of task T_i). For example, task T_3 in figure 5.5 has a normal version and two half sized versions (*smaller laying* and *smaller standing*) with an execution time that is twice longer.

Scheduling multi-shape hardware tasks does not really increase the algorithm complexity and runtime overhead. The many the versions per hardware task, the higher the probability of fitting tasks on the reconfigurable array. Thus, the task rejection ratio and the reconfigurable array utilization ratio are improved. Consequently, the philosophy beyond multi-shape is to partially shift the complexity of the runtime scheduling and placement from online time to offline or design time. Therefore, the extra effort at design time consists of generating as many versions (bitstreams) of each hardware task as possible.

There are at least two more reasons for using multi-shape tasks :

- *From a power consumption perspective* : providing various implementation versions of a hardware task is meaningful in power-aware systems. Indeed, the power consumption of a hardware task is quite influenced by its size, its operation frequency and the type of logic resources that implement it. For example, a pure CLBs-based module may consume more power than a module that mainly uses dedicated ASIC blocks (e.g DSP blocks, hard core processor, etc.). A system can switch from a normal that state to a power-aware state just by swapping tasks versions in and out the system depending on their power consumption. For example, the system may provide a high QoS in its normal state at the cost of power consumption, and may switch into a state that sacrifices the QoS for the sake of energy save.

- *From a cryptography and encryption perspective :* multi-shape hardware tasks may also be recommended in digital systems that fear of being spied at the physical implementation level by techniques like differential power analysis. By tracking the energy consumed by a mathematically secured digital system, a differential power analysis may collect enough information to break its encryption. However, the system may be more difficult to track if different variants of each task are dynamically swapped in and out the reconfigurable array, or relocated at runtime.

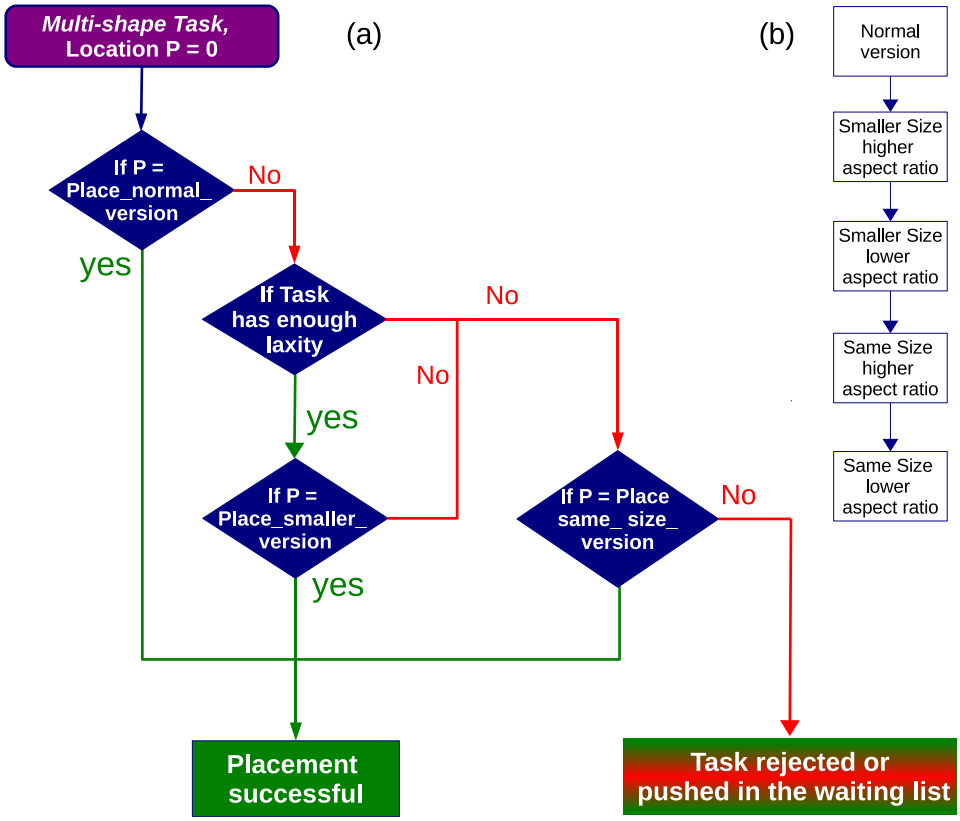


Figure 5.6: Flow chart of the multi-shape algorithm that selects task version to be placed.

5.5.2 The multi-shape basic algorithm

The multi-shape algorithm schedules a list of ready tasks that is sorted according to their arrival times. Hence, the algorithm differs from the *basic scheduling* algorithm presented earlier only by the fact that each hardware task provides more than one rectangular shape. Tasks are kept into the queue as long as they can still meet their deadline. At each time t_{sch}, the scheduler is invoked only in the event of task(s) termination (t_f) and/or task(s) arrival (t_r) and proceeds as followed:

- if ($t_{sch} = t_f$), the algorithm scans the queue from the head, attempting to place tasks as far as possible, removing (rejecting) tasks which cannot still meet their deadline.
- if ($t_{sch} = t_r$), the algorithm attempts to place the arriving task(s). If the attempt fails, the task(s) is inserted in the back of the ready tasks queue.

The algorithm chooses a version of the elected task among its existing versions in the order of priority mapped in figure 5.6(b). The latter figure shows that among two versions of different size, the algorithm prioritizes smaller size versions. Prioritizing smaller versions over other same size versions tends to reduce as much as possible the total amount of configurable resources used by each task. However, in the case of identical size versions, the algorithm prioritizes the version with the highest aspect ratio. The flow chart of figure 5.6(a) details the algorithm. While finding a location P for a multi-shape task, the algorithm first checks whether the normal version may fit on the reconfigurable array. In case of failure, it checks smaller versions if the corresponding task processing time will not lead to a completion time that violates the deadline constraint. Other versions that have the same size with the normal version are only checked if any smaller version hasn't been successfully placed. If any version of the task cannot fit at the scheduling time t_{sch}, the task is added or kept in the waiting list for a further placement attempt, if is can still meet its deadline.

Distributed Arithmetic as an enabling technique

Distributed Arithmetic (DA) is a computation algorithm that uses memory instead of multipliers to perform sum of products where one of the operand remains constant. The algorithm is denoted as multipliers-less (see section 7.9.1, Appendix 7.9. The equation 5.7 expresses the output of a FIR Filter. It is a good example of sum of products, as processing one output sample $Y(n)$ requires the accumulation of N product terms.

$$Y(n) = \sum_{l=0}^{N-1} H_l \cdot X_l(n) = \sum_{l=0}^{N-1} H_l \cdot X_l \tag{5.7}$$

H_0, H_1,..., H_{N-1} are N constant and time-invariant filter coefficients that are computed beforehand. N is the filter length. At each time n, the output response $Y(n)$ is function of the N lasts inputs samples X_0, X_1...X_{N-1} only. Therefore, n may be implicit as shown in the final equation. The output requires $2N-1$ arithmetic operations (N multiplications and $N-1$ additions). Different techniques that range from pure serial implementation to fully parallel may be used for FPGA implementation of the filter. Such a convolution is very common is DSP functions, and is suitable for DA implementation.

The *Appendix E* page 261 gives a detailed example of the filter, designed as a multi-shape hardware task using the DA algorithm. Different trade-offs between the configurable resources and the filter throughput are obtained. In the past, multipliers-less techniques were very useful as multipliers were very configurable resource-consuming. Fortunately, nowadays, FPGAs are embedding

numerous hardwired high performance DSP blocks.

The simulations results for *multi-shape tasks scheduling* are presented and discussed in Chapter 6, section 6.4.

5.6 Conclusion of the Chapter

This chapter has discussed different scheduling algorithms and some placement aspects of these algorithms. The chapter mainly focused on scheduling through two approaches : the *looking-ahead* scheduling and the *without-looking-ahead* scheduling approach. On one hand, a family of *without-looking-ahead* algorithms denoted as *tasks parameters based* has been studied . These algorithms was based on geometric and/or temporal parameters of the tasks. In addition, the *multi-shape scheduling* algorithm was proposed. The algorithm assumes that each task may be provided with more than one shape or size at its design time. On the other hand, *looking-ahead* scheduling algorithms were combined with low complexity placement strategies (e.g. 1D partitioned) in order to provide scheduling solutions that were likely to have low runtime overheads. Furthermore, a ternary tree that eases the area management for looking-ahead horizon scheduling was proposed and investigated.
This chapter relied on the methodology presented in Chapter 4. It also relied on some intuitive assumptions on improvements and performance that can be achieved by the above proposed scheduling/placement strategies. The next chapter is devoted to simulations and experiments results that will assess, validate or invalidate the studies above.

Chapter 6

Simulation Results of the Algorithms Proposed to Solve Online Real-Time Scheduling Issues

6.1 Introduction

This chapter presents and explains the simulation results of the experiments conducted in order to assess and compare the online scheduling algorithms presented in the previous chapter. Figure 6.1 depicts the global simulation environment. The input instances will be built first. These input (e.g. parameters of the tasks in the application) are generated by probability distributions. Therefore, they will be submitted to different scheduling algorithms and their placement strategies. The output data will then be collected in the form of performance metrics that have been presented earlier in Chapter 4. Afterwards, a quantitative analysis will be performed on these output data (figure 6.1), according to metrics. This analysis will be highlighted through meaningful diagrams. The simulation results will be analyzed, classified and discussed with respect to the input instances.

In this thesis, the analysis mainly relies on an *average-case analysis* as described in chapter 3, section 3.4.3. These analysis consider the average performance of the scheduling algorithms or heuristics over all or a range of the input instances. However, *worst-case analysis* (e.g. competitive analysis) remains the widely used methodology for guaranteeing the performance of online

scheduling algorithms on uniprocessor or multiprocessor systems (see Chapter 3, section 3.4.3). Thus, in this thesis, the *competitive analysis* was introduced and transposed in some cases of online scheduling on reconfigurable hardware, but not more than that. Fortunately, since a *competitive analysis* can only trap the worst case behavior of online algorithms, numerous experimental studies (e.g. Albers and Schröder, 2002) have shown that, on real world jobs, these algorithms quite often outperform the well known $c = 2 - \frac{1}{m}$ Graham et al. (1979)'s competitive ratio for $m - processors$ scheduling. Therefore, the following average-case analysis are valid.

Figure 6.1: Summarizing the scheduling problem as defined in this thesis.

6.2 Building the Inputs and the Testing Environment

For experiments purpose, it is assumed that the inputs consist of sets of tasks and the reconfigurable array. A tasks parameters generator has been implemented. It allows a user to generate the tasks parameters following two probability distributions : the *uniform* distribution and the *Gaussian* distribution. However, in order use more realistic tasks parameters, a great range of values was covered on one hand, and on the other hand, size and timing characteristics of real life hardware IPs were used.

In the following sections, some common IPs for DSP applications are characterized, their size estimated and the tasks sets generated accordingly.

6.2.1 Hardware Tasks Characterization

With the aim to remain close the the reality, especially in terms of size, a census of available IPs in XILINX COREGEN[1] IPs library was taken, in addition to some others fairly well documented

[1] Xilinx CORE Generator System^{TM} accelerates design time by providing access to highly parametrized Intellectual Properties (IP) for Xilinx FPGAs and is included in the ISE® Design Suite. CORE Generator provides a catalog of architecture specific, domain-specific (embedded, connectivity and DSP), and market specific IP (Automotive, Consumer, Military/Aerospace, Communications, Broadcast etc.). These user-customizable IP functions range in complexity from commonly used functions, such as memories and

from the Internet. In order to find realistic range for tasks parameters, it would have been necessary to develop a synthesis protocol which synthesize many versions of each IP for FPGA implementation. Hence, versions of the same IP would differ from each other by prioritizing one option over another, or by combining optional characteristics of the IP, such as the accuracy, the datapaths width, the security and encryption, the throughput, the use of synchronization signals or not, etc.

These combinations and options would have produced a tremendous amount of possible synthesis per IP. Therefore, a uniform distribution instead of a Gaussian one was used, as it best reflects the case. Each IP was synthesized twice, once in its lightest configuration, and once in its fullest configuration. Hence, it was assumed that, statistically, the values of the tasks size were uniformly distributed between these two limits.

Table 6.1 gives a framework of minimum and maximum sizes of IPs in terms of number of slices required. The IPs are grouped according to their application domain. A more detailed table of existing IPs and their size can be found in *Appendix D*, table 7.7, page 260.

IPs	*Size on Virtex2pro FPGA*	*Estimated size on Virtex5 FPGA*
Communication IPs	$\in [1000; 3000]$ slices	$\approx\in [350; 1500]$ slices
Floating Point operations IPs	$\in [100; 500]$ slices	$\approx\in [45; 300]$ slices
CORDIC algorithm	$\in [100; 600]$ slices	$\approx\in [50; 270]$ slices
FFT	$\approx$ 2000 slices	$\approx$ 950 slices
Video processing IPs	$\in [3000; 9000]$ slices	

Table 6.1: Approximate sizes of most common IPs (hardware tasks).

6.2.2 Estimating the Size of Tasks

Table 6.1 reports on the number of slices used, but does not provide any information on the number of slices per column (height) and per line (width). Therefore it is necessary to estimate the width and the height of IPs or tasks, according to the rectangular-shaped task model adopted

FIFOs, to system-level building blocks, such as filters and transforms. Using these IP blocks can save days to months of design time. The highly optimized IP allows FPGA designers to focus efforts on building designs quicker while helping bring products to market faster.

in this thesis.

Let S_{max} (resp. S_{min}) be the maximum (resp. minimum) number of slices required for a given IP application or set of n tasks Γ_n. One can estimate height and width parameters by a mean m and a standard deviation σ, so that in extreme cases (that correspond to maximum values of the uniform law), one reaches the limits of the number of slices. This is expressed as follows :

$$(m+\sigma)^2 = \quad S_{max} \quad \text{and} \quad (m-\sigma)^2 = \quad S_{min} \tag{6.1}$$

The solution of these equations gives an estimation of parameters m and σ :

$$m = \quad \frac{\sqrt{S_{max}}+\sqrt{S_{min}}}{2} \quad \text{and} \quad \sigma = \quad \frac{\sqrt{S_{max}}-\sqrt{S_{min}}}{2} \tag{6.2}$$

For the random generation of the size of hardware tasks, it is possible to do it either according to a specific application domain or in a more generic way. In the first case, the targeted application domain may allows the designers to evaluate S_{min} and S_{max} and therefore to estimate the parameters m and σ. m and σ may even feed free tasksgraph generators like TGFF[2]. In the second case, the parameters are simply randomly generated following the uniform distribution.

6.2.3 Final Inputs Values for Experiments

The size of the Xilinx's FPGA XCV1000 is used as reference size of the reconfigurable array. The corresponding width and the height of the array are respectively $W = 96$ and $H = 64$. Thus, there are 6140 CLBs available on the reconfigurable device. This size has been formerly used in few research papers.

However, nowadays, FPGAs are getting denser and may integrate 10 times more CLBs than the Xilinx's FPGA XCV1000. The sizes S of the generated tasks were also chosen accordingly in a way that the device may accommodate about 4 tasks of the maximum size S_{max} at a time. In most of the conducted experiments, the parameters of 100 sets of 50 aperiodic real-time tasks were randomly generated following the uniformly distribution in the intervals listed below :

- Size $S \in [50; 1500]$ CLBs.
- Aspect ratio $ar = \frac{h}{w} = \{\frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 1, 2, 3, 4, 5\}$.
- Processing time $e \in [5; 100]$.

[2] TGFF stands for Tasks Graph For Free. TGFF is an open source software created in 1998 by R.P. Dick and D.L. Rhodes in order to facilitate tasks graph generation for scheduling problems analysis.

- Laxity class A $\Rightarrow l \in [1;10]$, class B $\Rightarrow l \in [11;50]$ and class C $\Rightarrow l \in [51;100]$.
- Relative application computational load given by equation 4.17 page 161 and that is usually denoted in this chapter as *application load* App_{load}; $App_{load} = 50\%$.

The above parameters are used by default for simulations, unless new values are explicitly indicated.

6.2.4 The Running Environment

Apart from simulations that have been conducted on the microblaze embedded processors for accurate timing measurements, scheduling algorithms were simulated on a laptop computer. The latter hosts the Intel premium dual-core processor T2330, running at 1.6 GHz and featuring an 1MB L2 cache memory.

6.3 Tasks Parameters Based Scheduling

The simulation results of tasks parameters based scheduling are presented in this section.

Figures 6.2 and 6.3 depict the simulation results of the Basic, EDF, SSF and BSF scheduling algorithms that use single-version tasks. The placer manages non overlapping vertically split areas through a binary tree as described in Bazargan et al. (2000). Consequently, any improvement of one algorithm over another will be thanks to the scheduling scheme. Experiment results are expressed in terms of laxity classes.

6.3.1 Chip Utilization Ratio and Tasks Rejection Ratio

From the reconfigurable array utilization ratio and tasks rejection ratio perspective, simulation results shown by figure 6.2 are as follows :

- *For tasks of laxity class A*, the chip utilization is $\sim 30\%$ no matter which scheduling algorithm is used. The main reason is that once a task is released, its maximum waiting time is equal its laxity. Consequently, for a laxity class $l \in [1;10]$, the scheduling algorithms does not have enough time to schedule the tasks. As the underlying placement strategy is similar, the resulting chip utilization ratios are nearly similar.
  The tasks rejection ratio is also $\sim 22\%$ independently to the scheduling algorithm. Finally, BSF algorithm slightly outperforms the other algorithms in terms of utilization ratio while SSF behaves worst than others.

- *For tasks of laxity class B*, the chip utilization rises to $\sim 33\%$ while the tasks rejection ratio drops to $\sim 12\%$. Indeed, thanks to a higher laxity, there is more time for finding a fitting solution to tasks. Once again, BSF produces the best utilization ratio, and SSF the worst tasks rejection ratio.
- *For tasks of laxity class C*, the chip utilization ratio does not significantly increase. BSF outperforms with $\sim 34\%$. However, the tasks rejection ratio decreases to around $\sim 5\%$.

As shown in figure 6.2 and commented above, any of the tasks parameters scheduling does not significantly outplay another. However, significant improvements are gained in terms of laxity instead, especially on tasks rejection ratio. One may conclude that these results are highly tied to the underlying placement strategy used here, which is the above mentioned Bazargan et al. (2000)'s binary tree based placer.

The diagram at the bottom of figure 6.2 depicts the *differential quality metric* UR_{qm} as the difference between the utilization ratio and the tasks rejection ratio. The metrics gives an overall performance that takes into account both the utilization ratio and the tasks rejection ratio as expressed earlier in equation 4.24, page 163. A high chip utilization ratio increases UR_{qm} while a high tasks rejection ratio decreases it. Hence, the higher the differential quality metric UR_{qm}, the better the scheduling. From the UR_{qm} perspective, BSF remains the best parameters based scheduling while the BSF is the worst. Intuitively as bigger tasks are placed first, the utilization ratio is therefore higher. UR_{qm} may allow the designer to choose among two algorithms that share similarities either in terms of tasks rejection ratio, or in terms of chips utilization ratio.

6.3.2 Runtime Overhead

Figure 6.3 depicts the runtime overhead of different of tasks parameters based scheduling algorithms. The top left of the figure shows the average time the scheduling function has taken to run every it has been invoked. These time overheads are globally around 40 to $47us$. It can be noticed that, for example in laxity class A, the average time overhead slightly increases from one algorithm to another. This increase comes from the increasing difficulty in keeping the list of ready tasks sorted according to the considered criteria. Keeping the list of tasks according to their release time is easier. Identically, keeping the tasks in the list according to their deadline or their laxity is somewhat related to their release time. Conversely, sorting the released tasks according to their size is completely time independent. Therefore BSF and SSF lead to a more time consuming process.

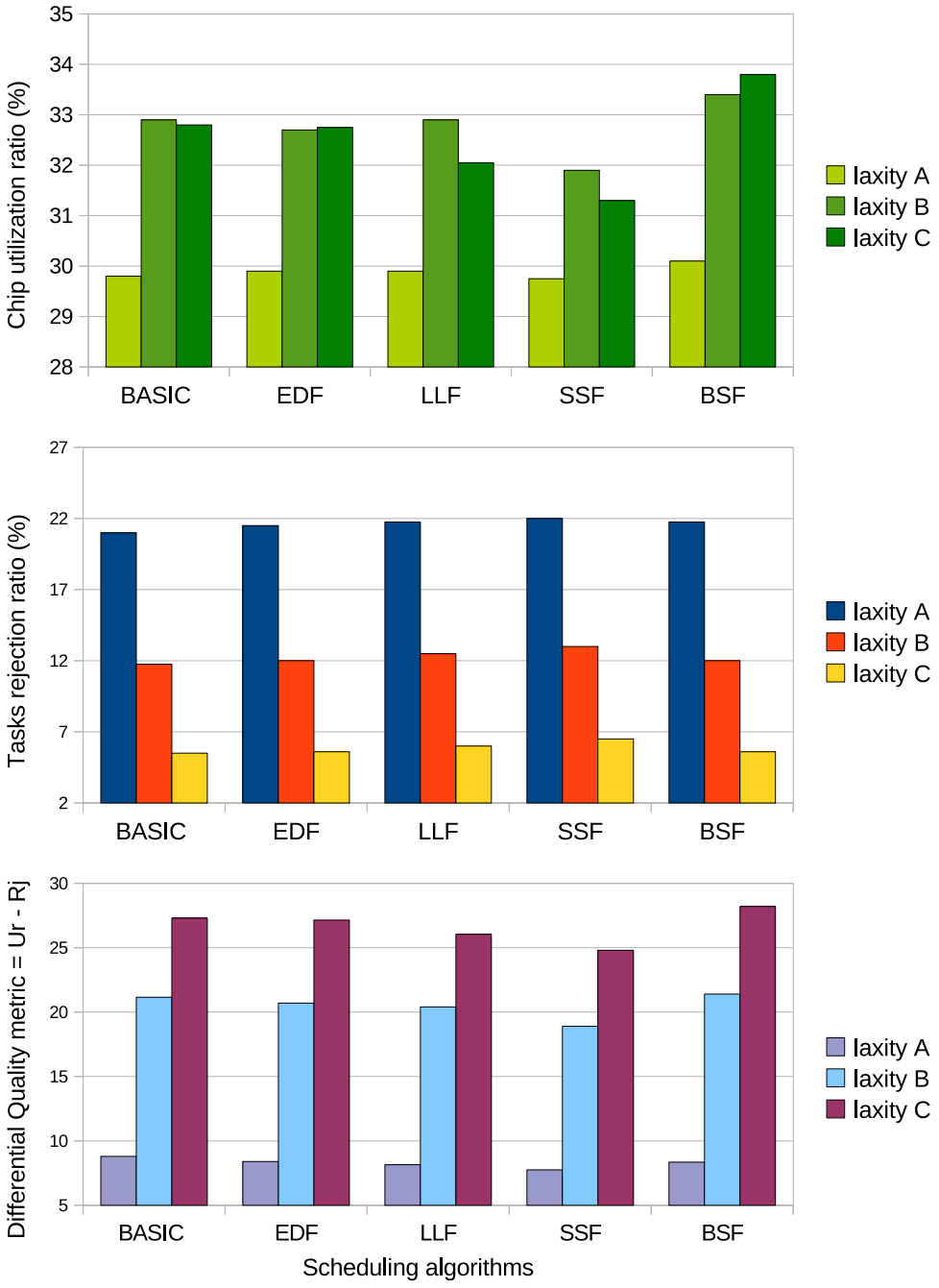


Figure 6.2: Utilization ratio (top), rejection ratio (middle) and quality metrics (bottom) : comparative results for EDF, LLF, SSF and BSF scheduling algorithms.

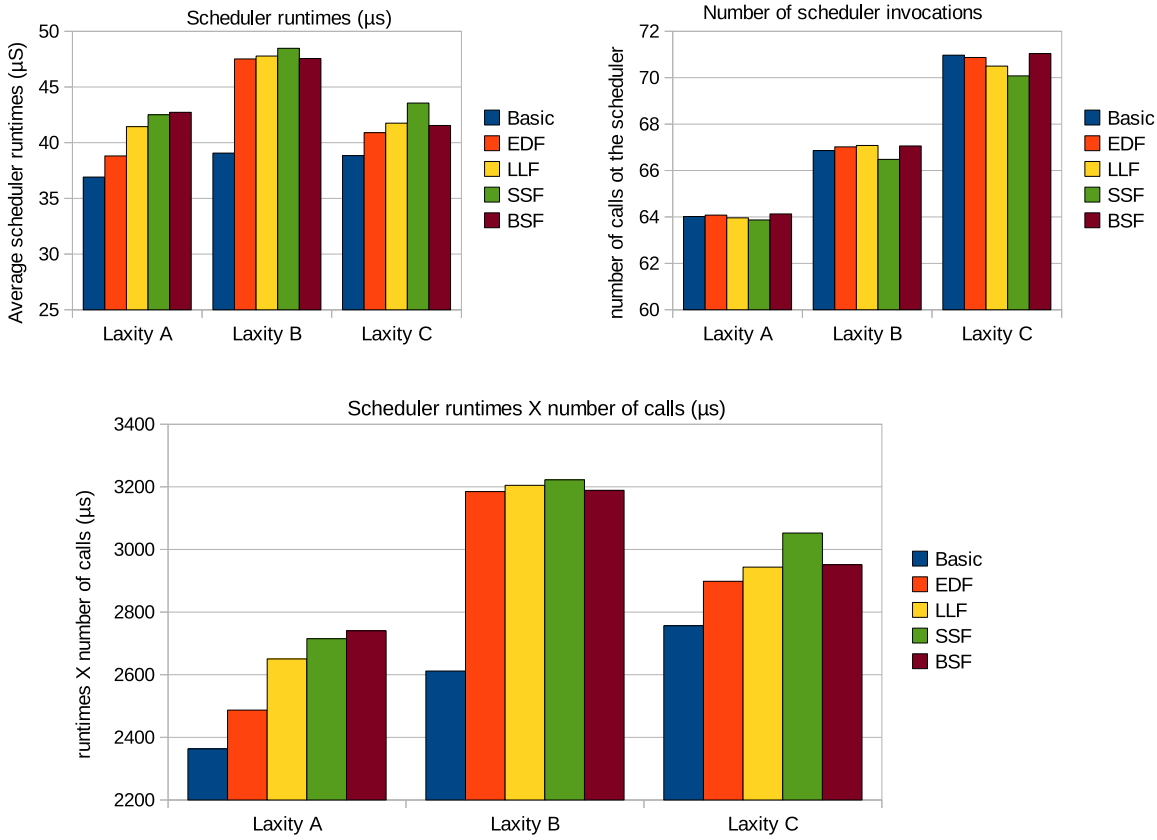


Figure 6.3: Scheduling runtime overhead, number of scheduling calls and cumulative scheduler runtime overheads : comparative results on EDF, LLF, SSF and BSF scheduling algorithms.

The average number of invocations may also affect the global time spent in running the scheduling algorithm. The top-right of the figure illustrates the average number of calls to the scheduler. Recall that the scheduler is invoked whenever a task completes or arrives. Figure 6.3-top-right shows that the higher the laxity, the higher the number of calls. The reason is that a longer laxity gives more opportunity to the scheduler to place each task, and therefore increases the number of calls. The average number of calls is around ~ 64 to 71.

Surprisingly, as shown in the top-left of figure 6.3, the average runtime overheads of scheduling algorithms are higher for tasks of laxity B. Even when considering the cumulative runtime overhead that takes the number of invocations into account, the trend is confirmed. The diagram at the bottom of figure 6.3 depicts the case. The cumulative runtime overhead corresponds to the average

time spent in the scheduling function every time it is invoked, times the number of calls to the scheduler. Hence, the third diagram results in the multiplication of the two diagrams above. Put it differently, the cumulative sum of the time spent in the scheduling algorithm throughout the simulation is calculated, in order to obtain a bigger and therefore more expressive value.

As all the tasks are not accepted in the present case, the makespan is meaningless. Indeed, an algorithm which rejects too many tasks is likely to have a shorted makespan. Therefore, the makespan of two scheduling algorithms are comparable only if both have accepted exactly the same tasks.

6.3.3 Conclusion on parameters based scheduling

Tasks parameters scheduling algorithms are simple to implement, do not have a high runtime overhead, and are based on one parameter of the task. Experiment results show that these algorithms are not far from each other in terms of scheduling and placement quality (e.g. runtime overhead, reconfigurable array utilization ratio and task rejection ratio). As they use the same placement strategy, the similarities in chip utilization ratio and tasks rejection ratio suggest that a significant improvement of scheduling may not come from a single parameter of the task. In all likelihood, it may be interesting either to combine temporal and geometric characteristics in order to build novel algorithms or to propose areas management approaches that does not increase the overall algorithm complexity and runtime overhead, but improves array utilization and tasks rejection ratio. Hereinafter are the experiments results for the multi-shape tasks based scheduling algorithms.

6.4 Multi-shape Tasks Based Scheduling

This section presents the simulation results of the multi-shape based scheduling. The algorithm has been previously studied in Chapter 5, section 5.5 and described in figure 5.6, page 194.

6.4.1 Multi-shape Tasks

Relying on equation 5.6 page 193, one or several versions of each task were produced. It is assumed that each task has at least one version, which is its *normal version*.
The *normal* version of a task is the version that results from a random generation of the tasks parameters with a given probability distribution, as discussed earlier. A version is denoted as

standing (Std) or *laying (Lay)* depending on its aspect ratio compared to the normal version, as depicted in figure 6.4. A version is denoted as *smaller (Sml)* or *same (Sm)* depending on its size compared to the normal size version, as also depicted in figure 6.4. Hence, while a *same* size version uses the same amount of configurable resources as the normal size, the *smaller* size version uses only half this amount.
A standing version will tend to increase its aspect ratio by doubling its height in comparison with the normal version's height. However, this height cannot exceed the height of the reconfigurable array. Therefore, the width of the standing version is adjusted accordingly in order to obtain a size either identical (*same*) to the size of the normal version, or *smaller*. Conversely, a laying version will tend to be twice wider than the normal version, as far as the width does not exceed the the width of the reconfigurable array. The height of the laying version is adjusted accordingly in order to generate either a same size task, or a smaller size task. As pictured in figure 6.4, several variants of multi-shape hardware tasks were generated. Various combinations of these variants can provide variants of the multi-shape algorithm. Each combination is also characterized by the number of versions per hardware task. Below are listed examples of possible combinations that were used in this thesis :

1. *Sm_Std(2)* ; where (2) represents the number of versions or variants per task. In this scenario illustrated in figure 6.4, each hardware task has a *normal* version and an additional version denoted as *Sm_Std*, which stands for *same size standing* version. The same standing version is another version that uses the same amount of resources than the normal version, but with a higher aspect ratio as described above. Figure 6.4 illustrates the *Sm_Std(2)*.

2. *Sm_Lay (2)* ; this scenario is similar to the former. However, the second version has the same size with the normal version, but with half the aspect ratio if possible. Here, *Sm_Lay (2)* stands for *same size laying* version.

3. *Sml_Std(2)* ; a *normal* plus a *smaller size standing* version (here *Sml* stands for smaller size). The latter version shares the same height with the normal version, but with half its width. Here, *Sml_Xxx (y)* stands for *smaller size*

4. *Sml_Std_Lay(3)* ; a *normal* plus a *smaller standing* and a *smaller laying* versions.

5. *Sm_Std_Lay(3)* ; in this scenario depicted in figure 6.4, each task has 3 variants : the *same standing* and the *same laying* in addition to the *normal* variant.

6. *Sm_Sml_Std_Lay(5)* ; this case is shown in figure 6.4 where each task has 5 variants : the

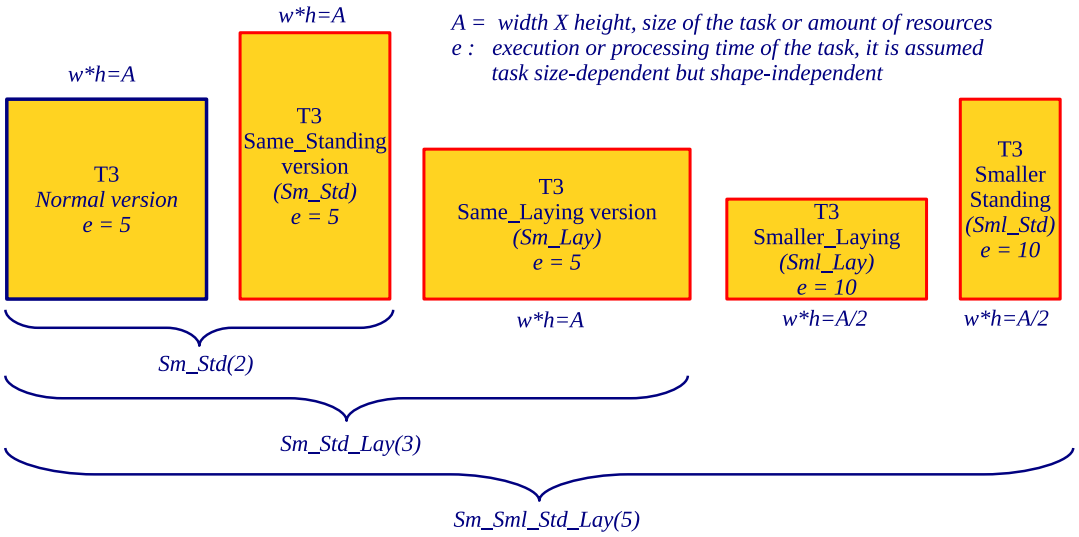


Figure 6.4: Different combinations of multi-shape tasks for variants of multi-shape scheduling algorithm

same standing, the *same laying*, the *smaller standing* and the *smaller laying* in addition to the *normal* variant.

7. *Shuffle(1...5)* ; this case reflects a more realistic scenario. Indeed in real life applications, it is quite difficult to have more than one task variant for each task. Hence, in the present case, it is assumed that the number of versions per task is $\in [1;5]$. The number and the type of versions are randomly generated for each task, following the uniform distribution.

Variants of the *multi-shape scheduling* algorithm above are compared to the *Basic scheduling* algorithm denoted as *Basic(1)* and described in Chapter 5. The latter algorithm schedules only the normal version of tasks. Both algorithms use the same aforementioned binary tree based placement strategy. Furthermore, both algorithms rely on the basic scheduling algorithm where the tasks in the ready list is sorted according to their release time. Therefore, comparing multi-shape scheduling algorithms with basic scheduling brings out the improvements gained by the use of multi-shape tasks. Simulations have been conducted on 100 sets of 50 tasks; Hence, the final simulation result is an average of 100 simulation results.

6.4.2 Chip Utilization Ratio and Tasks Rejection Ratio

Figure 6.5 shows the simulation results of the chip utilization ratio. The figure compares variants of *multi-shape scheduling* algorithms with the *basic scheduling* algorithm that uses only single-version tasks. Utilization ratios are also classified according different classes of laxity. Hence, it can be observed from the graph that :

For tasks of laxity Class A

The reconfigurable array utilization ratio is around $\sim 30\%$ (see figure 6.5) for the basic scheduling algorithm and four multi-shape scheduling algorithms which are: the Sml_Std(2), the Sml_Lay(2), the Sml_Std_Lay(3) and the Sm_Lay(2). The three first multi-shape algorithms use additional smaller versions tasks, while the fourth (Sm_Lay(2)) uses an additional same size laying version. The two main reasons for these similarities in the results are listed and analyzed as follows :

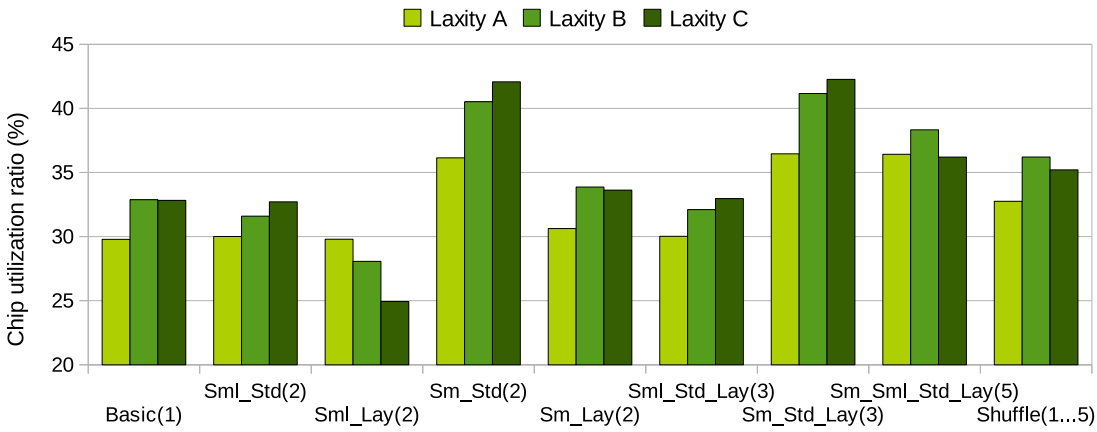


Figure 6.5: Multi-shape scheduling algorithms: simulation results of the utilization ratio, comparison with the Basic scheduling.

(i) In algorithms where each task provides a normal version and additional smaller size versions, the latter versions require a longer execution time on the reconfigurable array to complete. Hence, tasks may rarely meet their deadline unless their laxities are very long. It is not the case here where $laxity \in [1, 10]$ for tasks of laxity class A. *Sml_Std(i), Sml_Lay(i) and Sml_Std_Lay(3)* are such multi-shape algorithms, i being the total number of tasks versions.

(ii) In algorithms like *Sm_Lay(i)*, *Sml_Lay(i)*, etc. where each task provides a normal version

and additional laying versions, the latter versions are likely to be rejected if the areas splitting is done vertically. Indeed, vertically (resp. horizontally) split free areas are likely to produce more standing than laying areas (resp. more laying than standing areas). Therefore, standing (resp. laying) areas are more suitable for accommodating standing tasks which aspect ratio $ar > 1$ (resp. laying tasks which $ar < 1$). This results clearly bring out the influence of the underlying placement strategy on the overall quality of the scheduling algorithm.
This can also be observed in the results of the *Sm_Std(2)* and *Sm_Std_Lay(3)* that are similar, making the use of a third task version in *Sm_Std_Lay(3)* useless, as it does not bring any improvement.

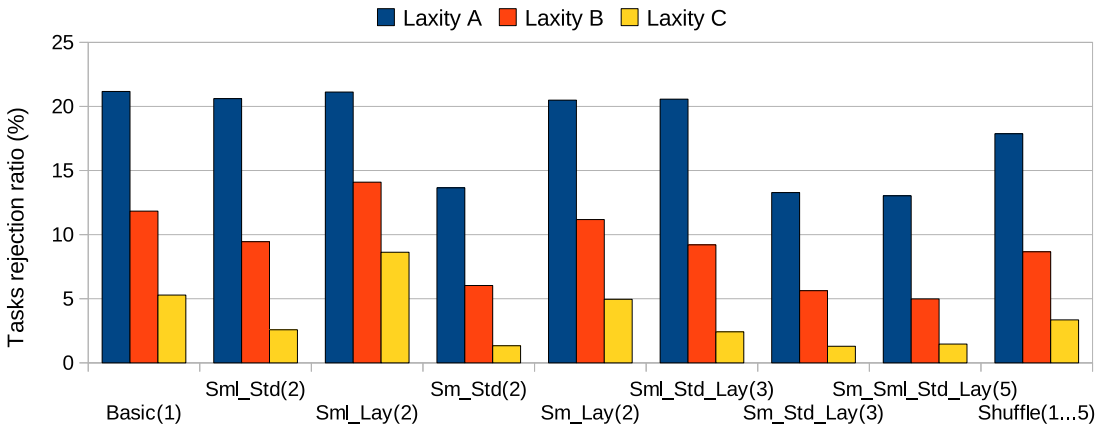


Figure 6.6: Multi-shape scheduling algorithms: simulation results of the tasks rejection ratio, comparison with the Basic scheduling.

Figure 6.6 depicts simulation results of the tasks rejection ratios. The tasks rejection ratios are almost similar ($\sim 21.5\%$) for the basic scheduling and the aforementioned four multi-shape scheduling algorithms. The two reasons for these similarities in results are the same listed and described above (tasks versions with longer execution time, or an unsuitable free areas splitting strategy).

Conversely, multi-shape scheduling algorithms that use same size standing versions as additional tasks versions significantly improve the chip utilization and the tasks rejection ratio for tasks of laxity class A. *Sm_Std(2)*, *Sm_Std_Lay(3)* and *Sm_Sml_Std_Lay(5)* are these algorithms. Their simulation results do not differ from each other, as shown in figures 6.5 and 6.6. It can be observed from the graph of figure 6.5 that these 3 multi-shape algorithms raise the chip utiliza-

tion ratio from $\sim 30\%$ to $\sim 37\%$ when compared to basic scheduling or any other multi-shape algorithm that does not use a same size standing version as additional version of tasks. Figure 6.6 also shows a significant reduction in tasks rejection ratios. The latter decrease from $\sim 21.5\%$ to $\sim 13.5\%$.

Regardless the number of tasks versions used, *Sm_Std(2)*, *Sm_Std_Lay(3)* and *Sm_Sml_Std_Lay(5)* scheduling algorithms provide slightly similar results in terms of chip utilization ratio and tasks rejection ratio. This similarity indicates that the improvements are exclusively brought by the use of the same size standing version, in addition of the normal version. Therefore, smaller size versions and same size laying versions that are provided to *Sm_Std_Lay(3)* and *Sm_Sml_Std_Lay(5)* algorithms are useless here as they bear the two drawbacks listed and described above (a very long processing time and/or an areas splitting strategy that is unsuitable).

For tasks of laxity Class B

Improvements are better, and follow the same trend as the results for the laxity class A above. Hence, using a laying version (same or small) does not improve the results because of the above-mentioned splitting strategy. Once again, best results are obtained while using a same size standing version as additional version. This can be observed in figure 6.5 and figure 6.6 where *Sm_Std(2)* algorithm increases the chip utilization ratio from $\sim 33\%$ to 40.6% and decreases the tasks rejection ratio from $\sim 12\%$ to 6% while compared with the basic scheduling algorithm. This corresponds to an improvement of 50% and 23% for tasks rejection ratios and chip utilization ratio respectively. In other words, the *Sm_Std(2)* algorithm that uses only two versions per task (normal plus same standing), rejects 50% less tasks and looses 23% less reconfigurable resources, compared with basic scheduling.

As the laxity class B is bigger here ($laxity \in [11, 50]$), tasks deadlines are longer accordingly. Therefore, there is more opportunity for placing the smaller version of tasks, even if they require longer processing times. Tasks with smaller but standing version slightly improve their rejection ratio, thanks to their laxity and to their aspect ratio. For example, the *Sml_Std(2)* algorithm records a decrease ($\sim 20\%$) in tasks rejection ratio compared with basic scheduling, for a slightly similar chip utilization ratio ($\sim 32\%$).

The impact of the areas management strategy on multi-shape scheduling algorithms is even more obvious here. For example, the basic scheduling algorithm that uses single-version tasks outperforms the *Sml_Lay(2)* algorithm in terms of chip utilization ratio and tasks rejection ratio. The reason is that, as the laxity class B is longer, the *Sml_Lay(2)* algorithm may fit the smaller

version of some tasks on the reconfigurable array, instead of the normal version. But these smaller versions require longer processing times. Therefore, on one hand, placing small size tasks lowers the chip utilization ratio [3] and leads to a more fragmented chip. On the other hand, as smaller version of tasks require more time to process on the reconfigurable array, they prevent the scheduler from placing newly arriving or waiting tasks. Furthermore, as the free areas are vertically split in the present case, they are mostly standing areas, and are less suitable for accommodating laying tasks.

For tasks of laxity Class C

Apart from some special observations, the comments made above on the simulation results for tasks sets of laxity class B are also valid for tasks sets of laxity class C. As depicted by figure 6.5 and figure 6.6, there are major improvements in chip utilization ratio for multi-shape scheduling algorithms that are provided with same size standing and/or smaller size standing versions of hardware tasks.

For example, compared to the basic scheduling algorithm, the *Sm_Std(2)* algorithm decreases the tasks rejection ratio by 75% and increases the utilization ratio by 28% compared with the basic scheduling algorithm. Hence, as *Sm_Std(2)* algorithm achieves a 1.33% tasks rejection ratio, it accepts almost all the tasks. These improvements are gained only by using two versions per task. One can observe that regardless of the number of versions per tasks, any other multi-shape algorithm does not outperform *Sm_Std(2)* algorithm. As previously stated, improving the scheduling through a multi-shaped tasks approach is not a matter of number of versions or shape per task, but essentially a matter of size and aspect ratio of the additional version(s). Figure 6.7 illustrates, on a single page, an overview of the simulation results of the chip utilization ratio and the tasks rejection ratio, along with the resulting *differential quality metric* UR_{qm}. The two first graphs of figure 6.7 replicate respectively the utilization ratio and the tasks rejection ratio graphs of figures 6.5 and 6.6 respectively, while the far bottom graph represents the *differential quality metric* UR_{qm}. The latter metric is simple but allows us to have a global and concurrent interpretation of both chip utilization ratio and tasks rejection ratio. The UR_{qm} confirms that *Sm_Std(2)* is the best multi-shape algorithm thanks to the size and the shape of the additional version. As shown by the graph, it is not really worthy using more than 2 versions per task, if the version takes the underlying placement strategy into account.

[3] as shown by the simulation results in section 6.3, SSF (smallest size first) based scheduling algorithm is outperformed by other tasks parameters based scheduling algorithms.

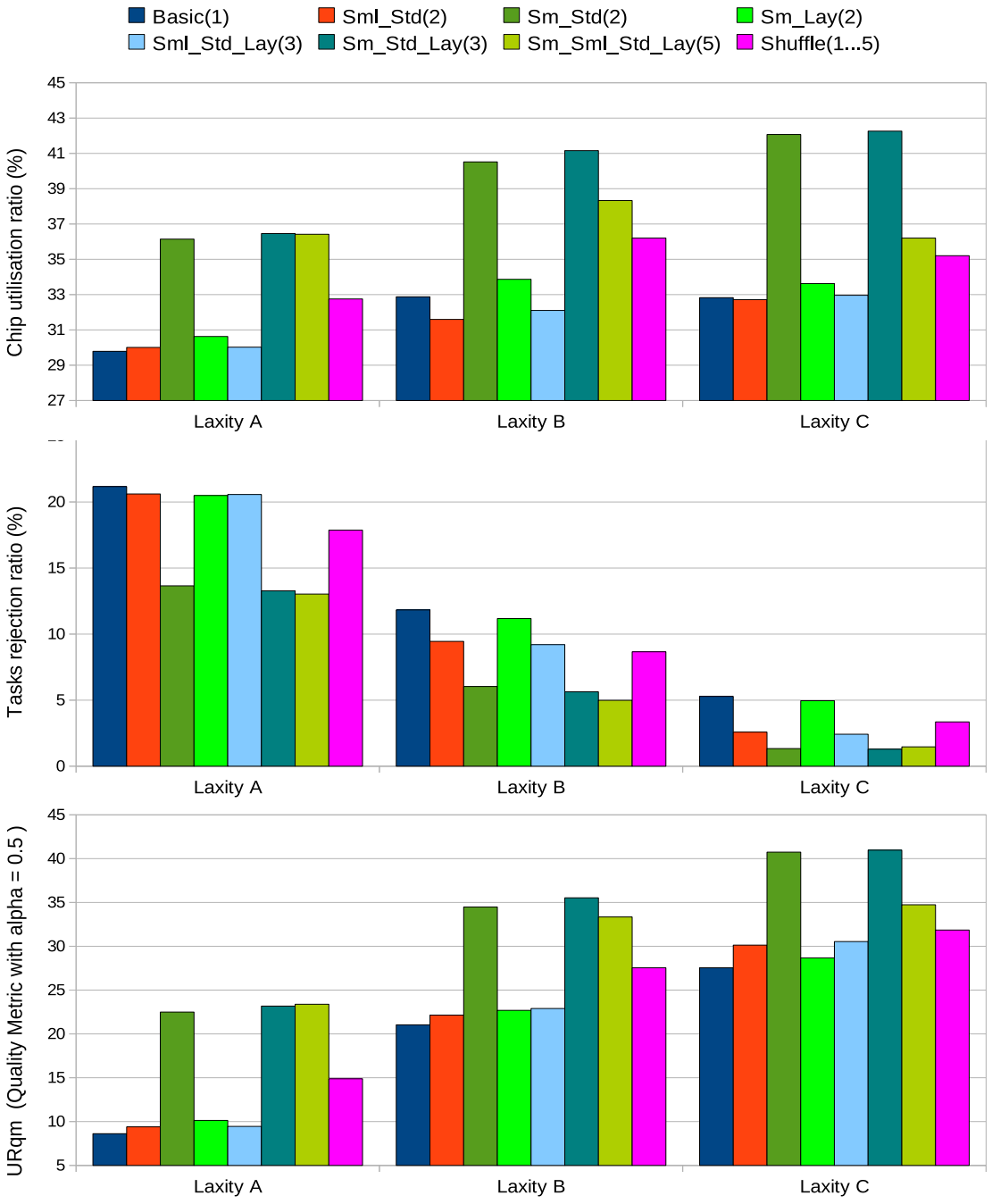


Figure 6.7: Utilization ratio, tasks rejection ratio and differential quality metric UR_{qm} (with $\alpha = 0.5$) : comparative results for basic scheduling and multi-shape scheduling algorithms.

The *Shuffle(1...5)* scheduling algorithm was described as bearing a more realistic case where hardware tasks can have randomly 0 up to 4 additional versions per task. Zooming in on the simulation results of the *Shuffle(1...5)* algorithm may therefore allow us to have a more realistic prediction on improvements achievable by a multi-shape algorithm. The *Shuffle(1...5)* algorithm reflects a scenario where prior to design, all available versions of hardware tasks or IPs that are required for the application are collected. Therefore, for each task, there may be only one or several versions. According to the results graphs depicted in figure 6.7, *Shuffle(1...5)* decreases the tasks rejection ratio by $\sim$ 16% for tasks of laxity class A, $\sim$ 27% for the laxity class B, and $\sim$ 37% for the laxity class C compared to the basic scheduling. At the same time, the algorithm improves the chip utilization ratio by $\sim$ 10% for the laxity classes A and B, and $\sim$ 7% for the laxity class C. It is noticeable that increasing chip utilization ratio is more difficult than improving the tasks rejection ratio. Indeed, a high ($\sim$ 100%) chip utilization ratio is rarely achievable because of the intrinsic fragmentation problem.

6.4.3 Makespan and Runtime Overheads

Comparing algorithms through their makespan is meaningful only if they do not reject any task. In the latter case, an algorithm will outperform another in terms of makespan only if it takes less time to schedule the same set of tasks. Simulation results for tasks of class laxity C depicted in figure 6.7 and commented above shows a nearly similar case. Indeed, according to these results, the tasks rejection ratio decreases to 5.33% for basic scheduling and to 1.33% for *Sm_Std(2)*. Such low tasks rejection ratios make a comparison of scheduling algorithms from the makespan perspective meaningful. Indeed, an algorithm that rejects too many task is likely to have a shorter makespan. The makespan is analyzed only on simulation results for tasks of laxity class C.

Figure 6.8 compares scheduling algorithms through their makespan. Once again, these results show that *multi-shape algorithms* that use same size tasks versions achieve a better makespan compared with *basic scheduling*. For example, *Sm_Std(2)* requires an average of 190 time units to run a tasks set to completion while the *basic scheduling* requires 204 time units. Conversely, the highest makespan is achieved by *Sml_Std(2)* and *Sml_Std_Lay(3)* algorithms that use smaller size tasks versions. The latter versions lengthen the makespan because of their longer execution times. It can also be observed that using additional laying tasks versions do not improve the results, as their aspect ratios do not suit to vertically split free areas. Therefore, *Sm_Std_Lay(3)* provides the same result as *Sm_Std(2)* and *Sml_Std_Lay(3)* the same as *Sml_Std(2)*, regardless

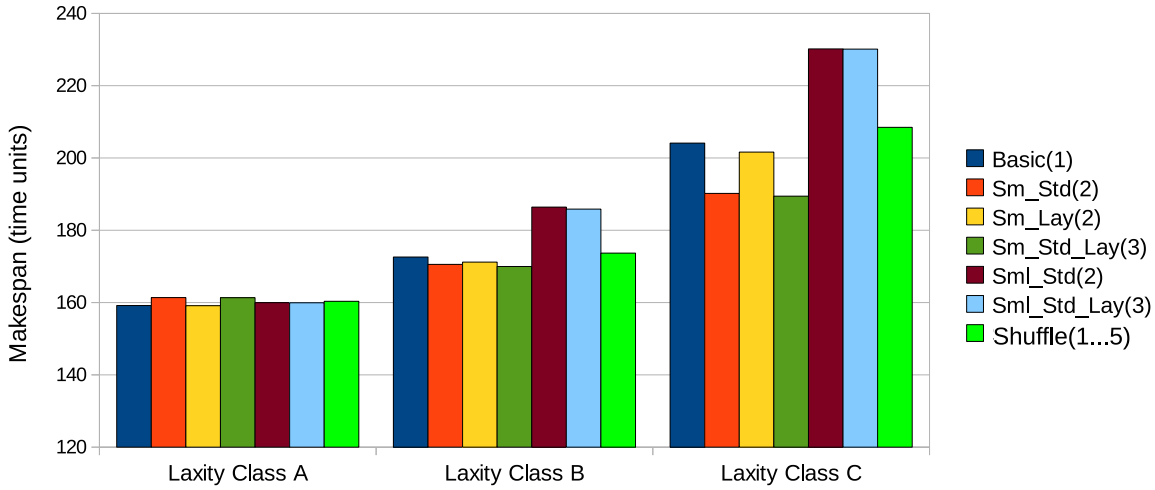


Figure 6.8: The average makespan : comparative results for multi-shape and Basic scheduling algorithms.

of the number of versions per tasks.

Detailed simulation results related to the runtime overheads are presented in the three figures in page 215. The top of figure 6.9 depicts the average runtime overheads of different multi-shape algorithms compared with the basic scheduling algorithm.

For tasks of laxity class A, the runtime overheads range from $37us$ to $47.5us$, the basic scheduling algorithm achieving the lowest one. Intuitively, the runtime overhead is longer for multi-shape scheduling algorithms. Indeed, it takes more time to search for the version that can fit on the reconfigurable array. Therefore, the more the versions per task, the longer the runtime overhead.

However, as shown by the results for the laxity class A, the rule is not that simple. The runtime overhead of each scheduler call also depends on the success or the failure of the current placement attempt. Each successful placement results in operations that can be time consuming. In the present case, the placer uses the hash matrix that have been presented in Walder et al. (2003) and briefly discussed in section 3.8.5, page 118. The matrix requires a long update at each task placement or withdrawal on the reconfigurable array. Hence, the more successful the tasks placements, the more matrix update processes are required, the latter processes being time greedy. Any data structure that holds the state of the reconfigurable array (e.g. binary tree) needs to be updated at each task placement or withdrawal. This impact of the update process on the runtime overhead

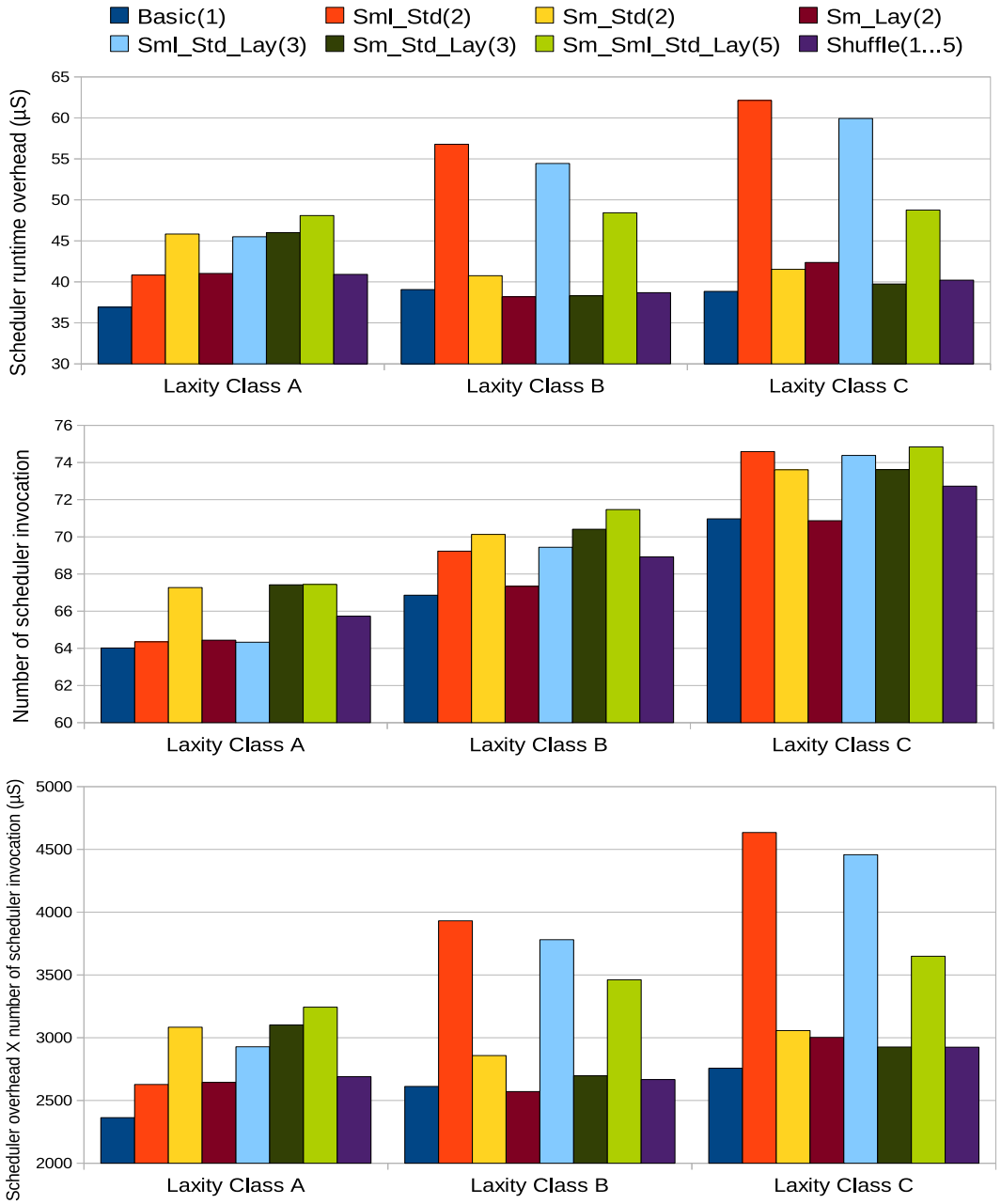


Figure 6.9: Multi-shape scheduling algorithms : the simulation results of the scheduling runtime overhead, with basic scheduling as reference scheduling.

can be seen in the results (top graph of figure 6.9) of multi-shape algorithms that use at least a same size standing version per task. It was noticed earlier that these algorithms (e.g. *Sm_Std(2)*,

Sm_Std_Lay(3) and *Sm_Sml_Std_Lay(5)*) reject less tasks in general, which means in other words that they successfully place more tasks. Therefore for tasks of laxity class A, as the number of updates grows, their runtime overheads are higher (see in the top graph of figure 6.9).
However, even if *Sml_Std_Lay(3)* algorithm rejects more tasks compared with the above mentioned three algorithms, it also achieves a high runtime overhead (see top of figure 6.9). The reason for this is the time spent by the algorithm in trying to place the smaller size version of the task to be placed. This reason is more highlighted by the results for tasks of laxity class B and C. The two highest runtime overheads are due to multi-shape scheduling algorithms which extra tasks versions are exclusively smaller size tasks versions. In the latter case, the algorithms always determine whether the smaller size version of the task can be used without violating its deadline. Let us remind that smaller size versions of tasks require longer execution times. This extra effort in verifying deadline violation significantly increases the runtime overheads of the *Sml_Std(2)* and *Sml_Std_Lay(3)* scheduling algorithms as highlighted on the top graph of figure 6.9, especially for tasks of laxity class B and C.
At this point, based on analysis of results in the top graph of figure 6.9, two main factors that influence the runtime overheads of scheduling algorithms and confirm the preliminary observations made in Chapters 3 and 4 can be identified:

(i). *the search for an accommodating area for a task* ; it can be observed in the above results that the more the versions per tasks, the longer the runtime. This is especially the case when the additional versions are smaller size versions. The latter require an extra effort to determine whether they are not violating the deadline. It results in the highest runtime overheads for *Sml_Std(2)* and *Sml_Std_Lay(3)* for tasks of laxity class B and C.

(ii). *the update of the data structure after a task placement or removal;* if time consuming, it affects the algorithms that achieve best tasks rejection ratios. *Sm_Std(2)*, *Sm_Std_Lay(3)* and *Sm_Sml_Std_Lay(5)* are good examples, especially in laxity class A.

The graph in the middle of figure 6.9 depicts the average number of calls to the scheduling function. Intuitively, as the scheduler is invoked at each task arrival and task ending, the algorithms that successfully placing more tasks induces more scheduler invocations. This intuition is confirm by the simulation results for tasks of laxity class A. Indeed, as shown in the graph, multi-shape algorithms (*Sm_Std(2)*, *Sm_Std_Lay(3)* and *Sm_Sml_Std_Lay(5)*) which provide better tasks rejection ratios are those with higher number of scheduler invocations . The graph also shows that the relationship between the tasks rejection ratio and the number of calls to the scheduler can be

extended to the cases of tasks of laxity class B and tasks of laxity class C. In the latter cases, one can observed :

- On one hand, regardless the scheduling algorithms, there is a general increase of the number of calls to the scheduler due to the decrease of the tasks rejection ratios.
- On the other hand, contrary to laxity class A, multi-shape algorithms that use smaller size tasks versions exclusively as additional versions also record a higher number of scheduler calls. For example, in the case of the *Sml_Std(2)* and *Sml_Std_Lay(3)* algorithms, the number of calls to the scheduler is among the highest for the laxity class B and especially for the laxity class C. At first sight, these results seem to contradict the primary intuition, as *Sml_Std(2)* and *Sml_Std_Lay(3)* algorithms reject more tasks than the three others first mentioned. However they do not invalidate the intuition because the bigger the laxity, the more the scheduler attempts to place the ready or waiting tasks. The number scheduler invocations increases accordingly.

The graph in the bottom of figure 6.9 gives a right idea of the runtime overhead of each scheduling algorithm. It accumulates the total amount of time devoted to the scheduling algorithm itself by the microprocessor running the algorithm. It results in a multiplication between the two first graphs (the average runtime overhead and the average number of calls). Considering the cumulative sum of the runtime overheads does not invert the trends of the runtime overheads results commented above.

To summarize, the lower the tasks rejection ratio, the higher the cumulative runtime overheads. One can say that when compared with the basic scheduling algorithm, a multi-shape algorithm like *Sm_Std(2)* decreases the tasks rejection ratio from $\sim 21.5\%$ to $\sim 13.5\%$ and raises the reconfigurable array utilization ratio from $\sim 30\%$ to $\sim 37\%$ at the cost of higher runtime overhead. The latter increases from $\sim 39us$ to $\sim 46us$ compared with the basic scheduling algorithm. Especially for the laxity classes B and C, one can observe that the two multi-shape algorithms which additional tasks versions are exclusively smaller versions (e.g. *Sml_Std(2)* and *Sml_Std_Lay(3)*), achieve the highest cumulative runtime overheads. These runtime overheads culminate at 62% and 60% respectively for the laxity class A. This observation is very important to point out because these algorithms have been also proven earlier to behave poorly in terms of chip utilization ratios and tasks rejection ratios.
Once again, one can see that the *Sm_Std(2)* scheduling algorithm provide the best trade-off when

considering all the metrics assessed above, and the additional efforts that are require to generate two versions of identical sizes per hardware task at design time. Indeed, the algorithm assumes that each task has two versions, the second version being from identical size with the normal version, but with a rectangular shape that suits to the underlying areas partitioning strategy used by the scheduler. Regarding the *Shuffle(1...5)*; the algorithm assumes a more realistic scenario where the number and the size of the versions are randomly generated. *Shuffle(1...5)* brings a significant improvement that demonstrates the usefulness of the multi-shape approach.

6.4.4 Conclusion on multi-shape scheduling

Multi-shape tasks approach shows through the above results that it can significantly improve the scheduling and placement quality. This improvement can be obtain from two versions per task, as far as the size and the aspect ratio of the second version is choosing in order to match with the areas management strategy used. Thus, simulation results have shown that the *multi-shape* approach is not only about the number of versions per task, but above all a question of trade-off between the number of versions per task, the areas partitioning strategy, and the tasks laxity. For example, generating smaller size extra version(s) for low laxity tasks is not recommended, as it does not bring any improvement. However, same size standing versions tasks are recommended when the free areas are vertically split. Indeed, vertically split areas fit standing tasks the best. All the improvements brought by multi-shape scheduling algorithms are at the cost of acceptable runtime overheads, as shown by the results. Therefore, it is worth using multi-shape scheduling in an online real time context, even if additional efforts are required at design time to generate several versions per hardware task.

6.5 Horizon Looking-Ahead Scheduling Algorithms

Hereinafter are presented preliminary simulation results of the *looking-ahead* scheduling algorithms that were presented in the previous chapter. *EAAF* and *SFAF* are the two first algorithms. They use the ternary tree structure proposed in Chapter 5, section 5.4.1 as an enabling structure for looking-ahead scheduling. Therefore the two algorithms use a 2D placement strategy. The third algorithm is the *1D variable slots looking-ahead scheduling* also described in the previous chapter, section 5.3.2.

6.5.1 Horizon Looking-Ahead Scheduling using a Ternary Tree

A ternary tree that suits to the management of the reconfigurable array for *looking-ahead* scheduling algorithms have been discussed in the previous chapter, section 5.4.1. Two variants of *horizon-looking-ahead* scheduling denoted as *horizon-EAAF* and *horizon-SFAF* were detailed. These algorithms are compared with two *without-looking-ahead* scheduling algorithms: the *Basic* scheduling and the *EDF* scheduling. The latter are described in Chapter 5. *Basic* scheduling tries to place the tasks in a *first come first served* basis. Thus, tasks in the ready list are sorted accordingly.

The simulations have been conducted with 20 sets of 50 aperiodic tasks with arbitrary arrival time. A *vertical split* partitioning strategy has also been used as shown in figure 5.4, page 190. Other parameters were uniformly distributed within the intervals as follows :

- FPGA size : $width = 96$ and $height = 64$. Tasks of size $\in [50, 1500]$ CLBs
- Tasks aspect ratio $\in [\frac{1}{5}, 5]$, laxity $\in [11, 50]$ which corresponds to laxity class B. Execution time $\in [5, 100]$.

The results are grouped in table 6.2.

Algorithms	$U_{av}(\%)$	$U_{max}(\%)$	$Rj_{av}(\%)$	Rd_{av}	Sc_{av}	mk_{av}
Basic	30.6	62	27.2	**34**	64	28.2
EDF	31.6	60.8	23.8	**31.8**	66	33
horizon-EAAF	30.5	55.5	24.6	**0**	60	33
horizon-SFAF	32.2	52.5	23.5	**0**	63	33

U_{av} : *FPGA utilization ratio* U_{max} : *Maximum FPGA utilization ratio* $= max(U_{av})$

Rj_{av} : *Tasks rejection ratio* Rd_{av} : *Task rejection delay* $= t_{rej} - a_i$

t_{rej} : *Task rejection time* Sc_{av} : *Number of scheduler calls*

mk_{av} : *makespan*

Table 6.2: Simulation results for looking-ahead scheduling using a ternary tree : comparison with basic scheduling and EDF scheduling.

Chip average utilization ratio U_{av} and tasks rejection ratio Rj_{av}

One can observe that the average utilization ratio U_{av} is around $\sim 30\%$ no matter which scheduling algorithm (looking-ahead and not-looking-ahead) is used. This result has already been found

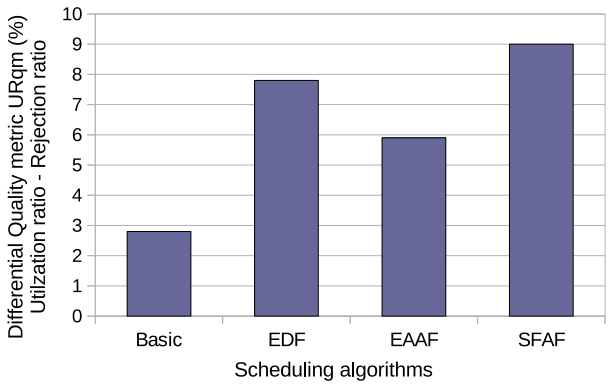


Figure 6.10: Differential quality metrics for horizon-EAAF, horizon-SFAF, Basic and EDF scheduling algorithms.

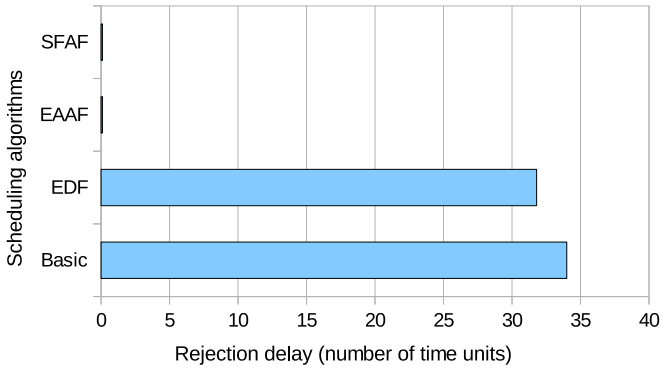


Figure 6.11: Rejection delay for horizon-EAAF, horizon-SFAF, Basic and EDF scheduling algorithms.

earlier for not-looking-ahead scheduling while presenting the simulation results of the tasks parameters based scheduling algorithms. It can be concluded that the chip utilization ratio highly depended on the placement strategy used. The *multi-shape* approach presented a way of improving the U_{av}. The worth of *horizon looking-ahead* scheduling algorithms is not in terms of reconfigurable array utilization ratio. Indeed, horizon (and even stuffing) scheduling accepts or rejects each task as soon as it is released. As a task cannot be rescheduled once accepted and planned, it can prevent the scheduling algorithm for finding a better scheduling for the future. Consequently, a looking-ahead scheduling cannot significantly outperform a not-looking-ahead scheduling that use the same underlying placement strategy. The same similarities can be observed on average

tasks rejection ratios Rj_{av} for the same reasons above. However, *SFAF* slightly outperforms other algorithms in terms of placement quality (U_{av} and Rj_{av}). This is emphasized by the differential quality metric depicted in figure 6.10.

Rejection delay $\mathbf{Rd}_{av}$

The real contribution of looking-ahead scheduling algorithms is on the rejection delay Rd_{av}. As show in the table, horizon-EAAF and horizon-SFAF algorithms immediately reject the tasks (Rd_{av}=0) which cannot fit in the reconfigurable array, while *not-looking ahead* scheduling (Basic and EDF) rejects them with a delay that is equal to their laxity (after $\approx$ 30 time units). Remember that a real-time task that is rejected so late prevent the scheduler from finding any other resource that can execution the task. The graph in figure 6.11 highlights the rejection delay.

Conclusion

The above results have shown which improvements are really brought by a looking-ahead scheduling approach. As only the *horizon looking-ahead scheduling* algorithms were used here, these results can be improved by a *stuffing* approach. The *stuffing looking-ahead scheduling* schedules the tasks even on currently unused parts of reserved areas. For example, as indicated in figure 5.4 (c') page 190, area nodes 5X2 and 2X6 can be used by other tasks in the time interval $[3, 8]$ without affecting the reservation made at node 7X6 for task T_3 to be started at time $t = 8$. This corresponds to the stuffing algorithms.
The runtime overheads of *horizon-EAAF* and *horizon-SFAF* scheduling algorithms have not been measured for these algorithms, contrary to others previously studied. As stated in Chapter 4, *looking-ahead* scheduling are far more time consuming compared to *without-looking-ahead* scheduling. However, on one hand the placement strategy used in the present case was simple as it relied on the ternary tree proposed in this thesis, on the other hand, one can observe in table 6.2 that the number of scheduler invocations is lower for looking-ahead scheduling algorithms.

6.5.2 1D Variable Slots Looking-Ahead Scheduling

This section presents the simulation results of a *1D variable size slots based looking-ahead* scheduling, the *1D-VSSH*. Details on the *1D variable size slots horizon (1D-VSSH)* scheduling algorithm can be found in Chapter 5, section 5.3.2.
The simulation parameters are identical to those used for the algorithm discussed in the previous

section, in terms of size of the reconfigurable array, size of tasks, number of tasks sets, arrival time, execution time, etc. However, the simulations were conducted only on tasks of laxity class B, where the laxity $l \in [11, 50]$.

In *1D variable size slots looking-ahead* scheduling, the number of slots depends on the size of arrival tasks. However, in the following simulations, the maximum number of slots were limited to $n_{max} = 6$. In addition, as the width of a slot is determined by the width of the first tasks in the slot, a width ratio $wr = \frac{task\_width}{slot\_width}$ has been defined. Therefore, while placing the first task in a slot that is empty and wider, an extra slot will be generated if and only if :

$$\begin{cases} slot\_width \geq \frac{FPGA\_width}{n_{max}} \\ wr \leq \frac{1}{4} \end{cases} \tag{6.3}$$

The 1D variable size slots horizon scheduling (1D-VSSH) is compared to the 1D scheduling and the 2D horizon scheduling, in terms of tasks rejection ratio and reconfigurable array utilization ratio. The results are enough to draw major conclusions on *1D variable size slots horizon scheduling*. As discussed in the previous chapter while presenting the algorithm, combining a low complexity (e.g. 1D-like) placement strategy with a looking-ahead scheduling approach is beneficial in terms of runtime overheads, especially when it does not increase the tasks rejection ratio and/or decrease the reconfigurable array utilization ratio.

Figure 6.12 depicts the simulation results of the tasks rejection ratio, the utilization ratio and the differential quality metric.

One can first observe that using a pure 1D placement strategy gives the worst results experienced so far with tasks of laxity class B. Hence, the *1D horizon* scheduling rejects more than 50% of tasks (51,2% exactly), where the *2D horizon* scheduling rejects 30%. The reconfigurable array utilization ratio does not exceed 22% and the differential quality metric is far negative ($\sim -30\%$). The differential quality metric was formerly defined as another way of expressing in a single metric, a placement metric that takes into account both the tasks rejection ratio and the reconfigurable array utilization ratio. This negative differential quality metric shows that a simple 1D placement strategy leads to a poor placement quality. All the *tasks parameters based* scheduling algorithms presented earlier would lead to slightly similar results if they use a 1D placement strategy. Indeed, when discussing the previous results, special attention was paid to the fact that when they use the same placement algorithm, a *looking-ahead* scheduling approach would not significantly perform better than a *tasks parameters based* scheduling algorithm in terms of tasks rejection ratio and array utilization ratio. However, a *looking-ahead* approach takes rapid scheduling decisions, a feature especially suitable for online real-time scheduling.

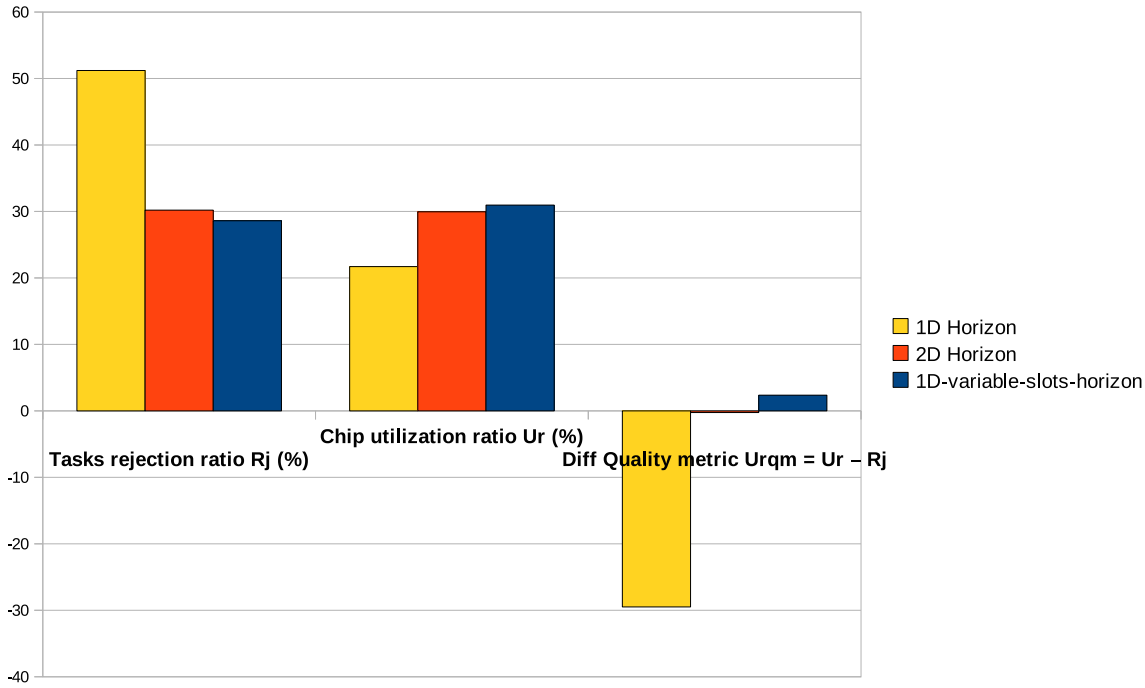


Figure 6.12: Tasks rejection ratio, reconfigurable array utilization ratio and differential quality metric for the proposed *1D variable slots horizon* scheduling, compared to 1D and 2D horizon scheduling from Steiger et al. (2004)

Interesting results are obtained when an 1D placement is combined with a slots-based area management. For example, *2D horizon* scheduling and *1D variable slots horizon* (1D-VSSH) scheduling achieve a slightly similar placement quality. They reject respectively 30% and 28.6% of the tasks, and both occupy $\sim 30\%$ of the reconfigurable array. Furthermore, *1D variable slots horizon* is a better scheduling algorithm as its differential quality metric is higher. Therefore, this result points out how beneficial the use of a slots-based placer in a looking ahead scheduling strategy. The simulation on the runtime overheads was not performed for this algorithm.

6.6 Conclusion of the Chapter

This chapter has presented the results of numerous simulations that were conducted. It first presented the simulation environment along with the parameters of the tasks sets. The output data (metrics) resulting from simulations were then collected. They were analyzed and com-

pared according to different scheduling algorithms. The results showed the great influence of the underlying placement strategy on scheduling algorithms. Thus, whenever possible, two scheduling algorithms were compared only if they were relying on the same placement strategy. Regardless of the considered metric, the suitability of the *multi-shape scheduling* approach for online real-time scheduling has been deeply established.

Regarding the *looking-ahead scheduling* approach; the results also showed that when the *looking-ahead scheduling* relies on a *1D placement* strategy combined with a *1D slots-based* areas management, it provided a slightly similar if not better performance in terms of tasks rejection ratio and array utilization ratio, compared to a *looking-ahead scheduling* that uses a 2D area management. In addition, the so-called *1D-variable-slots horizon looking ahead* scheduling may have a far lower runtime overhead.

Additional simulation results are presented as it is at the end of the thesis, in *Appendix B* (page 248), *Appendix C* (page 256) and *Appendix D* (page 260). In these simulations results, as many scheduling algorithms as possible are displayed on a single graph, giving more opportunity to compare their performance. Hence, such a graph can be a rich source of information on the behavior of scheduling and placement algorithms along with their influence on different scheduling and placement metrics that are meaningful in real-time multi-tasking on reconfigurable hardware devices.

The next chapter concludes the thesis and discusses future work.

Chapter 7

Conclusion and Future Work

7.1 Discussions

Embedded electronic devices have become part of everyday life. From DVB[1] to hand-held devices, the technology is becoming ubiquitous and the requirements of embedded applications in terms of computational power are skyrocketing. Moreover, these requirements are in addition to stringent constraints such as cost, size, power consumption, high data rate, shorter life cycle, etc. Nowadays, thanks to improvements in semiconductor technology, entire systems can be compressed onto a sliver of silicon that is smaller than a penny. The so-called System-on-a-Chip has become the solution as it combines the key features mentioned above. However, on one hand, this advanced in semiconductor technology is very costly, resulting in increasingly high non recurring engineering (NRE) costs. This thesis has discussed how using dynamically reconfigurable hardware devices can lower the NRE. On the other hand, SoC complexity constantly increases as it integrates heterogeneous components (including reconfigurable parts) in order to provide the required computational power. Thus, as SoC complexity increases, their design process needs to undergo significant transformation.

The original objective of the thesis was to address problem of scheduling online real-time hardware tasks on the partially and dynamically reconfigurable part of the SoC, and subsequently build a library of scheduling/placement algorithms for an RTOS-driven Reconfigurable-SoC design space exploration. Therefore, combining these two reasons made this thesis unique, as the investigated issues intervene at two stages :

[1] Digital Video Broadcasting

- ***After the design stage (at runtime);*** where scheduling and placement algorithms that enable online real-time scheduling of hardware tasks on the reconfigurable part of a SoC were investigated. These online real-time scheduling and placement algorithms were required to provide a reasonable trade-off between their time complexity and their performance in terms of chip utilization ratio, tasks rejection ratio, etc.
- ***Prior to and during the design stage (offline);*** where scheduling and placement algorithms dedicated to the reconfigurable part of a Reconfigurable SoC are required while exploring the design space of the SoC. It corresponds to the part of this work which consisted of implementing, assessing and classifying as many scheduling and placement algorithms as possible, and subsequently integrating them in a library. For example in the design methodology presented in Chapter 2 and depicted in figure 2.19 (page 60), using such a library of algorithms at system level can help to perform a more accurate partitioning of the application, and thus to refine the architecture of the Reconfigurable SoC with respect to system and application constraints.

7.2 Key Contributions

7.2.1 Algorithms for Online Real-time Scheduling/Placement on DPRHWs

Scheduling algorithms can be classified in two families, namely, *looking-ahead* scheduling and *without-looking-ahead* scheduling. The two scheduling families differ on the way the availability of the areas on the reconfigurable array is expressed, and on the way the algorithms check these available areas and assign them to tasks. To put it simply, *without-looking-ahead* scheduling keeps a task as far as it can still meet its deadline, and attempts to place it. Conversely, a *looking-ahead* scheduling decides as a task arrives, if it is rejected or accepted. Therefore, the rejection delay is equal to zero in the latter case. Depending on the family, the scheduling algorithm requires different quantity of placement operations.

1. ***Looking-ahead scheduling*** algorithm considers present and future states of the reconfigurable array to detect areas that are currently free, or those who will be free in the future. Thus, the scheduler knows whether the task can fit on the array presently or later. For example, an area occupied at the present time can be reserved to accommodate a task that starts later.

   The significant advantage of *looking-ahead* approach over *without-looking-ahead* approach is

its ability to take very fast scheduling decisions. This last feature is very useful for scheduling online real-time tasks. However, this speediness comes at the cost of numerous areas management operations. The latter operations mimic future tasks ending and starting impact on the reconfigurable array, in order to properly schedule arriving tasks. Consequently, the placement algorithms must be of low complexity to make the looking-ahead scheduling approach affordable in terms of runtime overhead. The contribution of the thesis on *online looking-ahead scheduling* is as follows :

(i). The study in the Chapter 4 section 4.2.2 demonstrated through a cycle accurate runtime overhead measurements that MERs-based placement strategies are very time consuming and, therefore, are not suited to online real-time *looking-ahead* scheduling. To the best of our knowledge, such cycle accurate timing measurements on MERs-based algorithms have not been conducted before on an embedded processor.

(ii). The simulation results detailed and discussed in Chapter 6 section 6.5.1 has shown that when they rely on the same placement strategy, *looking-ahead scheduling* and *without-looking-ahead scheduling* algorithms provide slightly similar performance in terms of tasks rejection ratio and reconfigurable chip utilization ratio. This finding suggests that *looking-ahead scheduling* is a worthy approach only in special cases where rapid scheduling decisions are required (e.g. online real-time scheduling on heterogeneous platforms that provide more than one implementation alternative). Indeed, equation 3.13 (page 99) shows that with *looking-ahead scheduling*, a task that cannot fit on the reconfigurable array is rejected immediately, allowing for alternative implementation solutions.

(iii). A new metric denoted as *differential quality metric URqm* was introduced. This metric better reflects the difference between two scheduling algorithms which seem at first glance identical in terms of utilization ratio and tasks rejection ratio.

(iv). A *ternary tree structure* were proposed and developed. The tree were inspired from the binary tree structure presented in Bazargan et al. (2000) and improved in Walder et al. (2003). The originality of the proposed ternary tree structure is its suitability for keeping the information on present and future states of the reconfigurable array.

Thus, the tree eases a *2D horizon looking-ahead* scheduling approach by providing a good visibility for future states of the array.

Additionally, two variants of *2D horizon looking-ahead* scheduling that use the tree were proposed. They are denoted as *horizon-SFAF* scheduling and *horizon-EAAF* scheduling algorithms. As shown by the simulation results in section 6.5.1 page 219, the *horizon-EAAF* scheduling algorithm performs better than the *horizon-SFAF* scheduling and the EDF scheduling in terms of tasks rejection ratio and chip utilization ratio. The differential quality metric confirms this result.

(v). *1D variable size slots horizon (1D-VSSH)* scheduling algorithm has been proposed and studied. The algorithm combines the main advantage of the the *looking-ahead scheduling* (which is its very short task rejection delay as expressed in equation 3.13, page 99) and the simplicity of a *1D placement strategy*. The simulation results discussed in detail in section 6.5.2 shows that, by dynamically partitioning the reconfigurable array in slots, and by applying a 1D placement in each slot, there is no loss in terms of tasks rejection ratio and chip utilization ratio, compared to a 2D placement. Furthermore, the *1D variable size slots horizon (1D-VSSH)* is similar if not slightly better in performance. This result is important since it suggests that in a *looking-ahead scheduling* approach, one can avoid the use of a time consuming 2D placement strategy without sacrificing placement quality.

2. ***A without-looking-ahead*** scheduling algorithm places the tasks only on currently available areas. Thus, when a fitting place is found, the task is place immediately and starts its execution. The contribution of the thesis on *online without-looking-ahead scheduling* is as follows :

   (i). Several *tasks parameters based* scheduling algorithms were developed. Their simulation results have shown that the performance of a scheduling algorithm in terms of tasks rejection ratio and reconfigurable array utilization ratio, highly depends on the quality of the underlying placement strategy. This finding was highlighted by a comparison of simulation results of these scheduling algorithms when they used the same placement strategy. According to these results discussed in the previous chapter (section 6.3), these algorithms do not significantly differ from each other in terms of utilization ratio and tasks rejection ratio. However, the BSF (biggest size first) al-

gorithm is slightly better, particularly in terms of *differential quality metric*, a more sensitive metric.

(ii). The *Multi-shape scheduling* approach was one of the key proposal of this work, to solve the issue of scheduling online real-time tasks on partially and dynamically reconfigurable hardware devices. The ability of the proposed approach to significantly improve the placement quality when only two versions per task are provided and when the area partitioning strategy suits to the aspect ratio of the tasks has been proven. The multi-shape scheduling algorithms provide far better results than any other algorithms, without significantly increase the runtime overheads.

7.2.2 Scheduling/Placement algorithms library for RTOS-driven design space exploration

In addition to proposing new online real-time scheduling algorithms for reconfigurable hardware devices, an important part of this thesis was devoted to the study, implementation and evaluation of existing algorithms in related work. The list of scheduling algorithms and placement heuristics that have been implemented within the framework of this work are shown in the appendix of this thesis in table 7.3 (page 256), table 7.4 (page 257), table 7.5 (page 258) and table 7.6 (page 259). The simulations results are also gathered in the *Appendix B* and presented as shown. Gathering the results of so many scheduling algorithms on the same graphs allows the designer to point out the advantages and drawbacks of each algorithm or class of algorithms, and to find suitable trade-offs between the metrics of the algorithms. The relevent metrics are the utilization ratio, the tasks rejection ratio, the differential utilization metrics, the algorithms runtime overheads, the makespan, etc. As discussed in Chapter 2, and shown in figure 2.19, these algorithms can be used to refine the dynamically reconfigurable part of the SoC, during the system level simulation. The algorithms were designed purposely in C++ language in order to insure a full compatibility with any C++/SystemC based SoC design methodology. The algorithms are based on various models of hardware tasks, reconfigurable array, scheduler and placer that are reusable and refinable.

7.3 Hypothesis and Limitations

The original hypothesis of the thesis can be summarized as follows :

- Hardware tasks are rectangularly shaped and relocatable, but cannot be rotated.
- The reconfigurable hardware device is partially and dynamically reconfigurable.
- A hardware task fits in a rectangular space of the reconfigurable device as far as there is enough contiguous free resources to accommodate the task.
- The online scheduling paradigm used in this thesis corresponds to the online clairvoyant paradigm.
- The scheduling is said *real-time* because every task in the system is submitted to a deadline constraint. A hardware task that cannot fit on the reconfigurable array and meet its deadline is rejected. However, rejecting a task does not lead the failure of the application, because it is assumed that there are other implementation alternatives for the task.
- Aperiodic tasks systems are used. They are likely to better reflect a very dynamic system where the release time of each job is totally unknown beforehand.
- Tasks parameters are statistically independent.
- It is assumed that there is a communication media that allows the tasks to communicate independently to their location on the chip. The thesis does not deal with the inter-tasks communication issues.

The aforementioned assumptions and limitations are widely accepted by the research community in Reconfigurable Computing to study the problem of scheduling and placing hardware tasks on DPRHWs. These assumptions rely on technological advances in devices such as FPGAs, and on research topics that address other problems in the field (e.g. communication, tasks relocation, tasks migration, etc.).

Though many optimal and non optimal scheduling and placement algorithms have been proposed in related work, their suitability for online real-time scheduling was not established, especially in an embedded environment. The foundation of this thesis was a methodology which first assessed the limitations of some areas management approaches in such an environment. Accordingly, few scheduling and placement methods have been proposed and discussed. In addition, combination of scheduling and placement strategies have been suggested.

The proposed solutions have been proven to improve the scheduling and placement quality without significantly modifying the algorithms complexity and runtime overheads. These results prove the primary hypothesis that resulted from the methodology proposed in this thesis.

7.4 Future Work

Several scheduling algorithms have been studied in Chapter 5. Most of these algorithms have been implemented and simulated, and the simulation results discussed. However, few have been either partially implemented or partially simulated, and some others have not been implemented.

In this thesis, the *n X 1D variable size slots scheduling* relies on an areas management strategy that has been used only with a *horizon-looking-ahead scheduling* algorithm. In Chapter 5, three variants of the algorithm, namely First Fit (FF), Next Fit (NF) and Best Fit (BF) have been discussed. Only the BF variant has been combined with the *horizon-looking-ahead scheduling* and simulated. Any experiment has been carried out with a *without-looking-ahead scheduling*.
An extension of this research could be to replace the horizon schedulings by *stuffing schedulings* in order to assess the improvements that are obtained.

Multi-shape scheduling algorithms have been proven to improve the scheduling and placement quality. *Looking-ahead scheduling* algorithms have been proven to be suitable for online real-time scheduling if they rely on a placer with acceptable complexity. We have separately assessed the two scheduling approaches. Combining them will improve the scheduling quality. On one hand, the *multi-shape scheduling* will increase the chip utilization ratio and decrease the tasks rejection ratio, while the *looking-ahead* approach will allow the scheduler to take fast scheduling decisions, which is very useful in an online real-time context.

In this thesis, the complexity of designing *multi-shape* hardware tasks has not been clearly studied. However, the assumptions made above on hardware tasks are the same for *multi-shape* hardware tasks (relocatability, etc.). The simulation results has shown that using two versions per tasks is sufficient to significantly improve the placement quality. Therefore, it can be roughly assumed that the *multi-shape scheduling* approach requires about twice as much memory as the single-version tasks scheduling approach, to store the bitstreams.

The *multi-shape scheduling algorithm* have been proposed in Chapter 5 as a solution for improving the scheduling and placement quality, in terms of tasks rejection ratio, chip utilization ratio and scheduling algorithm runtime overheads. The same algorithm could be applied to power-aware embedded systems. The power consumption of a hardware task consists of two main parts. The part due to the power and the time required to configure the task on the reconfigurable array, and the part due to the power dissipated by the hardware task when it runs to completion. In both cases, the power depends on the size of the task, in addition to other parameters. As the multi-shape tasks scheduling assumes that each hardware task can be instantiated in more than one size and/or shape, a *power-aware multi-shape scheduling* approach will always choose the instantiation that minimizes the power consumption of the chip.

A possible extension of this work may be to consider the preemption of hardware tasks.

The *competitive analysis* has been presented in Chapter 3 as the best tool for assessing online scheduling algorithms. The analysis has been discussed for software tasks scheduling on monoprocessor and multiprocessor systems, with the aim to transpose to hardware tasks scheduling on reconfigurable devices. This transposition is far from being obvious, as scheduling hardware tasks on a reconfigurable array is further complicated than multiprocessor scheduling. Thus, the *competitive analysis* approach has been used only in the study of the *1D variable slots scheduling with minimum makespan* in Chapter 5, section 5.3.3. As stated at the beginning of Chapter 6, for the sake of consistency, an *average case analysis* has been used instead, while carrying out the experiments. Using competitive analysis in online scheduling of hardware tasks on dynamically and partially reconfigurable hardware devices remains a niche.

Through lack of time, the implementation of a real life application on an embedded RSoC following the OS-driven RSoC design methodology discussed in Chapter 2 and Chapter 4 has not been done in this thesis. The so-called *OveRSoC methodology* has been proven suitable for exploring the design space of an an MPSoC (Miramond et al., 2009a). A case study for implementing a mobile robotic vision application on an MPSoC was presented in Verdier et al. (2008). However, the latter case was targeting an MPSoC without a dynamically reconfigurable hardware part. The next step in this work could be to include the dynamically reconfigurable hardware devices models along with the placer model, the hardware tasks models and the scheduling/placement strategies provided by this work into the methodology. Hence, as depicted in figure 2.19 (page

60), these models and algorithms will be taken into account by the architecture and RTOS services exploration strategy while refining the final architecture for the case study.

Publications

This thesis will be summarized and submitted as a Journal Paper.

Refereed Conference Papers

1. G. Wassi, G. Lawday, M-E-A. Benkhelifa, F. Verdier, "Online Real-time Scheduling of Multi-shape Hardware Tasks on Partially Reconfigurable FPGAs", In the proceedings of 22th National Symposium of Research Group on Signal and Images Processing, GRETSI 2009, 8-11 septembre 2009, Dijon, France.

Non-Refereed Workshop/Conference Papers

1. G. Wassi, M-E-A. Benkhelifa, F. Verdier, G. Lawday, "Tree Structure for Online Real-time Scheduling on Partially Reconfigurable FPGAs", In the proceedings of 4th National Symposium of Research Group on System-on-Chip & System-in-Package, GDR SoC/SiP 010, 2010, Paris, France.
2. Guy Wassi, Geoff Lawday, Amine Benkhelifa, Francois Verdier, "Online Scheduling and Placement of Real-Time Hardware Tasks on FPGAs", In the proceedings of 3rd National Symposium of Research Group on System-on-Chip & System-in-Package, GDR SoC/SiP 009, 2009, Orsay, France.

Bibliography

Leon Adam. Choosing the right architecture for real-time signal processing designs. White paper spra879, Texas Instruments, Nov 2002.

A. Ahmadinia, C. Bobda, and J. Teich. A dynamic scheduling and placement algorithm for reconfigurable hardware. *Architecture of Computing Systems (ARCS)*, pages 125–139, 2004.

Elias Ahmed and Jonathan Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 3–12, New York, NY, USA, 2000. ACM.

Susanne Albers and Bianca Schröder. An experimental study of online scheduling algorithms. *ACM Journal of Experimental Algorithmics*, 7:154, 2002.

Altera. www.altera.com. URL `http://www.altera.com`.

Altera. Introducing innovations at 28 nm to move beyond moore's law. Technical report, Altera Corp., 2010a. URL `http://www.altera.com/literature/wp/wp-01125-stxv-28nm-innovation.pdf`.

Altera. Altera unveils innovations for 28-nm fpgas. Technical report, Altera Corp., 2010b. URL `http://www.altera.com/corporate/newsroom/releases/2010/products-/nr-innovating-at-28-nm.html`.

P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, 3(26):11–18, 1993.

I. Bandara and C. Hudson. Detection and tracking of eye blink to identify driver fatigue and napping. In *HCI 2006: Engage!, The 20th BCS HCI Group conference in co-operation with ACM. London, UK.*, 11-15 September 2006.

V. Baumgarte, G. Ehlers, F. May, A. Naeckel, M. Vorbach, and M. Weinhardt. PACT XPP—a self-reconfigurable data processing architecture. *Journal of Supercomputing*, 26(2):167–184, 2003.

K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. In *IEEE Design and Test for Computers*, volume 17, pages 68–83, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

John Blyler. Navigating the silicon jungle: FPGA or ASIC ? *Chip Design Magazine*, June release, 2005.

Ivo Bolsens. The future of FPGAs. *Victorian Microelectronics Designers Network. Event, Melbourne*, pages 21–30, Jan 27 2005.

Celoxica. Handel-c language. Technical report, Celoxica, 2000. URL `www.celoxica.com`.

K. B. Chehida and M. Auguin. HW / SW Partitioning Approach For Reconfigurable System Design. *CASES 2002*, 2002.

Yuan-Hsiu Chen and Pao-Ann Hsiung. Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc. In *EUC*, pages 489–498, 2005.

E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In PWS Publishing Company, editor, *In D. Hochbaum, Approximation algorithms*, 1997.

K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, 2002.

Jin Cui and Qingxu Deng. An efficient algorithm for online management of 2d area of partially reconfigurable fpgas. In *in Proc. Design, Automation and Test in Europe (DATE*, pages 129–134, 2007.

K. Danne. *Real-Time Multitasking in Embedded Systems Based on Reconfigurable Hardware*. PhD thesis, PhD thesis, University of Paderborn, Germany, September 2006.

K. Danne and M. Platzner. Partitioned Scheduling of Periodic Real-Time Tasks onto Reconfigurable Hardware. In *International Parallel and Distributed Processing Symposium (IPDPS'06), Reconfigurable Architecture Workshop (RAW'06)*, 2006a.

K. Danne and M. Platzner. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In *International Conference on Field Programmable Logic and Applications*, pages 24–26, 2005.

K. Danne and M. Platzner. An EDF schedulability test for periodic tasks on reconfigurable hardware devices. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 93–102. ACM New York, NY, USA, 2006b.

K. Danne, R. Muhlenbernd, and M. Platzner. Executing hardware tasks on dynamically reconfigurable devices under real-time conditions. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6, 2006.

André DeHon. The density advantage of configurable computing. *IEEE Computer Society*, 33: pages 41–49, 2000.

C. Ebeling, Cronquist DC., and Franklin P. Rapid reconfigurable pipelined datapath. *Lecture Notes in Computer Science 1142 - Field Programmable Logic: Smart Applications, New Paradigms and Compilers*, pages 126–135, 1996.

Jeff Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

Fujitsu. White paper: Fujitsu develops new soc design methodology based on uml and c. Technical report, 2002.

A. Gatherer, S. Sriram, F. Moerman, Sengupta C., and K. Brown. *Cost Effective Software Radio for CDMA Systems. In: Walter H. W. Tuttlebee, Software Defined Radio: Baseband Technologies for 3G Handsets and Basestations.* ed John Wiley & Sons., England, 2004.

M. Gokhale and J. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98), 15-17 April 1998, Napa Valley, CA, USA*, pages 126–. IEEE Computer Society, 1998.

M. B. Gokhale and P. S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays.* Springer, 1 edition, 2006.

Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–, Washington, DC, USA, 2000. IEEE Computer Society. URL `http://portal.acm.org/citation.cfm?id=795659.795916`.

Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, R. Reed Taylor, and R. Reed. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33:70–77, 2000. URL `http://www.cs.cmu.edu/~seth/papers/goldstein-ieee00.pdf`.

R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In E.L. Johnson P.L. Hammer and B.H. Korte, editors, *Discrete Optimization II,*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979. URL `http://www.sciencedirect.com/science/article/B8G56-4SD21YG-M/2/4b302b1ea464cf17986f7e4642be86a1`.

M. Handa and R. Vemuri. Area fragmentation in reconfigurable operating systems. In *International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 77–83, 2004a.

M. Handa and R. Vemuri. An integrated online scheduling and placement methodology. In *Field Programmable Logic and Applications (FPL'04). . International Conference on*, pages 444–453, 2004b.

Manish Handa and Ranga Vemuri. An efficient algorithm for finding empty space for online fpga placement. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 960–965, New York, NY, USA, 2004c. ACM. URL `http://doi.acm.org/10.1145/996566.996820`.

R. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 564–570, New York, NY, USA, 2001a. ACM.

R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001b. IEEE Press.

Paul Heysters, Gerard Smit, and Egbert Molenkamp. A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems. *J. Supercomput.*, 26(3):283–308, 2003.

ImpulseC. Impulsec language. Technical report, Impulse Accelerated Technologies, www.impulseaccelerated.com. URL `http://www.impulseaccelerated.com`.

P. A. Jackson J. L. Tripp and B. L. Hutchings. Sea cucumber: A synthesizing compiler for fpgas. *In Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, 2002.

Ahmed A. Jerraya. Long term trends for embedded system design. In *DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems*, pages 20–26, Washington, DC, USA, 2004. IEEE Computer Society.

Chi jui Chou, Satish Mohanakrishnan, and Joseph B. Evans. Fpga implementation of digital filters. In *In Proceedings of the Fourth International Conference on Signal Processing Applications and Technology*, pages 80–88, 1993.

A. Kaplan, M. Sarrafzadeh, and R. Kastner. A survey of hardware/software system partitioning. Technical report, 2003.

D. Koch and J. Torresen. Advances and trends in dynamic partial run-time reconfiguration. Department of Informatics, University of Oslo, Norway, 2010.

Markus Koester, Mario Porrmann, and Heiko Kalte. Task placement for heterogeneous reconfigurable architectures. In *FPT*, pages 43–50, 2005.

Markus Koester, Heiko Kalte, and Mario Porrmann. Relocation and defragmentation for heterogeneous reconfigurable systems. In *ERSA*, pages 70–76, 2006.

Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM. ISBN 1-59593-292-5. doi: http://doi.acm.org/10.1145/1117201.1117205. URL `http://doi.acm.org/10.1145/1117201.1117205`.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973. URL `http://doi.acm.org/10.1145/321738.321743`.

Carey Douglass Locke. *Best-effort decision-making for real-time scheduling.* PhD thesis, Pittsburgh, PA, USA, 1986. AAI8702895.

Y. Lu, T. Marconi, G. N. Gaydadjiev, K.L.M. Bertels, and R. J. Meeuws. A self-adaptive online task placement algorithm for partially reconfigurable systems. In *Proceedings of the 22nd Annual International Parallel & Distributed Processing Symposium (IPDPS 2008) - RAW2008*, page 8, April 2008.

P. Mahr, S. Christgau, C. Haubelt, and C. Bobda. Integrated temporal planning, module selection and placement of tasks for dynamic networks-on-chip. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium*, 2011.

T. Marconi, Y. Lu, K.L.M. Bertels, and G. N. Gaydadjiev. Online hardware task scheduling and placement algorithm on partially reconfigurable devices. In *Proceedings of International Workshop on Applied Reconf. Computing (ARC)*, pages 306–311, March 2008.

G. Martin. *The History of the SoC revolution. In: G. MARTIN, H. CHANG, Winning the Soc revolution.* ed. Kluwer Academic Publishers., USA, 2003.

MathWorks. In *www.mathworks.com; www.mathworks.com/products/slhdlcoder;*, www.mathworks.com.

Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. pages 61–70. 2003.

Uwe Meyer-Bäse, Divya Sunkara, Encarnacion Castillo, and Antonio Garcia. Custom instruction set nios-based ofdm processor for fpgas. volume 6248, page 62480O. SPIE, 2006.

J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proceedings of Design, Automation and Test in Europe 2003 (DATE 03), IEEE Computer Society*, pages 986–991, March 2003.

B. Miramond, E. Huck, F. Verdier, Mohamed El Amine Benkhelifa, B. Granado, M. Aichouch, J.-C. Prévotet, D. Chillet, S. Pillement, Th. Lefebvre, and Yaset Oliva. Oversoc : a framework for the exploration of rtos for rsoc platforms. *International Journal on Reconfigurable Computing*, 2009(450607):1–18, dec 2009a. URL `http://publi-etis.ensea.fr/2009/MHVBGAPCPL009`.

B. Miramond, F. Verdier, and M. Aichouch. Dogme distributed operating system graphical modeling environment, 2009b.

J. Noguera and R. M. Badia. Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling. 3(2):385–406, 2004.

V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc. In *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*, Paris, March 2003.

V. Nollet, P. Avasare, D. Verkest, and H. Corporaal. Exploiting hierarchical configuration to improve run-time mpsoc task assignment. In *ERSA'06*, pages 49–55, 2006.

M. Middendorf H. Schmeck O. Diessel, H. ElGindy and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable fpgas. In *In IEE Proc. on Computers and Digital Techniques*, pages 181–188, 2000.

R. J. Petersen. *An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing.* PhD thesis, Master's Thesis, Brigham Young University, 1995.

Polis. A codesign environment. *University of California Berkeley, Center for Electronic System Design, http://embedded.eecs.berkeley.edu/Respep/Research.*

Ptolemy. An environment for modelling, simulation, and design of concurrent real-time embedded systems. *University of California Berkeley, Center for Electronic System Design, http://embedded.eecs.berkeley.edu/Respep/Research.*

Jan M. Rabaey. Wireless beyond the third generation wireless beyond the third generation: facing the energy challenge. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 1–3, New York, NY, USA, 2001. ACM.

Steven Phillips Rajeev Motwani and Eric Torng. Non-clairvoyant scheduling. In *Theor. Comput. Sci*, 1994.

S. Roman, H. Mecha, Mozos D., and Septien J. Partition-based dynamic 2d hw multitasking management. *In Proceedings of the 9 EUROMICRO Conference on Digital System Design (DSD'06)*, 2006.

Jonathan Rose, Abbas El Gamal, Senior Member, and Albert Sangiovanni-vincentelli. Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency. *Proceedings of the IEEE*, 25:1217–1225, 1990.

P. Schaumont and I. Verbauwhede. Thumbpod puts security under your thumb. In *Xilinx Xcell Journal*, 2003.

E. Schüler and L. Tan. *XPP – An Enabling Technology for SDR Handsets. In: Walter H. W. Tuttlebee, Software Defined Radio: Baseband Technologies for 3G Handsets and Basestations.* ed John Wiley & Sons., England, 2004.

Jirí Sgall. On-line scheduling - a survey. In *Online Algorithms: The State of the Art, Lecture Notes in Computer Science 1442*, 1998.

Satwant Singh, Jonathan Rose, Paul Chow, and David Lewis. The effect of logic block architecture on fpga performance. *IEEE Journal of Solid-State Circuits*, 27:281–287, 1992.

D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. In *Communications of the ACM*, page 28:202–208. ACM New York, NY, USA, 1985.

Lodewijk T. Smit, Gerard J.M. Smit, Johann L. Hurink, Hajo Broersma, Daniël Paulusma, and Pascal T. Wolkotte. Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture. In *IEEE International Conference on Field-Programmable Technology, FPT*, pages 421–424. IEEE, 2004. URL `http://doc.utwente.nl/48501/`.

John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

C. Steiger, H. Walder, and M. Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE TRANSACTIONS ON COMPUTERS*, 53(11):1393–1407, 2004.

SystemC. Systemc language, www.systemc.org. Technical report, www.systemc.org.

SystemVerilog. Systemverilog, an unified hardware description and verification language (hdvl) standard, http://www.systemverilog.org. Technical report, www.systemverilog.org.

J. Tabero, J. Septién, H. Mecha, and D. Mozos. A low fragmentation heuristic for task placement in 2d rtr hw management. In *Field Programmable Logic and Applications, 2004. FPL'04. International Conference on*, pages 241–250, 2004.

J. Tabero, J. Septién, H. Mecha, and D. Mozos. Task placement heuristic based on 3d-adjacency and look-ahead in reconfigurable systems. In *Asia and Pacific Design Automation Conference (ASP-DAC)*, pages 396–401, 2006.

Michael Taylor, Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs, 2002.

Tensilica. www.tensilica.com.

R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 3(28):7–27, 2001.

Jan C. Van Der Veen, Sándor P. Fekete, Mateusz Majer, Ali Ahmadinia, Christophe Bobda, Frank Hannig, and Jürgen Teich. Defragmenting the module layout of a partially reconfigurable device. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas*, pages 92–101. CSREA Press, 2005.

F. Verdier, B. Miramond, M. Maillard, E. Huck, and Th. Lefebvre. Using high-level rtos models for hw/sw embedded architecture exploration: Case study on mobile robotic vision. *EURASIP Journal on Embedded Systems*, Special issue on Design and Architectures for Signal Image Processing, 2008. URL `http://publi-etis.ensea.fr/2008/VMMHL08`.

A.S. Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18:23–33, 2001.

H. Walder and M. Platzner. Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform. *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, pages 24–30, 2002.

H. Walder, C. Steiger, and M. Platzner. Fast online task placement on FPGAs: free space partitioning and 2D-hashing. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8, 2003.

Xilinx. In *www.xilinx.com*, www.xilinx.com.

Xilinx. Two flows for partial reconfiguration: Module based or difference based. Technical report, Xilinx Inc., 2004. URL `www.xilinx.com`.

Xilinx. *Distributed Arithmetic FIR Filter V9.0*. Xilinx Inc., San Jose California, April 2005. URL `http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_fir.pdf`.

Xilinx. In *Company reports*, Company reports.

Appendix

7.5 Appendix A : Table classifying related work on scheduling and placement strategies

Area Models	Authors & Algorithms	Strengths	Drawbacks	Comments
Homogeneous Area Model	Bazargan et al. (2000), KAMER MERs-based	Optimal placement (2D)	Very high complexity $O(n^2)$, slow	Static, offline placement. Used as a comparison reference. *Without-looking-ahead* scheduling
	Bazargan et al. (2000), Nonoverlapping	Faster than KAMER, complexity $O(n)$	Quality lost (fragmentation, tasks rejection)	Static, online, BF, FF, BL, Partial reconfiguration (higher fragmentation and tasks rejection), *without-looking-ahead* scheduling
	Walder et al. (2003), OTF & EOTF	Hash matrix. Complex. $O(1)$ for area search	Very long update process: up to $O(w \cdot h)$, width/height of the array	OTF and Enhanced OTF partitioning, Enhanced Bazargan et al. (2000)'s partitioner. Static, online/offline placement (BF, FF, WF), up to 70% better than Bazargan et al. (2000) (nonoverlapping).
	Ahmadinia et al. (2004), Clusters based.	Complex. O(n) Less fragmentation.	1D placement. Not really fragmentation based.	Enhanced Bazargan et al. (2000)'s partitioner. Runtime overhead 15 to 20% better than KAMER, for a slightly similar tasks rejection ratio. Dynamic placement, equal size slots. Prototyped example.
	Roman et al. (2006)	Complex. O(1). Outperforms the First Fit algorithm	Less efficient for heterogeneous (equal size) task sets	Slots with different sizes, runtime size adjustment. Queue for scheduling

Table 7.1: Related work on scheduling and placement strategies (homogeneous reconfigurable array model)

Area Models	Authors & Algorithms	Strengths	Drawbacks	Comments
	Koester et al. (2005) SUPFit & RUPFit algorithms	Both have better 1D-placement than BestFit.	RUPFit has a high Run-time complexity.	SUPFit rejects less tasks. RUPFit has a better utilization ratio and the least relative task rejections. Priority feature on tasks. Heterogeneity (static *vs* configurable cells). BestFit used for benchmarking. Prototyped example.
Heterogeneous	Steiger et al. (2004), Horizon(1D), Stuffing(1D and 2D)	Tasks rejection ratio $\leq$ 10% for the 2D stuffing, models similar to ours	No details on placement strategies used	Online scheduling & placement of real-time tasks. Tasks relocatability, Better scheduling performance with standing tasks. Prototyping of the 1D model using a Xilinx FPGA and the microblaze as host CPU.
	Schaumont and Verbauwhede (2003)	faster reconfiguration, Java security architecture	No runtime tasks assignment feature	The ThumbPod architecture: Hierarchical configuration concept used from a design time point of view. Run-time reconfiguration. Easy design process and programming,
Area	Nollet et al. (2006)	Hierarchical configuration		Hierarchical configuration concept in an MPSoC, Processing Elements allocation.
	Nol	MPSoC approach, Task assignment scheme		The two works are more about tasks assignement in an MPSoC than hardware tasks placement. Configuration hierarchy introduced here can be applied to hardware tasks placement heuristics. Deals with communicating tasks (Network-on-Chip).
Model	Smit et al. (2004)	Run-time tasks assignment. Tasks priority feature.		Heterogeneous area model taken into account. Scarcity of the resources is not used to adjust the Run-time tasks assignment algorithm.

Table 7.2: Related work on scheduling and placement strategies (heterogeneous reconfigurable array model)

7.6 Appendix B : Additional Simulation Results

Here are presented global results of *without-looking-ahead* scheduling algorithms. The results are aggregated in order to provide a global overview. The results for scheduling algorithms that use tasks with a single version are put beside those for the multi-shape scheduling algorithms in order to ease the comparison.

Figure 7.1: The tasks rejection ratio for paramaters based scheduling and multi-shape tasks based scheduling.

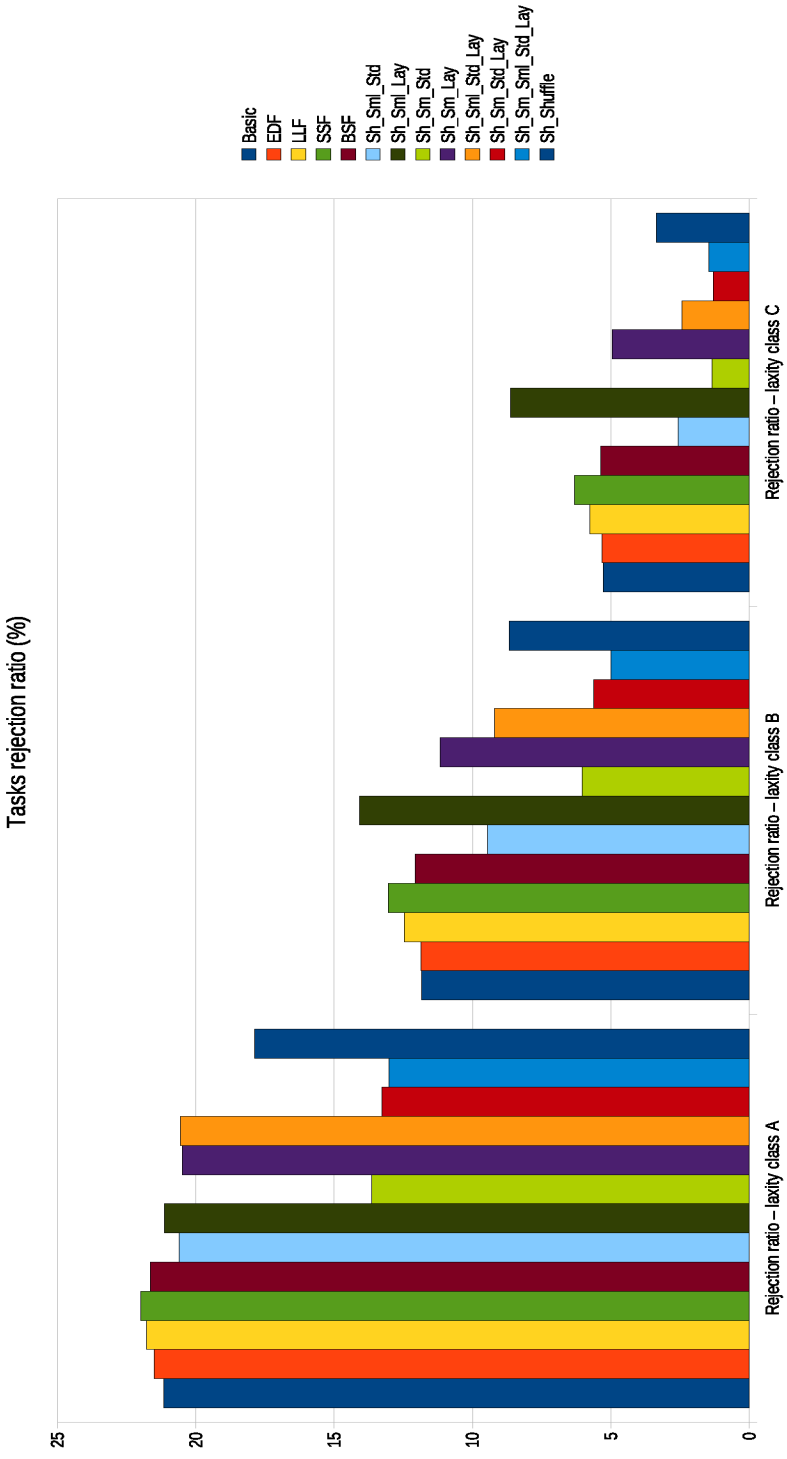


Figure 7.2: The reconfigurable array utilization ratio for paramaters based scheduling and multi-shape tasks based scheduling.

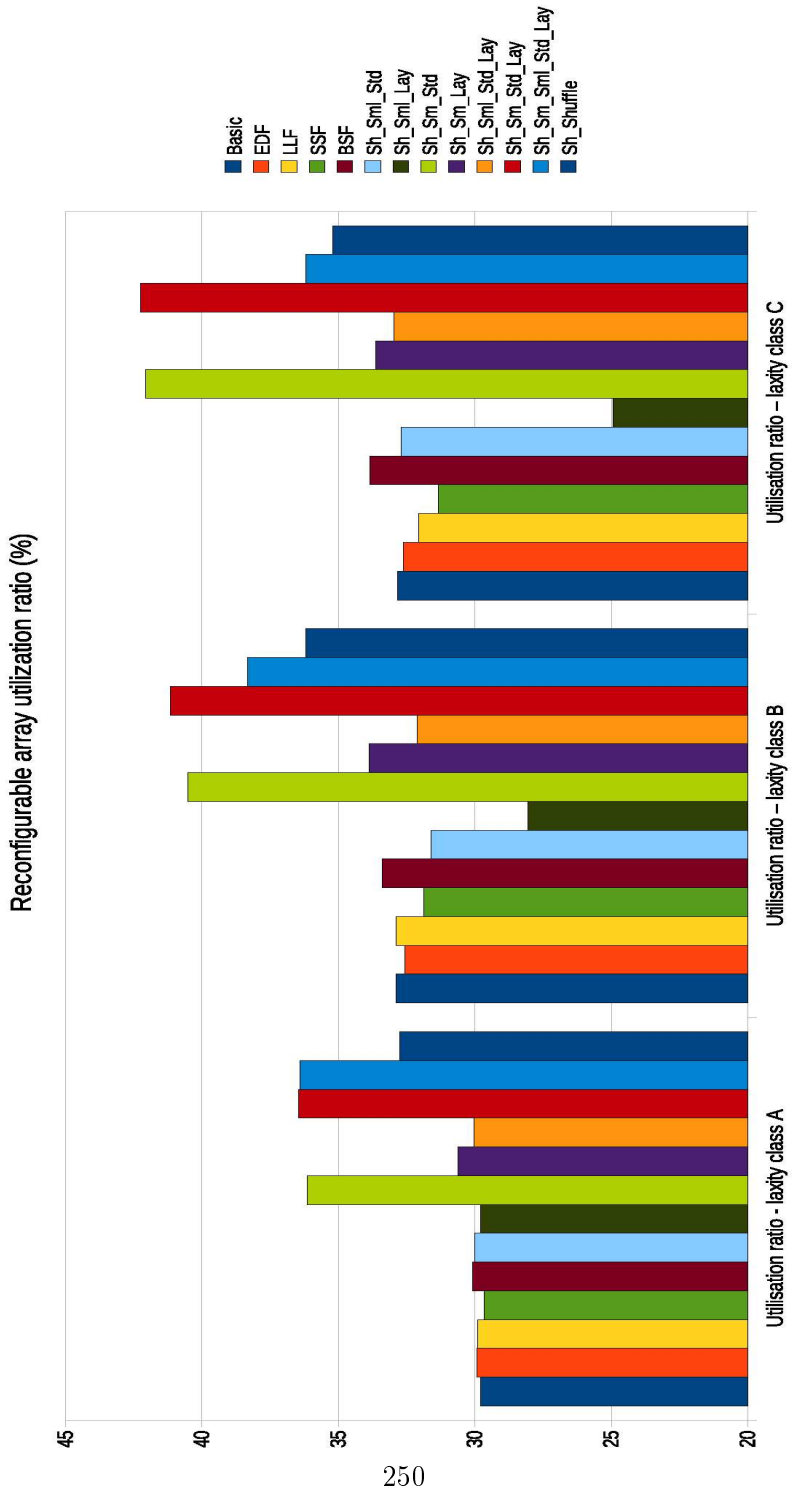


Figure 7.3: The runtime overheads of without-looking-ahead scheduling algorithms (paramaters based scheduling and multi-shape tasks based scheduling)

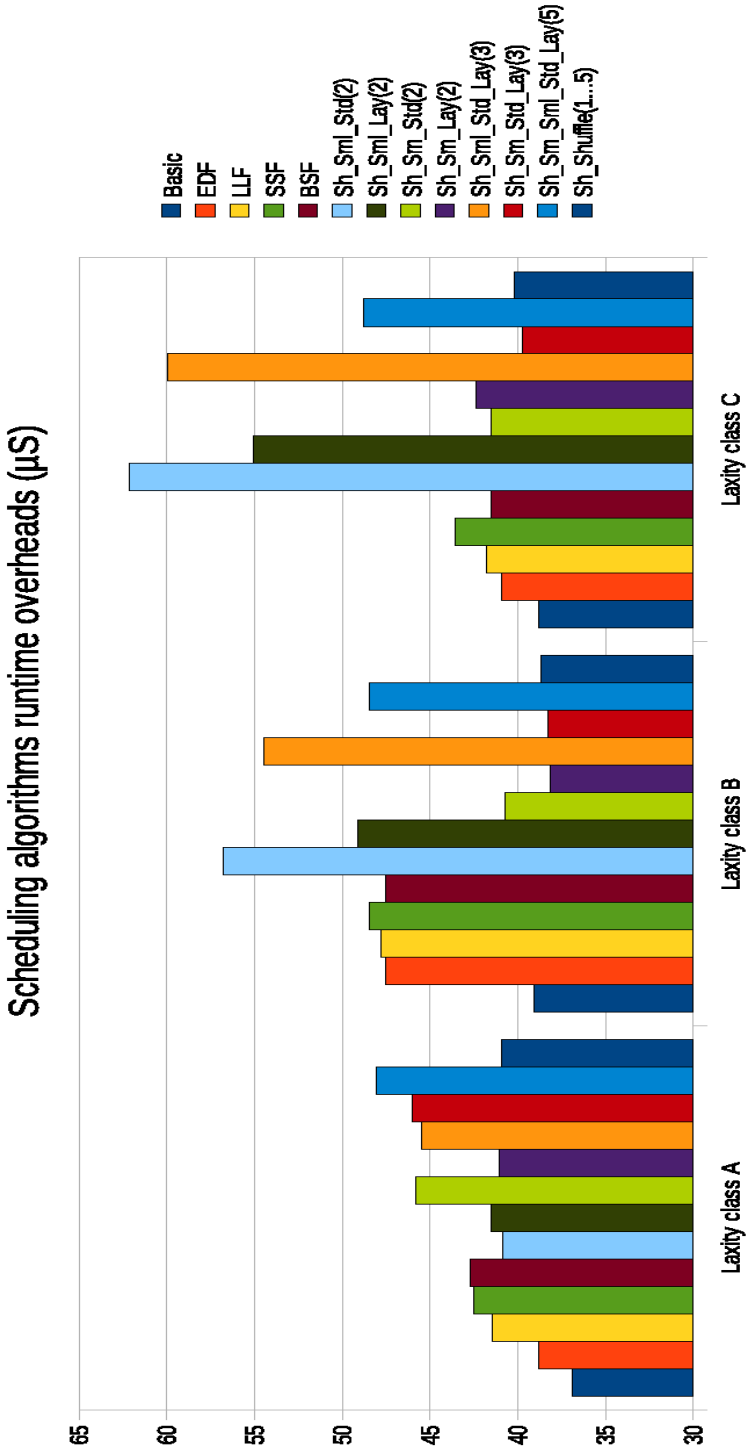


Figure 7.4: The number of scheduler invocations for without-looking-ahead scheduling algorithms (paramaters based scheduling and multi-shape tasks based scheduling)

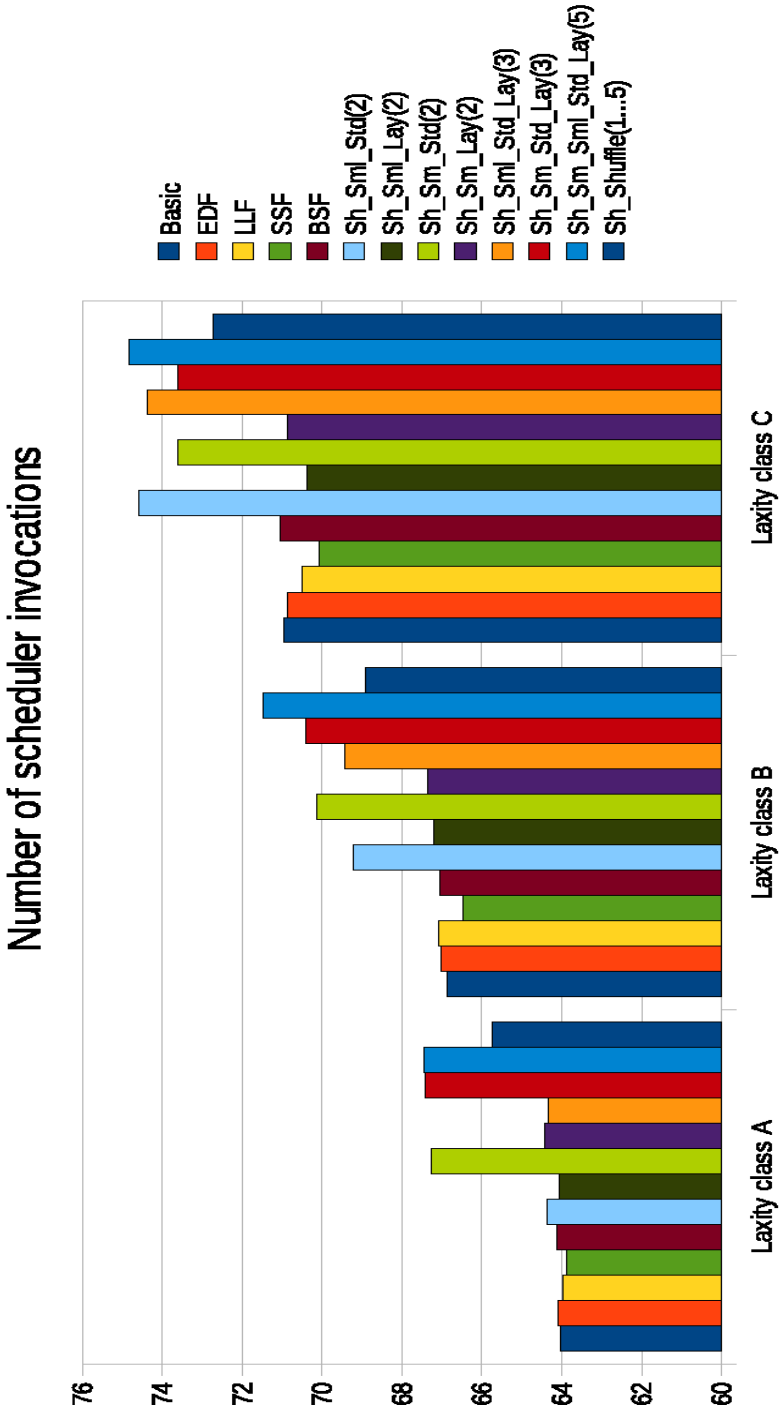


Figure 7.5: The cumulative runtime overheads for without-looking-ahead scheduling algorithms (paramaters based scheduling and multi-shape tasks based scheduling)

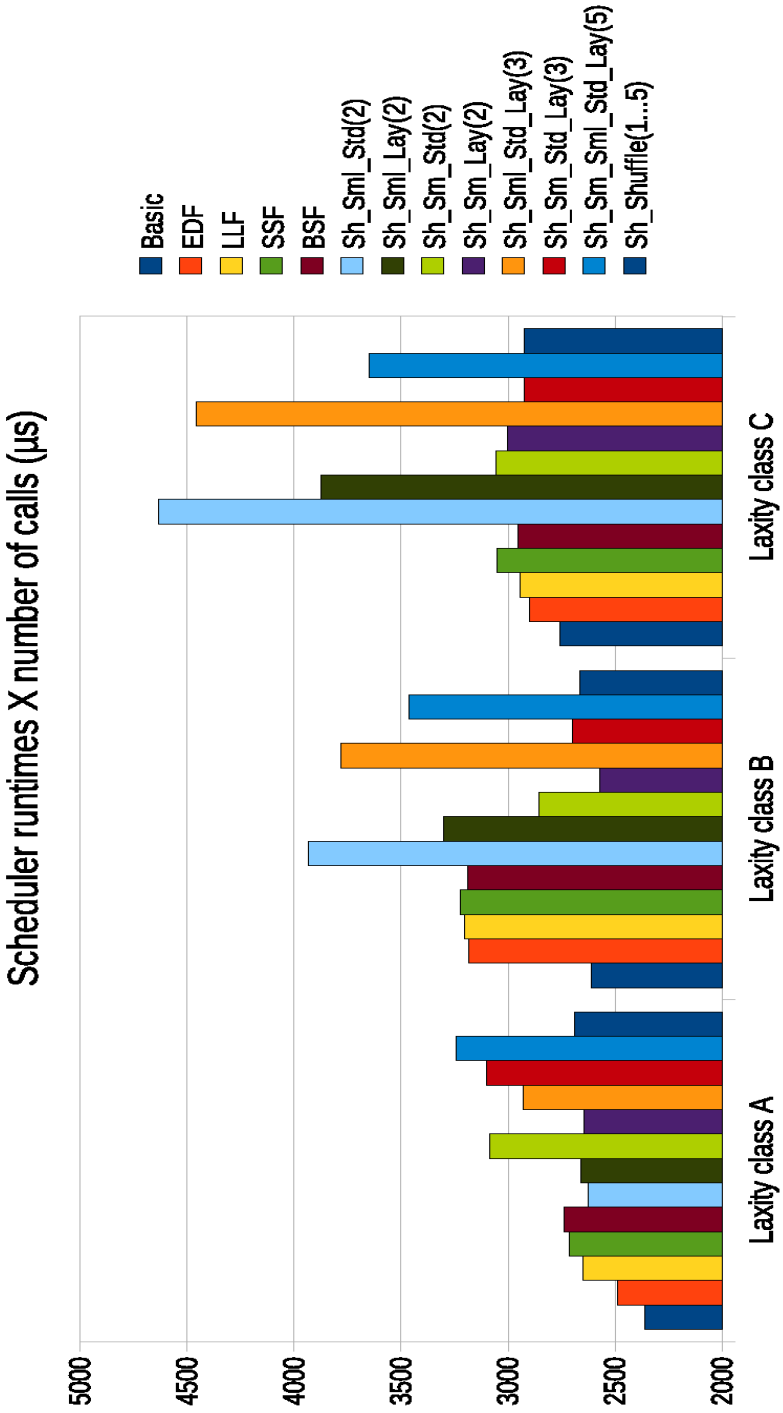
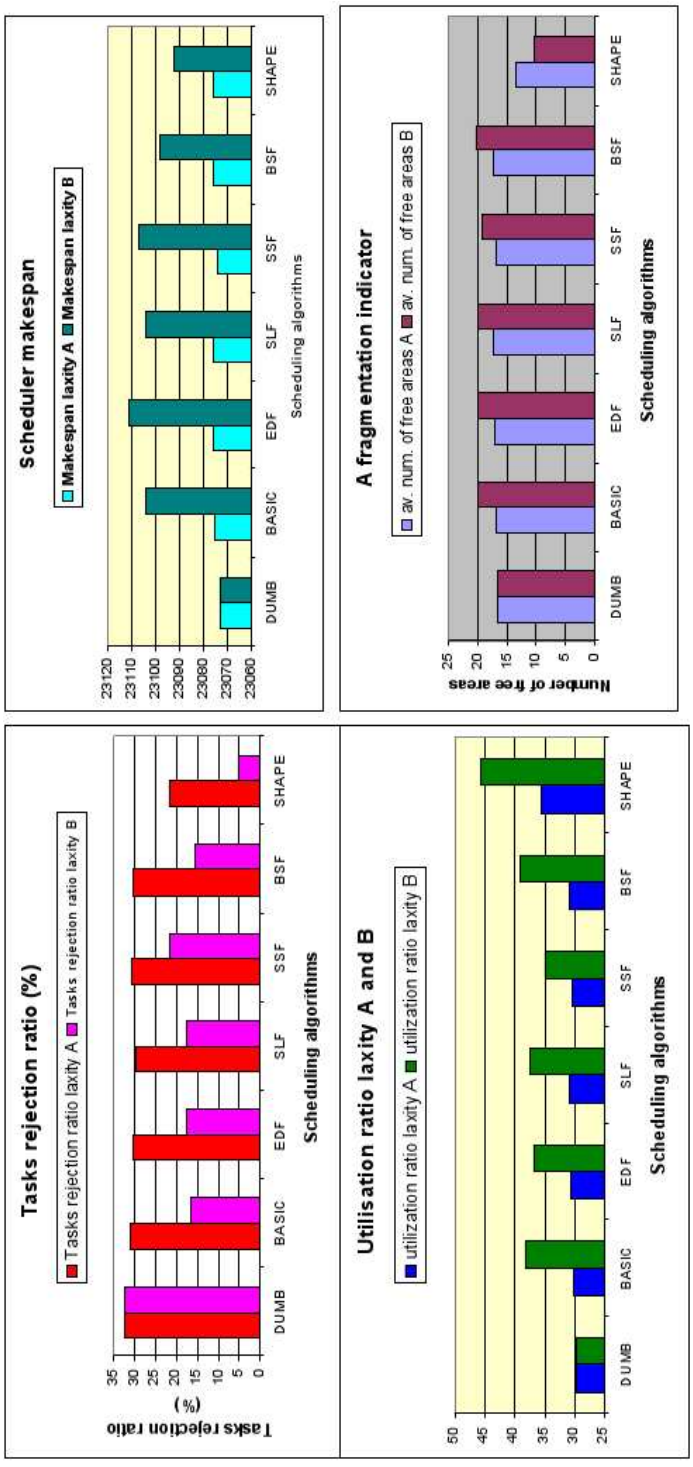


Figure 7.6: Tasks parameters based scheduling algorithms *vs* multi-shape algorithm : Simulation on a large number of tasks (10 sets of 5000 tasks).



Tasks parameters based scheduling algorithms that use a single version per task are compared with a multi-shape algorithm where each task has 3 variants (a normal, a same size standing, and a smaller size). DUMB algorithm accepts a task and places it as it arrives, or directly rejects it. These simulations have been conducted on 10 sets of 5000 tasks, in order to see the behavior of the algorithms over a long time. Multi-shape scheduling algorithm outperforms the others. These results follow the same trend as in the case of 100 sets of 50 tasks.

7.7 Appendix C : Tables of algorithms and data structures implemented

Table 7.3: Scheduling algorithms implemented.

RECAPITULATIVE TABLE OF SCHEDULING AND PLACEMENT HEURISTICS IMPLEMENTED

Scheduling algorithms (those highlighted are from us)

Not-Looking-Ahead scheduling (blind algorithms)				***Looking-Ahead algorithms (prospecting future states of the array)***	
Parameters based algorithms *(Completed & Simulated)*	*Comments (eg. Placer used)*	**Multi-shape algorithms** *(Completed & Simulated)*	*Comments (eg. Placer used)*	**Algorithms** *(partially completed)*	*Comments (eg. Placer used)*
BASIC	*Comparison reference*	BASIC	*comparison reference*	Horizon & Stuffing 1D	*Completed and simulated with areas list scan approach*
EDF (Earliest Deadline First)	*Tree + Hash matrix*	Sh_Sm_Std(2)	*hash matrix + tree*	Horizon & Stuffing 2D	*2D horizon (completed and simulated) Placer 1 : with ternary tree that ease the merging. Placer 2 : with areas list scan approach, with different splitting and merging strategies*
BSF (Biggest Size First)	*using Hash matrix + Tree*	Sh_Sm_Lay(2)	*hash matrix + tree*		
SSF (Smallest Size First)	*using Hash matrix + Tree*	Sh_Sml_Std(2)	....	Simplified Horizon	*to be done...(adding all the areas at the horizon in one time)*
SLF (Smallest Laxity First)	*using Hash matrix + Tree*	Sh_Sml_Lay(2)	....	Simplified Stuffing	*ongoing*
SRF (Smallest Resources First)	Not simulated	Sh_Sm_Std_Lay(3)	....	Improved Horizon 1D-variable slots	*Completed and simulated. The FPGA is dynamically divided in clusters and on each cluster is perfomed a 1D Horizon & Stuffing (1D placement vertical-wise)*
BRF (Biggest resources First)	Not simulated	Sh_Sml_Std_Lay(3)	....	Improved Stuffing	*Completed and not yet simulated. Same principle as above*
		Sh_Sm_Sml_Std_Lay(5)	....	Combining shape based and horizon/stuffing	*Not yet completed ...applying Horizon & Stuffing to tasks with many variants in order to improve the multi-shape approach*
		Sh_Shuffle(1....5)	....		

Multi-shape tasks based scheduling:

0. Sh_Normal(1) : The task does not have any other variant apart from the normal one. The algorithm is then similar to basic scheduling. (i) implies that i is the number of versions per task

1. Sh_Sml_Std(2) : Shape based scheduling using a normal task variant and a smaller standing one with a longer execution time.

2. Sh_Sml_Lay(2) : Shape based scheduling using a normal task variant and a smaller laying one with a longer execution time.

3. Sh_Sm_Std(2) : A normal task variant and a standing task variant, same execution time.

4. Sh_Sm_Lay(2) : A normal task variant and a laying task variant, same execution time.

5. Sh_Sm_Std_Lay(3) : A normal task variant + a standing task variant + a laying task variant, same execution time.

6. Sh_Sml_Std_Lay(3) : A normal task variant + a smaller standing and a smaller laying, the two latter with longer execution time.

7. Sh_Sm_Sml_Std_Lay(5) : A normal task variant + a standing task variant + a laying task variant + a smaller standing + a smaller laying.

8. Sh_Shuffle(1....5) : The variants are randomly uniformly distributed among the 8 above. In this algorithm; the Placer checks a place for a given task in this order if the version exists :

normal_version → smaller_standing → smaller_laying → same_standing → same_laying

Parameters based scheduling:

BASIC : Scheduling algorithm using a running tasks list, and a waiting tasks list which is sorted in the increasing arrival time and placed in that order.

EDF : Earliest Deadline First

BSF / SSF : Biggest Size First / Smallest Size First

SLF /LLF : Smallest Laxity First or Least Laxity First

SRF : Smallest Resources First (Task resources = width x height x execution time)

BRF : Biggest Resources First

Table 7.4: List of placement algorithms implemented.

Placement			
Placement algorithms and structures		**Comments**	**Scheduling algorithm associated with**
Hash_matrix *Walder et al. (2003)*		*Completed and simulated*	*Tasks parameters based scheduling* *Multi-shape tasks based scheduling*
Binary search Tree *Bazargan et al. (2000)*		*Completed and simulated*	*Tasks parameters based scheduling and* *Multi-shape based scheduling*
Staircase (MER) *Handa and Vemuri (2004c)*		*Completed and simulated even on an embedded platform, with accurate measurements*	*Basic scheduling*
SLA (MER) *Cui and Deng (2007)*			
IF (Immediat Fit) *Marconi et al. (2008)*		*Completed*	*Basic scheduling*
Areas list scanning (First Fit, Best Fit, overlapping, nonoverlapping, etc...)		*Completed and simulated*	*1D and 2D Horizon, 1D Stuffing (Steiger et al. ,2004)* *Tasks parameters based scheduling, Multi-shape, ...*
Ternary search tree		*Completed and simulated*	*- 2D Horizon (Steiger et al. ,2004)* *- 2 Variants of Horizon : EAAF and SFAF scheduling*
1D-variable slots		*Completed and simulated*	*1D-variable slots looking ahead scheduling (denoted as improved 1D horizon and stuffing)*

Table 7.5: List of placement structures implemented (1): The areas partitioning (existing works are cited and those from us are highlighted).

PLACEMENT OPERATIONS (1)					Comments	Scheduling used
Areas partitioning	**Areas pre-partitioning approach (slot-based or cluster-based)**	IF (Immediat Fit) algorithm *Marconi et al. (2008)*			pre-partitioned areas that ease the splitting and merging processes (completed)	Basic scheduling
		1D-variable slots (slots are dynamically resized).			*n x 1D-variable slots looking-ahead* scheduling, *n* being the maximum number of slots or clusters (completed and simulated)	Nx1D-variable slots looking-ahead scheduling. Also denoted as Improved 1D Horizon and Stuffing
	Partitioning after tasks placement	Non Overlapping splitting	Vertical split		Completed and simulated. non overlapping; simulations are mainly done with the vertical split which gives better results	Tasks parameters based scheduling, Multi-shape tasks based scheduling, etc...
			Horizontal split			
			SSEG (shortest segment), *Bazargan et al. , 2000)*		Completed, not simulated	
		Overlapping splitting			Completed but not simulated.	
		Smart vertical/Smart horizontal (partitioning depending on the aspect ratio of the resulting rectangles)			Inspired from the binary tree. Smart vertical (completed, not simulated) Smart horizontal (not yet completed)	
	Any partitioning	Scanning approach (Cells level scanning, MER-based)			Maximum Empty Rectangles are detected and updated, optimal placement solution.	Staircase algorithm, *(Handa and Vemuri , 2004c)* SLA algorithm *(Cui and Deng , 2007)* Flow Scan algorithm *(Marconi et al., 2008)*

Table 7.6: List of placement structures implemented (2): Finding fitting areas and merging free areas (existing works are cited and those from us are highlighted).

PLACEMENT OPERATIONS (2)						Comments	Scheduling used
Finding a fitting area for a task	**hash matrix**	Full size hash matrix , *Walder et al. (2003)*			*nonoverlapping*	*Completed and simulated.*	*tasks parameters based and multi-shape tasks scheduling algorithms*
					overlapping	*Not yet completed*	
		down-sized hash matrix			*Reduces the number of scans, at the cost of an extra internal fragmentation (not yet completed)*		
	Areas list scanning	*Scanning a list of areas, sorted according to a criteria (Best Fit, etc...)*			*nonoverlapping*	*Areas list scanning is used in every cases where the hash matrix is not used. Scanning a sorted list is easier than any structure, except the hash matrix (completed and simulated)*	*tasks parameters based, multi-shape , Horizon and Stuffing scheduling algorithms*
					overlapping		*Basic scheduling*
	Cells level scanning (MER)				*MERs*		
					1D areas		
		Staircase (*Handa and Vemuri, 2004c*)			*Completed and simulated*		
		SLA algorithm, (*Cui and Deng, 2007*)			*Completed and simulated*		
		Flow Scan (*Marconi et al., 2008*)			*Completed, not yet simulated*		
Merging free areas	**With a Binary and Ternary tree based**	Intuitive merging using a tree Fragmentation bounded			*non overlapping areas (completed and simulated)*		*tasks parameters based, multi-shape , Horizon and Stuffing*
					overlapping areas (completed and simulated)		*tasks parameters based, multi-shape*
	Merging Non overlapping areas	IM - Intelligent Merging, (*Marconi et al., 2008*)	MON (Merge Only if Needed)		*No merge after a task withdrawal. Merge only if one fails to place a task. Non overlap, completed but not yet simulated, easy to do so.*		
			PM (Partial Merging)		*Merge only the minimum subset needed for the task (completed).*		
			DC (Direct Combined)				
		Merging non overlapping areas in a list			*perfect neighbours (completed)*	*Lead to an ever increasing fragmentation, therefore, need a time to time defragmentation rule (not yet implemented)*	*tasks parameters based, multi-shape*
					Aspect ratio based merging (completed)		
					Merge if worthy (completed)		
	Merging MER	List of MERs update process			*Completed and simulated for SLA and Staircase*		*Basic scheduling*

7.8 Appendix D : Size of IPs from the Xilinx core generator

	Application	**Parameters**	**Number of slices on Virtex2pro**	**Projection on Virtex5**
Bus	CAN bus		569 to 885	247 to 385
	Flexray		3089 to 3500	1350 to 1520
	MOST		2306	1000
	Optic Fiber		5000	2173
	PCI	master/target	500 to 1000	217 to 434
		target	500	217
	USB	2	2000	869
	ethernet 10/100 Mbits		1000 to 2000	434 to 869
Floating Calculation	addition / substraction		470	204
	multiplication		424	184
	division		867	377
	square root		524	228
	Comparison		45	20
DSP	COMPLEX MUL	32 bits	645	280
	CORDIC			
		Arctan	582	253
		cosine 16 bits	622	270
		square root 16 bits	113	49
		square root 32 bits	755	328
	FIR32		136	59
	multiply/accumulate	32 bits	215	93
	FFT		1541	670
		32 points - 16 bits		
		multipliers cablés	1550	673
		32 points - 16 bits	2088	907
		multipliers (LUTs)		
		linear *wrt*		
		number of points	400 to 2500	174 to 1086
	Modulations	cosine, 1 line	17	7
		cosine, 4 lines	81	35

Table 7.7: Few IPs for Virtex2pro FPGAs from the XILINX Core Generator System[1].

[1] Xilinx Core Generator System provides a library of user-customizable IPs (or hardware tasks) for Xilinx FPGAs.

7.9 Appendix E : DA Implementation of a Multi-shape Hardware Task : the FIR Filter

A FIR (Finite Impulse Response) filter is commonly used in Digital Signal Processing applications to implement low-pass, band-pass and high-pass filters and other convolution functions. Traditionally, digital filtering algorithms were most commonly implemented using DSP processors for low rates applications (e.g. audio) and ASICs for higher rates. The dataflow representations of direct structure of an $N - order$ FIR filter has been illustrated in figure 2.4, chapter 2 page 30 and the corresponding equation is given by :

$$Y(n) = \sum_{l=0}^{N-1} H_l \cdot X_l(n) = \sum_{l=0}^{N-1} H_l \cdot X_l \tag{7.1}$$

H_0, H_1,..., H_{N-1} are N constant and time-invariant filter coefficients that are computed beforehand. N is the filter length. At each time n, the output response $Y(n)$ is function of the N lasts inputs samples X_0, $X_1 ... X_{N-1}$ only. Therefore, n may be implicit as shown in the final equation.

The output requires $2N - 1$ arithmetic operations (N multiplications and $N - 1$ additions). Different techniques that range from pure serial implementation to fully parallel may be used for FPGA implementation of the filter.

A *fully parallel implementation* is meant to map all the functional blocks depicted in the dataflow representation of figure 2.4, page 30. As one output sample is delivered at every clock cycle, this implementation provides a higher throughput but at the cost of more logic resources consumption.

In a *serial implementation*, input samples are conveyed serially in the filter and computed one by one. The implementation is denoted as bit-serial. As the same hardware is re-used to compute bits one by one, this approach saves hardware resources but requires many clock cycles to compute one output sample (figure 7.7).

Digit-serial architecture is another alternative that combine bit-serial and fully parallel implementations. In Digit-serial, a $W - bit$ data word is processed in units of $N - bit$ digit in P clock cycles, where $P = W/N$. Digit-serial implementation approach is a good trade-off between a lower throughput bit-serial implementation and a higher hardware resources consumption bit-parallel implementation. A range of trade-off may be found between the throughput of the filter and the amount of configurable resources required.

7.9.1 Distributed Arithmetic as an enabling technique

Distributed Arithmetic (DA) is a computation algorithm that uses memory instead of multipliers to perform sum of products where one of the operand remains constant. Hence, DA suits to implement sums of products similar to the FIR equation 7.1 above which may be any convolution where processing one output sample requires the accumulation of N product terms.

If inputs samples and coefficients are two's complement signed number where a binary number X_l equation is given by :

$$X_l = -X_{l,B-1} \cdot 2^{B-1} + \sum_{b=0}^{B-2} X_{l,b} \cdot 2^b = -X_{(l,B-1)} \cdot 2^B + \sum_{b=0}^{B-1} X_{(l,b)} \cdot 2^b \tag{7.2}$$

where $X_{l,B-1}$ is the sign bit of X_l, $X_{l,0}$ its less significant bit, and $X_{l,b}$, $b \neq B-1$ the bit at position b of digit X_l.

If scaled by a factor $S = \frac{1}{2^{B-1}}$, the resulting two's complement scaled number representation is as followed :

$$X_l = -X_{l,B-1} + \sum_{b=0}^{B-2} X_{l,b} \cdot 2^{b-(B-1)} \tag{7.3}$$

The scaling operation maximizes the signal to noise ratio (SNR). Equation 7.3 can be rewritten in a different way as below by swapping b and $-(b-B+1)$:

$$X_l = -X_{l,B-1} + \sum_{b=1}^{B-1} X_{(l,B-1-b)} \cdot 2^{-b}, \qquad \text{with } |X_l| \leq 1 \tag{7.4}$$

The scaled representation of equation 7.4 applied to the FIR filter equation gives :

$$\begin{aligned} Y = &\ \textstyle\sum_{l=0}^{N-1} H_l \cdot X_l \\ = &\ \textstyle\sum_{l=0}^{N-1} H_l \cdot [-X_{(l,B-1)} + \sum_{b=1}^{B-1} X_{(l,B-1-b)} \cdot 2^{-b}] \\ \Rightarrow Y = &\ -[\textstyle\sum_{l=0}^{N-1} H_l \cdot X_{(l,B-1)}] + \sum_{b=1}^{B-1} \ [\sum_{l=0}^{N-1} H_l \cdot X_{(l,B-1-b)}] \cdot 2^{-b} \end{aligned} \tag{7.5}$$

Equation 7.5 is commonly used to express the scaled output Y of a FIR filter; $|Y| \leq 1$ for scaled inputs and coefficients.

Developing Y in the above equation leads to equation 7.6 below, easier to memorize and to manipulate. The latter equation 7.6 consists of $N \cdot B$ products terms. These products terms may be implemented without multipliers (the so-called *multipliers-less* implementation). Each product term $H_l \cdot X_{(l,b)}$ is a binary AND operation between a single bit $X_{(l,b)}$ of the input sample X_l and

its corresponding constant coefficient H_l.

$$\begin{aligned}
Y = \; & -[\textstyle\sum_{l=0}^{N-1} H_l \cdot X_{(l,B-1)}] + \sum_{b=1}^{B-1} \; [\sum_{l=0}^{N-1} H_l \cdot X_{(l,B-1-b)}] \cdot 2^{-b} \\
= \; & -[\textstyle\sum_{l=0}^{N-1} H_l \cdot X_{(l,B-1)}] \\
& + \textstyle\sum_{b=1}^{B-1} [H_0 \cdot X_{(0,B-1-b)} + H_1 \cdot X_{(1,B-1-b)} + \cdots + H_{(N-1)} \cdot X_{(N-1,B-1-b)}] \quad \cdot 2^{-b} \\
= \; & -[H_0 \cdot X_{(0,B-1)} + H_1 \cdot X_{(1,B-1)} + \cdots + H_{(N-1)} \cdot X_{(N-1,B-1)}] \\
& + [H_0 \cdot X_{(0,B-2)} + H_1 \cdot X_{(1,B-2)} + \cdots + H_{(N-1)} \cdot X_{(N-1,B-2)}] \cdot 2^{-1} \\
& + [H_0 \cdot X_{(0,B-3)} + H_1 \cdot X_{(1,B-3)} + \cdots + H_{(N-1)} \cdot X_{(N-1,B-3)}] \cdot 2^{-2} \\
& \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
& + [H_0 \cdot X_{(0,1)} + H_1 \cdot X_{(1,1)} + \cdots + \cdots + H_{(N-1)} \cdot X_{(N-1,1)}] \cdot 2^{-(B-2)} \\
& + [H_0 \cdot X_{(0,0)} + H_1 \cdot X_{(1,0)} + \cdots + \cdots + H_{(N-1)} \cdot X_{(N-1,0)}] \cdot 2^{-(B-1)}
\end{aligned} \tag{7.6}$$

As coefficients H_0, $H_1 ... H_{(N-1)}$ are known beforehand, all possible results of each partial product may be also known beforehand as follows:

$$H_l \cdot X_{(l,b)} = \begin{cases} H_l & \text{if} \quad X_{(l,b)} = 1 \\ 0 & \text{if} \quad X_{(l,b)} = 0 \end{cases}$$

The possible results may be pre-stored in a look-up table and addressed by different bits of X_l. Hence, the same bits $X_{(0,i)}$, $X_{(1,i)}$, $...X_{(N-1,i)}$ of the N input samples X_0, X_1, $...X_{(N-1)}$ are used to address small LUTs where partial products terms are stored.

In addition, a power of two scale factor makes multiplication and division simpler using shift registers. Indeed, multiplying (resp. dividing) multiplicands by a power of two' number (2^i) is equivalent to shifting right (resp. left) i times a binary point. Furthermore, as the numbers are 2's complement signed, adders may implement subtraction and addition. It is relatively easy to remove this gain factor at the output of the filter just by shifting back the binary decimal point.

7.9.2 Implementing Y using LUT-based DA

In figure 7.7 is shown an $N - order$ FIR filter implementation using Look-Up-Table based serial DA. Input samples are sent as a one-bit-serial stream (via "time skew buffer") in N shift registers. At each clock cycle, the LUT is addressed by the i^{th} bit of each of the N input samples. The corresponding partial product stored in the LUT is applied to the *scaling accumulator*. The scaling accumulator automatically sums partial products and scales the output Y. Let T_c be the time to compute one output sample :

$$T_c = B \cdot t_{sm}$$

where B is the bit width of input samples X_l, and t_{sm} the time required by the scaling accumulator to perform a scaled-summation .

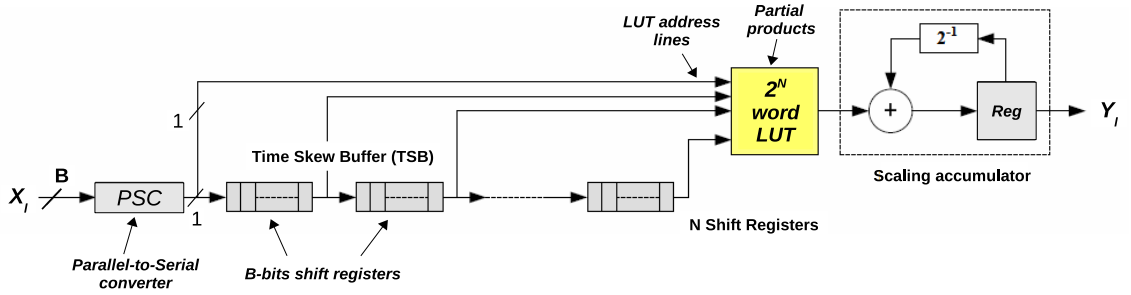


Figure 7.7: Serial Distributed Arithmetic

Thanks to its simple architecture, serial DA may operate at high frequency and thus achieve the same throughput with parallel implementation, but with a greater latency. However, one drawback of this architecture is the rapid growth of the required memory with respect to the filter length. Indeed, a $N-order$ filter requires 2^N words size memory storage capacity[2] to store partial products. For example, a 16 coefficients filter requires a $2^{16} = 65.5\text{x}10^3$ words LUT.

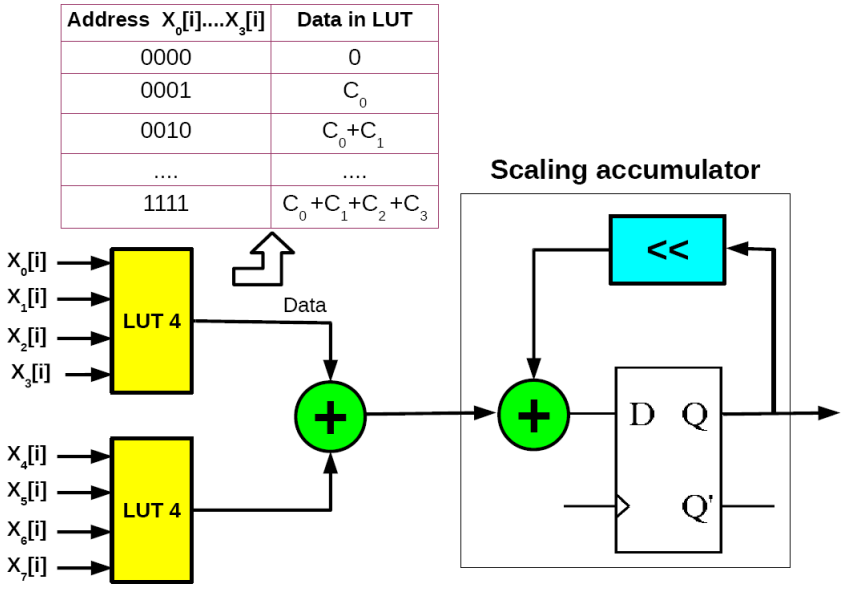


Figure 7.8: Serial-Parallel Distributive Arithmetic

[2] only the half $\frac{1}{2} \cdot 2^N = 2^{N-1}$ is required for a linear phase response FIR filter

Fortunately, as shown in figure 7.8 one may use n LUTs of size $2^{\frac{N}{n}}$ instead of 2^N words size LUT to implement an *N-order* filter. Indeed, as partial products could be computed in parallel, many LUTs may be used in parallel. In the example of figure 7.8, two LUTs of 2^4 words size are used instead of a 2^8 words size to implement a $8 - order$ FIR filter. This way, one saves $2^8 - 2 \cdot 2^4 = 224$ words size LUT at the cost of one more adder. All these implementation trade-offs lead to variant tasks size with variant execution time as illustrated earlier in figure 5.5, page 192.

7.9.3 Throughput *vs* reconfigurable resources trade-off

Table 7.9 depicts examples of trade-off between reconfigurable resources utilization and throughput in DA-based implementation of a FIR filter. On one hand, the fully parallel implementation provides the highest throughput at the cost of more resources utilization (3072 slices). One output sample is delivered every clock cycle. On the other hand, various combinations of multi-bit serial DA implementation require more than one clock cycle to compute and deliver each output sample, but utilizes less resource. As multi-bit serial technique uses several serial units, it provides a trade-off between high resources utilization of fully parallel and low throughput of fully serial. The table illustrates examples where the many the number of clocks cycle required compute one output sample, the less the slices used.

The throughput of the filter, therefore its processing time, depends on the amount of configurable resources used (Xilinx, 2005)

Number of Clock Cycles per Output Sample	**Slice Count**	**Filter Sample Rate (MHz)**
1	3072	150
2	1571	75
3	994	50
4	802	37.5

Figure 7.9: Example of resources utilization for different DA implementation of a FIR filter