Diploma Thesis: Hardware Task Scheduling Targeting a Realistic FPGA device

by
George Charitopoulos

Department of Electronic and Computer Engineering
Technical University of Crete

Chania, February 2013

Advisor Professor: Prof. Dionisios Pnevmatikatos

Evaluation Committee:
Professor Dionisios Pnevmatikatos
Professor Apostolos Dollas
Associate Professor Ioannis Papaefstathiou

Abstract

The last few years FPGAs have penetrated the mainstream and have experienced wide usage through the users. Also the concept of reconfigurable computing has benefited numerous application domains, with FPGAs being the stronger representative of that. One of the most crucial technologies incorporated in some specific FPGA families is called partial reconfiguration. It allows for the reprogramming of part(s) of the FPGA chip without disturbing the rest of its operation, even during runtime.

FPGAs have been widely adopted in embedded systems. Partial reconfiguration technology can leverage these systems by swapping in and out task modules in an operating-system fashion. A task can be downloaded upon arrival or when needed, during the system operation. To this direction one of the most important parts of said embedded system is the Scheduling Algorithm.

The Scheduling Algorithm is responsible for the placement and scheduling of hardware tasks on the device when those are needed. Thus far many scheduling algorithms have been proposed by the research community. However these algorithms have the drawback that they are not compatible with the target devices, due to the neglecting of several technology restrictions.

In this thesis we present a novel scheduling algorithm that manages the arrival of hardware tasks and places them on the FPGA. This algorithm could be incorporated at any complete runtime system inhabited on a FPGA.

Dedication

I would like to dedicate my thesis in my passed away grandmother, Harikleia, with the hope that this work will finally convince her that I, in fact, was accepted and finished my education at the Technical University of Crete.

Acknowledgments

I would like to thank, first and foremost, my family, who I deeply love, for their love and support, during all these years.

Also I would like to thank Dr. Kyprianos Papadimitriou, for without his help and guidance this work would never see the light of day, also for always being there to talk about the difficulties of this work and making said talks easy, pleasant and educational, while giving them a friendly vibe, which was very kind and helpful.

Also I would like to thank Professor Dionysios Pnevmatikatos for giving the opportunity to work on this very interesting field of research and for his support throughout the elaboration of this thesis.

Additionally I would like to thank Professor Apostolos Dollas and Associate Professor Ioannis Papaefstathiou, for taking the interest in my work and agreeing to evaluate it.

Last but not least I would like to thank the Theatrical Team of the university, without which my studies here would have been longer and more difficult and also my friends in Crete and in Athens for coping with me all these years.

Contents

Chapter 1

Introduction

In this chapter, an introduction is made about FPGAs, partial reconfiguration, scheduling and the problems regarding an embedded operating system. Also, a presentation is made regarding the subjects of this thesis.

1.1 Field Programmable Gate Arrays

The Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to by configured by the end-user, in order to execute different applications. The final FPGA configuration is specified, by using a hardware description language (HDL), similar to that used for an application specific integrated circuit (ASIC). FPGAs can be programmed to implement any logical function that an ASIC could perform. Over the years FPGAs have increased their popularity amongst users due to the ability to update their functionality after shipping, partial re-configuration of a portion of the design and the low non-recurring engineering costs relative to an ASIC design.

FPGAs contain programmable logic components called “configuration logic blocks” (CLB) and a reconfigurable network used for interconnection between those blocks. Logic blocks can be configured in order to perform complex combinatorial functions or simple logic gates like AND and XOR. Also this blocks can include memory elements like flip-flops or more complete blocks of memory and many other more specific pieces of hardware logic (dedicated multipliers, Block RAMs etc). A representation of the internal structure of an FPGA, is shown in Figure 1.1.

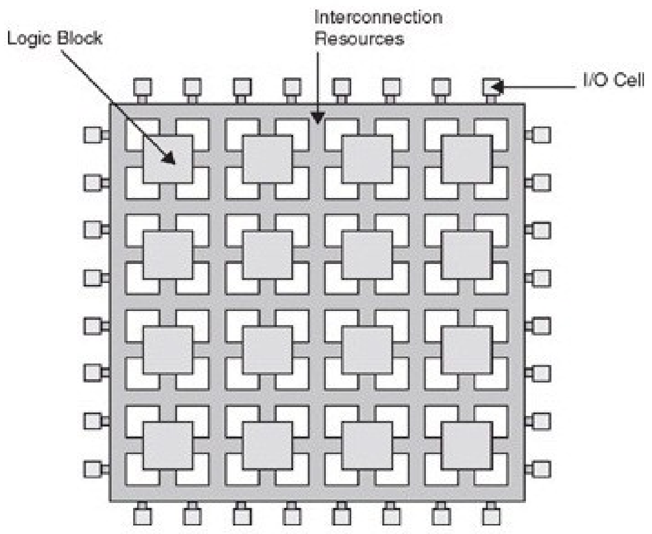


Figure 1.1: A graphic representation of the internal structure of a FPGA.

In the recent years, FPGAs have become a powerful tool, due to their many advantages and are broadly used in several applications such as, digital signal processing, software-defined radio, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bio-informatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

1.2 Partial Reconfiguration

Partial Reconfiguration (PR) is the process of changing the portion of a reconfigurable hardware circuitry. This can be done, either when the rest of the design is operational and the device is active (dynamic partial reconfiguration) or when the device is inactive (static partial reconfiguration). PR gives the ability to share the same hardware amongst different applications, increase resource utilization, update the hardware remotely and to adapt hardware algorithms. Some of the benefits of PR are:

- **Increased System Performance:** PR provides the ability, to the user, to use more efficient implementations of a design for different situations, while the rest of the system continues its execution without performance loss or disputed results during the reconfiguration process.
- **Reduced Power Consumption:** In power-constrained designs the user can simply download a more power-efficient version of a module, or a blank bitstream when the particular region of the device is not needed, therefore reducing the power

consumption of the design.

- **Adaptability:** Designs with the use of PR can adapt to changes in their environment, their input data or even their functionality. This capability makes the design more efficient, compared to a more generic one.
- **Hardware Upgrade and Self Test:** The ability to change hardware. Xilinx FPGAs can be updated at any time, locally or remotely. Partial reconfiguration allows the end user to easily support, service and update hardware in the field. Also with the use of self test components, the user can check, on demand, the integrity of the system. [1]
- **Shorter reconfiguration times:** The configuration time is directly proportional to the size of the configuration bitstream. Partial reconfiguration allows you to make small modifications without having to reconfigure the entire device. By changing only portions of the bitstream – as opposed to reconfiguring the entire device – the total reconfiguration time is shorter. [1]

The first device able to support partial reconfiguration of their circuitry was released by Xilinx in the mid 90s and it was the model XC6000. Since Xilinx has released many devices with the capability of partial reconfiguration ranging, from the high-end Virtex-6 to the low-end Spartan 3/E family. Xilinx supports two basic styles of partial reconfiguration, module-based and difference-based.

1.2.1 Module-Based Partial Reconfiguration

Module-based partial reconfiguration uses modular design concepts to reconfigure large block of logic. These modules are known as partially reconfigurable modules (PRM) and each of them implements a certain function or algorithm. These modules are loaded in certain regions, defined statically, called partial reconfigurable regions (PRR). In order to perform the loading of a PRM to a certain PRR the device needs information, which is encrypted in a file called partial bitstream. A partial bitstream is created at design time and is used to reconfigure a specified region with a PRM. It is important to note that a partial bitstream is in direct accordance, first to the PRM for which it was created and to the PRR, this PRM will be loaded on.

The communication access to the PRR is achieved through special pre-defined static buses called Bus Macros. These bus macros are the "gateways" through which all the System – PRM communication processes will occur, with the exception of clock signals. Bus macros can be placed, during compile time, on the boundaries of PRRs in order to define pins where PRMs can hook themselves. These macros are made with pairs of CLBs; one side of the CLB pair is connected to a RR signal, while the other is connected to a static logic signal.

1.2.2 Difference-Based Partial Reconfiguration

Difference-based partial reconfiguration can be used when a small change is made to the design, for example changing LUT equations or BRAM contents. In order to use this method the user needs to make modifications of the device layout and routing with the use of low-level software. The resulting partial bitstream contains only information about the differences, between the current design structure, that resides in the FPGA, and the new content of the FPGA. Despite the fact, that this method allows really fast reconfiguration it introduces two limitations:

- Difficulty to change the design's routing.
- It cannot be used with a wide range of applications, due to the fact that it applies only in simple designs.

1.2.3Applications of Partial Reconfiguration

As stated before partial reconfiguration is a very important ability the FPGAs offer and with the help of, have become more popular amongst users. Currently, many application domains are being studied, in order to explore the benefits gained from their implementation with partial reconfiguration technology. Currently PR is the cornerstone for power-efficient and cost-effective software-defined radios (SDRs) [2].

Another usage of partial reconfiguration is in the migration and recovery of single-event upsets (SEUs). More specifically applications that are space-based have a high probability of experiencing SEUs. With the use of partial reconfiguration, a system can detect and repair SEUs in the configuration memory of the device without disruption of its operations or completely reconfiguration of the FPGA.

Implementation of cryptographic systems, can also benefit from the use of PR. PR in these systems gives the ability to support a range of cryptographic algorithms without the need of having them all integrated in the design. The system can swap algorithms in and out on demand with the use of PR. This not only makes these designs more simple but allows for more inexpensive FPGAs to be used, thus reducing the overall cost.

1.3 The Embedded Operating System Hypothesis and the Scheduling Issue.

Since the first time Xilinx released FPGAs with the ability of dynamic partial reconfiguration, many research groups contemplated the construction of an operating system accommodated in a FPGA. More specifically, the goal of the research community, was to create an OS that would manage the hardware implementations of numerous tasks and their execution on hardware. The advantages of such an OS were many, first the hardware acceleration produced by the device would make certain processes run much faster that they would in software, also the portability of the device was a great asset, plus the fact that even without an embedded OS the FPGAs were experiencing a vast use amongst experienced and simple users. The first field the researchers worked on towards their goal of an embedded Operating System was the Scheduling Algorithms.

The term scheduling was first introduced to describe algorithms used for operating systems. The main job for a scheduling algorithm is to give access to threads, processes or data flows, to the system resources, mainly the CPU time or the memory. In real-time systems the scheduler is important to ensure that the processes can meet their deadlines. A good scheduler is characterized by its ability to keep fairness between the distribution of system resources to the several processes, a low declined processes rate, when our tasks have deadlines, good utilization of the device's resources and a low response time.

Thus, from the beginning a great deal of work was spent on the design of said Scheduling Algorithms for the OS. Instantly the researchers saw that the traditional Scheduling Algorithms derived from the software OS area could not work. The main problem was, that in hardware, we are concerned about the space used by a task in the device and as a conclusion, in order to plan tasks in the future; we need to implement a free space manager or placement manager, as it is usually referred to, inside the scheduling algorithm or in co-operation with the scheduling algorithm.

In those first attempts researchers tended to work towards the 2D area model. Despite the fact that the 2D area model was not yet available in the Xilinx devices, the researchers used it because it offered the ability to better manage the free space and produced less area fragmentation. The placement manager was responsible for finding, at each point in time, the maximum rectangles in the device. This problem can be compared to the 2D bin packing problem, which is known to be a combinatorial NP-hard problem. For this problem several solutions have been proposed, most of them consider different ways to partition the device, in order to create maximum rectangles. Many of those works evaluate their solutions in a discrete time framework constructed in C language. At that time, the researchers choose that way to implement their designs because it seemed logical, the same program could run in the Microblaze processor inhabited in the device with minimum changes to the code, that has been already created.

However after many attempts to a complete Embedded OS system inhabited in a Partially Reconfigurable FPGA, the researches tend to neglect other restrictions induced by the target device. First there is the PRRs restriction, which states that the device has to be pre-partitioned in PRRs, that clearly state the position a PRM while be accommodated on. Also it is important to state that a runtime change in those PRRs is not feasible.

Second the FPGA has certain restrictions regarding the binding between a PRR and a PRM, that binding is done through the bitstream as it is necessary, in order for a PRM to be placed on a PRR, that a bitstream would be created that binds these two. Finally the researchers have made many wrong assumptions regarding the use of module relocation on modern FPGAs.

All the above, prevent the algorithms created thus far and that will be presented in Chapter 2, to be implemented and work on a realistic FPGA device, thus rendering them useful, only for theoretical work and research, or applicable on the future, when a different partial reconfiguration technology will be available.

1.4 Thesis Contribution.

In this thesis we consider the module-based dynamic partial reconfiguration of a FPGA. This thesis studies the scheduling algorithms developed through the years for Reconfigurable Hardware Operating Systems and presents a novel scheduling algorithm we constructed.

With this work we attempt to create a scheduling algorithm that not only successfully schedules hardware tasks on a FPGA device, but also takes in consideration all the known restrictions induced by the device. That way this thesis offers one of the few complete and implementable scheduling algorithms for hardware tasks done so far.

More specifically the thesis's subjects are the following:

- First, we study previous scheduling algorithms along with other related work on the field of an Operating System inhabited in a FPGA.
- Second, we present the results of our implementation off three scheduling

algorithms we chose to implement. The implementation of other scheduling algorithms was important in order to take ideas and gain useful knowledge on how to create efficiently a scheduling algorithm.

- Then, we discuss why any of the implemented or presented algorithms are not implementable in a realistic Partially Reconfigurable FPGA. Here we present also, a thorough analysis of each algorithm and which particular FPGA restrictions it violates.
- Moreover, we present a scheduling algorithm, we created, targeting a realistic FPGA with partial reconfiguration capability. The scheduling algorithm we created is implementable in almost any device, as it obeys to every known technology restriction, contrary to most of the current state of art.
- Finally, we implement and evaluate our algorithm on a simulating framework. The data we feed our framework with simulate one of the most well known technologically realistic FPGAs.

1.5 Thesis Structure.

In Chapter 2 we will provide references and analysis of the related work used in this thesis, mainly the preexisting scheduling algorithms we studied and research work regarding the creation of an Operating System inhabited on an FPGA. In Chapter 3 we present, the implementations we made of three preexisting scheduling algorithms, along with a study of, why any of the algorithms presented or implemented, are not applicable in a realistic scenario. Also presented here are, the technology restrictions of the Partial Reconfiguration process. In Chapter 4 we present, our ideas regarding a novel scheduling algorithm obedient to the technology restrictions analyzed previously, which also uses the process of Module Relocation. In Chapter 5 we present evaluations we made on our algorithm not only with task sets created by us, but also with task sets taken from other researchers or similar to them. Finally, Chapter 6 summarizes our work and provides ideas for future modifications of the existing algorithm.

Chapter 2

Related Work

So far, several researches have been published regarding the implementation of scheduling and placement algorithms for hardware tasks. Also there are few projects trying to implement an Operating System based on a FPGA partially reconfigurable device. In this chapter we describe these works, we managed to collect information for and then we continue with the previous work, that has been conducted in the Microprocessor and Hardware Laboratory of the Technical University of Crete.

2.1 Related work regarding OS for partially reconfigurable FPGAs.

One of the first works, regarding the idea, of creating an operating system inhabited in a partially reconfigurable device was [3]. In this study J. Burns et al. depending on three different applications, manage to extract a set of common requirements and design a runtime system for managing the dynamic reconfiguration of FPGAs. The resulting system incorporates operating-system style services, that permit sophisticated and high level operations on circuits. Even though their work shows an understanding of the technology restrictions, some aspects of the above-mentioned high level operations on circuits are not yet applicable in current devices. However, despite the fact that, one might consider their job chronologically old regarding the knowledge the research community had for the technology restrictions induced by the FPGAs, they correctly consider that circuits can be downloaded at pre-defined, at compile time, areas on the

FPGA. Also the term “library of circuits”, that is often mentioned in their work, is an important fact of the scheduling process. It is important in order to have a quality scheduler to have many choices in different implementations of the same circuit.

A more complete study regarding the advantages, that can be derived from the use of Partial Runtime Reconfiguration (PRTR) in high-performance reconfigurable computing is done in [4]. There, El-Araby et al. first analyzed the execution model of PRTR, exploring what are the key aspects that affect its performance and then experimentally verified his findings. This work showed that PRTR can become the trend for improving the performance, in high-performance reconfigurable computing, as the experiments showed that PRTR can be almost twice faster than the Full Runtime Reconfiguration (FRTR) alternative. Although the researchers had to assume that practical considerations induced by the technology, mainly the overhead induced by the slow ICAP, might overweight the gains. However with the potential future use of an *Operating System* it is clear that PRTR is far more beneficial than FRTR for versatility purposes, multitasking applications and hardware virtualization.

Shortly after, P. Lysaght, B. Blodget and J. Mason presented a partial reconfiguration design flow that helps the end-user efficiently create dynamic reconfigurable designs in [5], along with a description of some enhancements done to Xilinx FPGAs in order to provide better support for the creation of dynamically reconfigurable designs. Their work offers important insight, on the way a system should be built, in order to be implementable in a realistic partially reconfigurable device. Moreover the authors here present terms like PRR, PRR-PRM binding via a bitstream and device partitioning at compile time, that are crucial on understanding correctly the PR process.

Also, a work regarding the implementation of an operating system can be found in [6]. In their work, H. Walder and M. Platzner, present a runtime environment that partially reconfigures and executes hardware tasks on a Xilinx Virtex device. Their implementation splits the reconfigurable surface of the device into vertical task slots that can accommodate hardware tasks. The static region that includes all operating system modules is organized into two OS frames. Also a bus-based communication infrastructure is created that allows tasks communication and I/O. A system graphical representation is shown in Figure 1. Their works has been tested in a Virtex-II device allowing for task partial reconfiguration and execution.

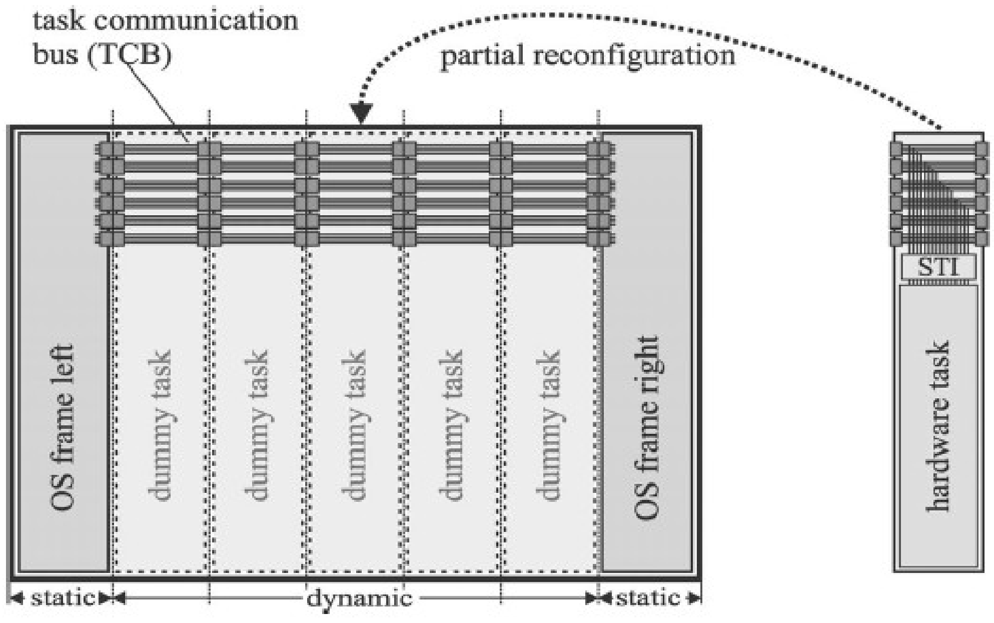


Figure 2.1 : Here we see the system model presented by Steiger. In this Steiger considers pre-defined PRRs and a Task Communication Bus, based on hardware macros.

A work that has made all this possible can be found in [7], where B. Blodget et al. offered many insights towards the improvement of the already existing external reconfiguration control interface called internal configuration access port (ICAP). Their Self Reconfiguring Platform (SRP) defines two APIs. The lower level one is the ICAP API, which provides access to the configuration cache and controls reading and writing the cache to the device. The higher level API is the Xilinx Partial Reconfiguration Toolkit (XPART), which abstracts the bitstreams details providing access to select FPGA resources.

Although all these years many works have tried to create a complete runtime system for reconfigurable devices, the problem is that it has yet to penetrate the mainstream. The reason why this happens is examined in [8], where K. Compton and W. Fu also offer some solutions to this issue. According to their work, one of the biggest problem is the management of reconfigurable hardware in a multi-threaded environment. Also in their work they propose simple schedulers like, a *Most Frequently Used* scheduler, a *Best Fit* scheduler, both of which are based on simplistic greedy methods and a more complex *Multi-Constraint Knapsack* scheduler. These schedulers not only choose which kernels, i.e. hardware tasks, should be implemented in hardware for each scheduling interval, but also the specific hardware implementation for those kernels.

One of the most thorough works on the field of creating a complete Runtime Manager has been shown in [9]. In this paper a Run Time Manager (RTM) is introduced able to map multiple applications on the underlying hardware and execute them concurrently. The target architecture the researchers use for designing their system is shown in Figure 2. Moreover it is illustrated how to generate this RTM and how its modular implementation, along with the use of partial reconfiguration, allows the user to explore different policies regarding reconfiguration, task scheduling and resource assignment. Here the researchers take in consideration almost every restriction induced by the device and create a complete Runtime Manager that schedules task on the device. However one of the setbacks this work has is that the researchers do not consider deadlines for their tasks, which allows them to eventually place all the task on the FPGA.

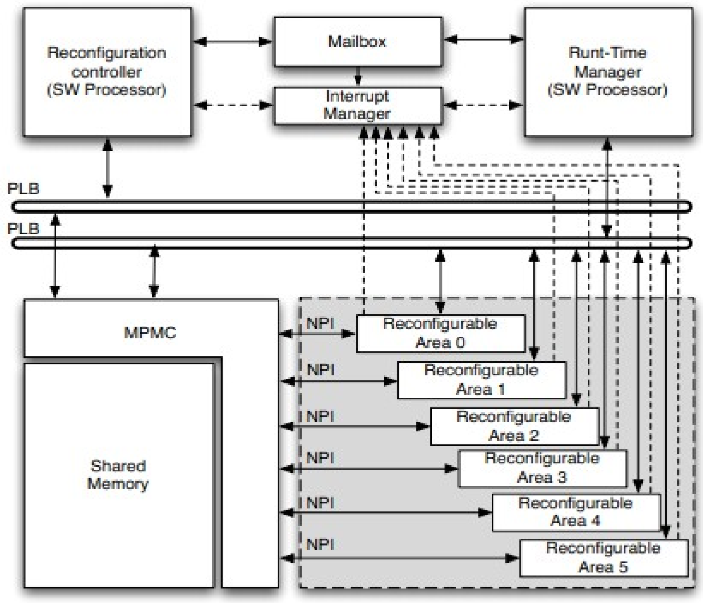


Figure 2.2: The target architecture of the device the researchers used on [9].

2.2 Related work on Scheduling and Placement Algorithms.

Many researchers after proposing their system have tried to create unique schedulers coupling them with that design. One of the first works in the field regarding a hardware task scheduling algorithm was done by C. Steiger et al. at [10]. Their work offered the first two online scheduling algorithms designed for operating systems for reconfigurable embedded platforms, which, since then, have influenced many researchers. In this paper, design issues for reconfigurable hardware operating systems are first discussed similar to the ones presented in [6]. Then for the 1D and 2D resource models the scheduling problem is formulated and two heuristics, the *horizon technique* and the *stuffing technique* are presented. For evaluation, a discrete-time simulation framework has been devised. Further analysis of Steiger's *horizon* and *stuffing techniques*, as well as, our C-language implementation of these algorithms can be found in Chapter 3 of the current thesis.

A great deal of work in creating a scheduler for managing hardware tasks has been spent from T. Marconi et al. One of the first schedulers presented by this research group is in [11]. There T. Marconi and Yi Lu, inspired by Steiger's work in [10] and Y.-H. Chen and P.-A. Hsiung work in [12], create a scheduling and placement algorithm for partially reconfigurable devices. The main idea used by the algorithm is to place each task on the opposing side of the device in order to have more space in the middle and thus better space utilization. The proposed algorithm outperforms the existing algorithms in terms of reduced total wasted area up to 89.7%, has 1.5 % shorter schedule time and 31.3% faster response time. The simulation experiments where done by implementing several algorithms in ANSI-C and run them on a Pentium-IV 3.4 GHz PC using the same artificial task sets.

Furthermore, in [13] T. Marconi et al. present a novel 3D total contiguous surface heuristic, for equipping a scheduler with the "blocking-awareness" capability. The proposed algorithm tends to allocate tasks at positions where blocking of future tasks will be avoided, in order to achieve that, the algorithm calculates the horizontal and vertical contiguous surface between the new incoming task, previously or next scheduled tasks and the FPGA boundary. The main idea of the algorithm is to compactly pack the tasks on the device. The resulting algorithm is evaluated in a discrete-time simulation framework in C.

Moreover, in [30] T. Marconi et al. present the scheduling algorithm that encompasses the 3D total contiguous surface heuristic. However this algorithm is also applicable in 3D Partial Reconfigurable FPGAs, mainly the Virtex-6 family. Nevertheless we can see that the logic of the algorithm is easily applicable to 2D Partial Reconfigurable FPGAs too. The algorithm presented first tries to pack compactly on the device the tasks on space and then in time, thus maximizing the acceptance rates. The resulting algorithm was evaluated in a discrete-time simulation framework in C.

In addition Yi Lu and T. Marconi have created, in [14], the first scheduling algorithm that takes into account the data dependency and the data communication, amongst hardware tasks and between hardware tasks and external devices. The algorithm has three steps, first a suitable placement for the task is found regarding its size, then the configuration port is checked for any conflicts and finally the task is scheduled in a free time and space slot, in accordance with, its communication requirements. The resulting algorithm is evaluated in a discrete-time simulation framework in C.

In order to create an efficient scheduling algorithm, it is very important to create an equally efficient placement algorithm. The speed and the quality placement of hardware tasks on the device, in addition to the proper free space management, are very important attributes, not only for a good online placement algorithm, but also for a good scheduler. Actually those issues have been the first thing researchers examined when the partial reconfiguration feature became available. In order to be more accurate, almost every proposed scheduling algorithm comes along with a placement one.

A very important work on this field is presented in [15]. There K. Bazargan et al. offered the first methods and heuristics for fast and quality online and offline placement of templates on reconfigurable computing systems. The methods introduced here, were the main inspiration for many placement algorithms created in the future. Also the work done by Bazargan here, offers the first proof that, the creation of many representations of the same PRM (library of bitstreams) can improve up to 10% the system's acceptance rate.

Also in the field of task placement a highly respectable work was presented by K. Compton and et al. in [29]. There K. Compton inspired by the concept of task relocation proposes several techniques, that optimize the already existing process. Those techniques include several task transformations in order to achieve better defragmentation via task relocation. However, the transformations proposed here were not applicable at the time so K. Compton proposes a new Relocation Architecture, that implements and fully exploits the benefits of her proposed transformations.

H. Walder, C. Steiger and M. Plantzer in [16] try to create methods that outperform the ones introduced by K. Bazargan and are focused on finding efficient ways to partition the reconfigurable resources space, as well as, creating a hash matrix data structure to maintain the free space. More specifically the researchers try to avoid the direct partitioning of the device after the insertion of a PRM and maintain a series of overlapping rectangles. According to simulations, which were made in a discrete time simulation framework, their placement methods offer an improved placement quality against previous art in the field.

In [17] H. Walder and M. Platzner focus on a major aspect of a reconfigurable operating-system; task placement and transformation. First they discuss the task characteristics and system models, and then they investigate task placement techniques for non-rectangular, coarse-grained tasks and propose footprint transforms; that change task shapes, in order to find possible mappings. However, many of the proposed footprint transformations are not realistic, due to the complexity of the re-routing process.

In [18] T. Marconi and Yi Lu propose three techniques regarding space management: Merging Only if Needed (MON), Partial Merging (PM), and Direct Combine (DC). These technique focus on the merging of non-overlapping empty rectangles, created by the placement of a task on the device and the partitioning done by its placement. The algorithm proposed uses the above techniques dynamically to exploit their advantages. For their simulations they constructed in a discrete-time simulation framework in ANSI-C, to evaluate the performance of the proposed techniques and compare it to related art.

Also, in [19] T. Marconi et al. present a new strategy for online placement algorithms on 2D partially reconfigurable devices, termed the Quad-Corner. The main difference between QC and other work in the field are the abilities of quad-corner spreading of the tasks in the reconfigurable surface, i.e. the algorithm spreads the hardware tasks close to the four corners of the device thus maximizing the free area in the middle, an idea also used in [11]. The resulting algorithm is evaluated in a discrete-time simulation framework in C.

In [20] T. Marconi et al. present yet another task placement algorithm. Their goal is first to exploit the fast search capabilities, offered by the pre-partitioned model. For that purpose in their work the FPGA is pre-partitioned into three different size logic blocks, small, medium and large. Also in order to manage the FPGA resources more efficiently the algorithm performs split, merge and recover operations. Even though the first premise of this algorithm is obedient to the FPGA restrictions (pre-partitioning model), the operations done during runtime by the algorithm are in violation of other FPGA restrictions. The algorithm is programmed using C language, and executed under Linux 2.6 with Intel(R) Pentium(R) 4 CPU 3.00GHz.

Finally in [21] A. Montone et al. try a different approach in hardware task placement and space management. Their work focuses on a resource- and configuration-aware floorplacement framework, using an objective function, based on external wirelength. Their simulations showed a great reduction in wirelength and a huge reuse probability of existing links, i.e. pre-placed, at design time, bus macros, but has introduced significant area fragmentation.

2.2 Related work done in the Technical University of Crete's Microprocessor & Hardware Laboratory.

In the works presented above, several aspects of reconfiguration time, throughput and the overall performance of partial reconfiguration in FPGA systems, are vague and often wrong assumptions are made by the researchers. Also, there is no clear clarification of the metrics used to evaluate the scheduling and/or the placement algorithms presented. For example, the assumptions made by T. Marconi et al. at [18] regarding the reconfiguration time of a task are very simplistic. Also the term "quality of placement" is yet undefined. T. Marconi et al. in all of their work consider the placement quality a measure defined by the total wasted area in a reconfigurable device during scheduling, whereas H. Walder et al. in [16], focus more on the *total execution time* of the algorithm and the *average waiting time* of the scheduled tasks. Additionally it is proven to be really difficult for researchers to determine the *penalty factor* induced to a system from executing a task's software implementation instead of its hardware one. For example in [15] K. Bazargan simply multiplies a declined task's dimension with the time that it would be accommodated on the device.

However in our lab a substantial amount of work has been made in order to clarify these aspects of reconfiguration. In [22] and [23] K. Papademetriou et al. offer detailed descriptions of how reconfiguration works internally and an exact cost model that measures accurately the time spent in the reconfiguration process and the actual throughput of FPGAs. That way, we can have a more specific way of measuring the overhead introduced in the system by the scheduling algorithms.

Moreover in [24] the researchers discuss the feasibility for keeping transparent the acceleration of certain kernels to the user, by injecting automatically configuration bitstreams into the FPGA co-processor. Also, in [25] and in more details in [26], K. Papademetriou and A. Dollas present a novel idea, regarding task prefetching and preloading in dynamically reconfigurable processors that can prove very beneficial in future scheduling algorithms.

2.3 Conclusion

As presented above, many researchers have proposed and created many scheduling algorithms, placement algorithms, for managing hardware tasks on a FPGA based partially reconfigurable operating system, and complete runtime systems inhabited in partially reconfigurable devices. However, as we can see from the above descriptions of these works, very few of the scheduling and/or placement algorithms have been evaluated using an actual partially reconfigurable device. On the contrary most of them, if not all, have been evaluated using a C-language discrete-time framework.

That is due to the inability of those designs to be implemented on a realistic FPGA device, which derives from the technology restrictions neglected, while creating the algorithms. Also almost all of the works listed, have made many and sometimes over-simplifying assumptions regarding the partial reconfiguration process.

In Chapter 4 of this thesis, those assumptions will be presented, additionally with the technology restrictions and we will present, why most of the above-mentioned works cannot be implemented in a real device.

Chapter 3

Development of scheduling algorithms for Partially Reconfigurable FPGAs.

In this Chapter, after studying and analyzing the most notable works on the field of hardware task scheduling in partially reconfigurable FPGAs, we will present three scheduling algorithms implementations, we made. The algorithms implemented, in C language, and presented here are:

- Steiger's 1D Horizon Technique
- Steiger's 1D Stuffing Technique
- Steiger's 2D Stuffing Technique, with Bazargan's Shorter Segment Heuristic

Our work was focused on Steiger's algorithms, because these algorithms are the ones that influenced most of the following work done on the area. First, we briefly analyze the two main models used for the mapping of tasks in a reconfigurable device. Then we present and analyze our implementations of the above-mentioned algorithms and we continue analyzing, if any of these algorithms, along with the algorithms presented in Chapter 2 are implementable in a realistic Partially Reconfigurable FPGA. Finally we begin the discussion of the restrictions induced by the FPGA technology and how these restrictions affect the implementation of, almost every algorithms presented. It must be noted that all the schedulers presented in this Chapter consider tasks with deadline, i.e. a task must complete its execution before its deadline time is met.

3.1 Basic Device Area Models.

The complexity of mapping tasks to devices depends heavily on the area model used. In the area of Reconfigurable Computing two area models are commonly used from researchers. In the simpler 1D area model, tasks can be placed anywhere along the horizontal device dimension, the vertical dimension is fixed and covers the total height of the hardware task area. Despite the fact that the 1D area model leads to simplified scheduling and placement problems, it suffers from two types of external fragmentation.

The first one occurs when the hardware tasks do not utilize the full height of the reconfigurable area. The second one occurs when the remaining area is split into several small but non adjacent vertical stripes. The creation of external fragmentation is a huge disadvantage of scheduling algorithms as it can prevent the placement of a task on the device, despite the fact that enough area exists.

The more complex 2D area model allows us to place hardware tasks anywhere on the reconfigurable area and thus creates less external fragmentation. As a result, the 2D area model, offers higher device utilization.

3.2 A C-Language Implementation of Steiger's 1D Horizon Technique.

In [10] Steiger et. al presented the H*orizon Technique*, which was the first attempt in creating an online scheduling algorithm designed for operating systems targeting reconfigurable embedded platforms. Our C-language implementation of the 1D Horizon Technique will be presented in this section.

The 1D Horizon Technique maintains three linked lists, the *reservation* list, the *execution* list and the *scheduling horizon.* The reservation list (R) holds the currently scheduled, but not yet executed tasks, the list entries the task's number, its placement and its starting time and is sorted according to increasing starting times. The execution list (E) hold all the currently executing tasks, the list entries, the task's number, its placement and its finishing time, the list is sorted according to increasing finishing times. Finally the scheduling horizon list (H) consists of all the intervals [x_I, x_J] in the device with their release times. In order for such an interval to exist in the list at a certain point in time, it must not be occupied after its release time, by any task neither in the reservation nor execution lists, the list is sorted according to increasing release times. The linked lists are a very useful structure in maintaining the data needed for the scheduling process. An example of the lists state at a random point in the algorithms execution time can be seen in Figure 1.

$$\begin{aligned}\underline{t = x}\text{: } H &= \{[1,5]@20,\ [5,10]@14\}\\ E &= \{(T_2, 5, 10), (T_3, 1, 14)\}\\ R &= \{(T_4, 5, 10), (T_5, 7, 10), (T_6, 1, 14)\}\end{aligned}$$

Figure 3.1: A graphical representation of the lists used in the *Horizon Technique* at a random point in the execution of the algorithm.

The Horizon Technique's main execution flow can be described in three steps. At each time the online method first checks for terminating tasks, i.e., tasks with finishing time equal to the current time. Then reserved tasks with starting time equal to the current are removed from the reservation list and added to the execution list. Finally for each newly arrived task the scheduling function is called, which either accepts a task, therefore adding it to the reservation list, or rejects it.

When a new task arrives the scheduler walks through the list of all horizon intervals with release times equal to the arrival time of the task and checks whether the task can be appended in the horizon. At any point of this walk the scheduler maintains a new list L, which contains all these intervals (line 2). Then the scheduler calls the *BestFit* function (line 4), which selects the smallest interval from the L list, that is large enough to accommodate the new task, the function then returns either an empty set, when no such interval exists, or the placement *x* for the task. If such a placement exists, the task is added to the reservation list, if not the schedulers proceeds to the next release time of the horizon list and merges adjacent horizon intervals with this release time. If at any point, the next release time is bigger than the latest starting time of the task, the task is rejected. A task's latest starting time is calculated, by the scheduler, and it is the difference between its deadline time and its execution time. As a result the task's scheduling period is the time interval [arrival time, latest starting time]. The pseudocode for the horizon scheduler is shown in Algorithm 1.

Algorithm 1: 1D Horizon Scheduler $\sigma_{1D\text{-}horizon}(T_i, H)$

```
1.  $t \leftarrow a_i$
2.  $L \leftarrow$ horizon interval with $t = t_r$
3.  while ($t \le s_{i\text{-latest}}$) do
4.      $x = BestFit(L, w_i)$
5.      if $x \neq 0$ then
6.          add reservation($T_i, x, t$)
7.          return(ACCEPT)
8.      end if
9.      $t \leftarrow t_r$ of the next horizon interval
10.     $L \leftarrow MergeIntervals(L, H, t)$
11. end while
12. return(REJECT)
```

In order to make the scheduler more easy-to-understand many functions were created. The main functions were, *schedule, BestFit, MergeIntervals* and all the functions needed for list management, i.e. add, remove and print. All these were the functions that were presented by Steiger, however there is another one, that is not mentioned on Steiger's work. That is the *UpdateHorizon* function, which must be called after every function that performs a change in the execution and/or the reservation lists. After this analysis we continue to the implementation of the scheduler. At the first phase we tried to maintain the order that Steiger suggested in his work, first check for terminating tasks, then execute waiting tasks and finally schedule.

After many tries it was concluded that in a C simulation of the scheduler this order cannot be maintained. In C things are done sequentially or the use of threads and multiprogramming is required. The main problem with Steiger's order was that when a task arrives at t_sim=*x,* it is scheduled with t_start=*x.* Therefore, this task will never be added to the execution list, because the scheduling at Steiger's order is done at the end, so when the algorithm tries to add the task to the execution list t_sim would be *x+1,* which will be different from the starting time assigned to the task. Thus, we decided to first schedule tasks, then complete the execution of any finishing tasks and finally add starting tasks to the execution list.

Subsequently, we began the development of the four above-mentioned functions. While

Steiger fully explains the way, the *schedule* function works*,* the absence of pseudocode for the *BestFit, MergeIntervals* functions, required some improvisations. That was more necessary with the *UpdateHorizon* function's implementation, which is based in an original idea of ours.

The *schedule* function's inputs are: the task to be scheduled, the horizon list and the maximum width of the reconfigurable region. The function returns a pointer to a structure consisting of the task's placement, starting time, number and a number representing whether the task has been accepted or not. The way the function works is exactly as described by Steiger, with no modifications made by us.

The *BestFit* function's inputs are: the *L* list, which consists of horizon intervals, the task's width, the current scheduling time, i.e. the current *L* intervals' release time, and the maximum width of the reconfigurable region. The function walks through the *L* list trying to find the smallest interval, that can accommodate the scheduled task. If found the function returns the x_l value of the interval, as the placement of the task or the value -1, if no such interval exists.

The *MergeIntervals* function's inputs are: the current *L* list, the horizon list and the time *t* at which the merging will occur. For each horizon interval *X* in the horizon list with release time equal to *t* the function checks all the horizon intervals with equal or lower release times and if they are adjacent to *X* it merges them. After all the intervals are checked with *X*, the final merged interval is added to the *L* list and the next horizon interval undergoes the same processing, only if it was not one of the intervals that *X* was merged with. Upon completion the function returns the head of the *L* list.

Finally, we explain, the *UpdateHorizon* function, which was the most demanding function to implement as there was no description of it in Steiger's paper. The only information for the function, was its functionality:

- *The UpdateHorizon function updated the horizon list with adding or removing intervals according to the current state of the reservation and execution lists.*

From that first description of the function its inputs were determined; the reservation list, the execution list, the task list, in order to have specific information for every task, the current simulation time and the maximum width of the reconfigurable device. First a temporary horizon list with one-width intervals [0, x] and release time the current simulation time is created, that fully partitions the spatial resource dimension.

Then every interval is cross-checked with all the entries of the reservation list, if at any point any task uses the interval, its release time is set as the maximum finishing time, i.e.

the starting time of the task, plus its execution time. Then upon completion of the check, the new interval was added in the temporary horizon list. The same check is performed with the tasks in the execution list. After these checks the temporary horizon list consists of many one-width intervals with various release times, in order to list every interval only once we kept in the list only the one-width intervals with their maximum release times. Finally the adjacent intervals with same release time were merged and the resulting interval was added to the horizon list. The final horizon's list head was the output of the function.

The other functions needed for the completion of the Horizon Scheduler were trivial list management functions, for each list used in it. In order to examine the accuracy of our implementation, we created a task set same to the one Steiger uses in his paper, then we cross-checked the state of the reservation, execution and horizon lists, at simulation times 1, 2, 3 and 21 with Steiger's. The task set we created is shown below:

Steiger's Task Set Table

Task	Arrival Time	Execution Time	Deadline	Width	Height
T1	0	20	30	3	3
T2	0	3	10	7	5
T3	1	12	15	3	5
T4	1	3	10	2	2
T5	2	2	10	3	4
T6	2	3	20	5	1
T7	3	2	20	3	2

The above task set produced the same results as Steiger mentions in his work. At Figure 2 a graphical representation of the reconfigurable area after the arrival and scheduling of all the tasks is shown.

Thus having confirmed the functionality of the Horizon Technique we proceeded in testing the Horizon Technique in a corner situation, where it produced high space and time fragmentation. The task set we constructed was:

High Fragmentation Task Set Table

Task	Arrival Time	Execution Time	Deadline	Width	Height
T1	0	5	20	10	2
T2	0	10	15	3	4
T3	1	3	10	2	2
T4	1	3	10	4	2
T5	1	3	15	3	5
T6	2	4	20	10	3
T7	3	3	20	3	2

With this task set the Horizon Technique has one rejected task (T7). Although if we observe the scheduling process and the utilization of the reconfigurable area it is clear that this rejection can be avoided. More specific at t_sim=2 the reconfigurable area is shown in Figure 3.

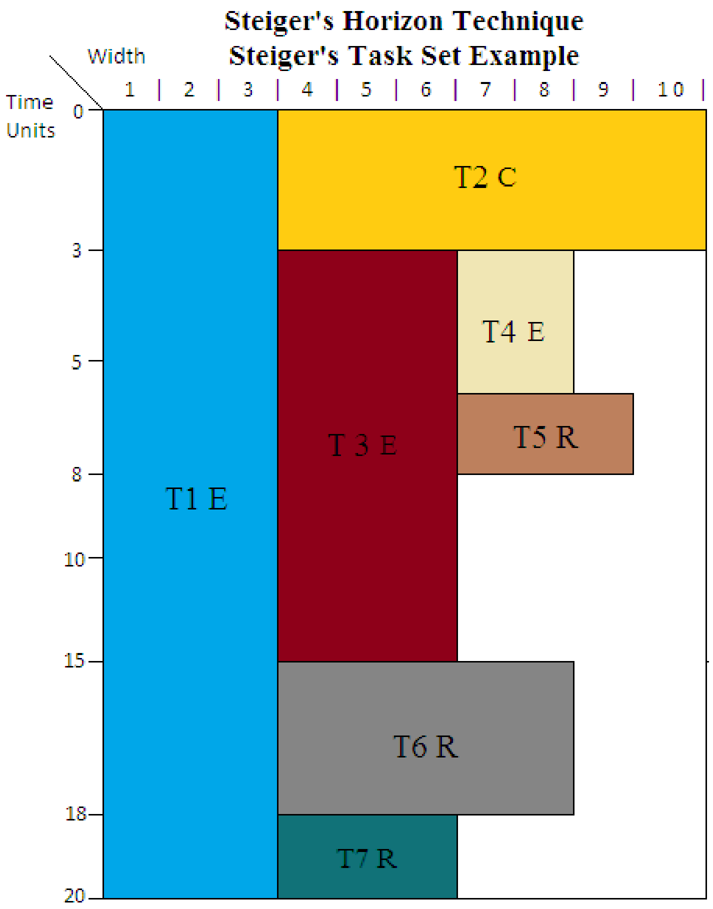


Figure 3.2: A graphical representation of the reconfigurable area at t_sim=3. The E means that the task is currently in execution, the R means it is reserved and the C means the task is completed

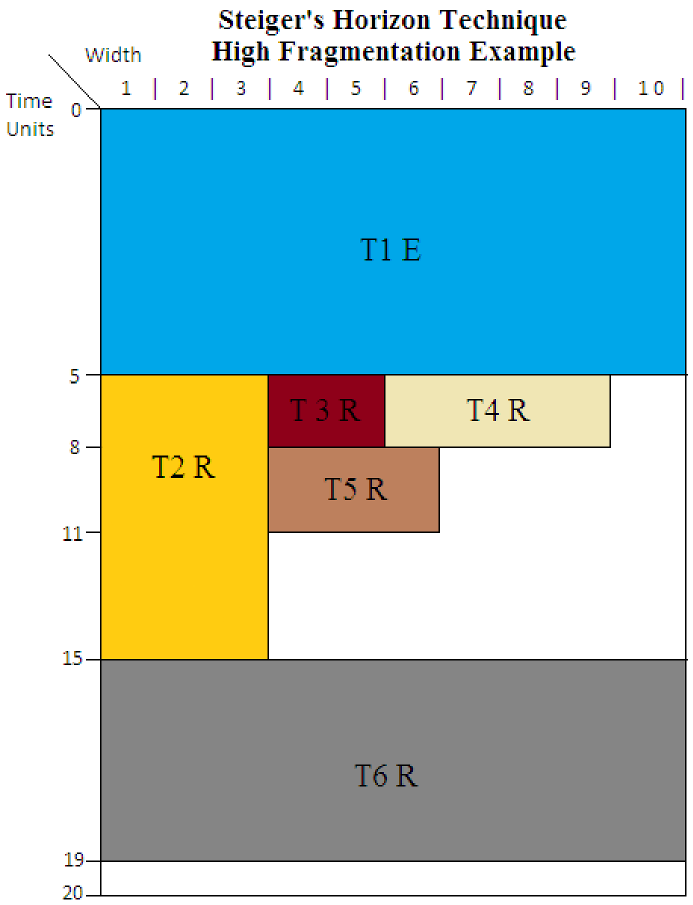


Figure 3.3: A graphical representation of the reconfigurable area at t_sim=2. The E means that the task is currently in execution, the R means it is reserved. In this example task 7 is rejected.

When Task 7 arrives plenty of area exists in the device for it to be accommodated in. In fact two placements for T7 can be found, either placement 7 with starting time 8 or placement 4 with starting time 11, but instead the task rejected. In order to understand why that happens, we need to observe the Horizon List, the algorithm produces. Specifically the Horizon List after the scheduling of T6 is:

```
******HORIZON LIST******
Horizon interval x1 1, x2 10, release time 19.
```

So the Horizon Technique, in order to list an interval at the Horizon List, needs the interval to be free from the time of its release and to not be occupied afterward, as mentioned before.. That is the reason why the intervals 8-9 at release time 8 and 4-6 at release time 11 are not considered as possible entries in the Horizon List, as both of them are later occupied by Task 6. That is the main disadvantage of the Horizon Technique as it produces high area fragmentation.

3.3 A C-Language Implementation of Steiger’s 1D Stuffing Technique.

In the previous section we presented Steiger's Horizon Technique. It should be acknowledged that the Horizon Technique has faults, especially regarding the free area management. However it was a first attempt from Steiger to successfully schedule hardware tasks in a partially reconfigurable device. Having realized the disadvantages induced by the maintenance of the Horizon List, Steiger tried to make the area management process more efficient, the resulting algorithm was the Stuffing Technique. In this section we present our C-language implementation of the 1D Stuffing Technique.

The Stuffing Technique also maintains three linked lists, the *reservation* list, the *execution* list and the *free space list.* The reservation (R) and execution (E) lists hold the appropriate information for each task listed in one of them, task number, starting time, placement for the reservation list and task number, finishing time, placement for the execution list. Finally the free space list consists of all the intervals [x_I, x_J] that identify currently unused resource intervals in the device, sorted according to increasing *x*-coordinates. The main difference between the Horizon and the Stuffing Technique can be seen in the *free space list,* while in the Horizon Technique the intervals are list only if there release is permanent until the current time, in the Stuffing Technique lists the all the intervals that are currently unused, regardless of whether or not they will be used in a future time.

The scheduling sequence for the Stuffing Technique is the same as the Horizon one, first the algorithm checks for terminating tasks, i.e. tasks with finishing time equal to the current time. Then reserved tasks with starting time equal to the current are removed from the reservation list and added to the execution list. Finally for each newly arrived task the scheduling function is called which either accepts a task, therefore adding it to the reservation list, or rejects it.

When a new task arrives, the scheduler starts walking through the task's planning period, simulating all future allocations of the device, by simulating future task terminations and starts together with the underlying free space management. In line 1 of Algorithm 2, the current free space is copied to a simulated free space list, which is then modified during the scheduling process. At any given time, the scheduler first checks for terminating tasks and then reserved tasks are started. In line 12, the function *BestFit* returns all the intervals in the simulated free space list that can accommodate the newly arrived task or returns an empty set if no such interval exists. The reported intervals are then checked for conflicts with existing reservations in a best-fit order. If an interval without conflict is found, Ti is accepted and the planning stops. Otherwise, the scheduler proceeds to the next event, until the end of the scheduling period of the task. The pseudocode for the Stuffing Technique is shown in Algorithm 2.

Algorithm 2: 1D Stuffing Scheduler $\sigma_{1D\text{-Stuffing}}(T_i, F)$

```
1.  $F_S \leftarrow F$; $t \leftarrow a_i$
2.  check ← TRUE
3.  while ($t \leq s_{i\text{-}latest}$) do
4.      for all $T_j \varepsilon E$ with ($f_j = t$) do
5.          TerminateTasks($T_j, F_S$)
6.          check ← TRUE
7.      end for
8.      for all $T_j \varepsilon R$ with ($s_j = t$) do
9.          StartTask($T_j, F_S$)
10.     end for
11.     if check then
12.         $X \leftarrow BestFit(F_S, w_i)$
```

```
13.         for all ($x \varepsilon X$) do
14.             if (($x, t, x+w_i, t+e_i$) is not conflicting any
                reservation in $R$) then
15.                 add reservation ($T_i, x, t$) to $R$
16.                 return(ACCEPT)
17.             end if
18.         end for
19.         check ← FALSE
20.     end if
21.     $t$ ← next event form $E$ U $R$
22. end while
23. return(REJECT)
```

This scheduler was also created with the use of many functions for increased efficiency and less complexity. A first examination showed that the needed functions were, *schedule, BestFit* and all the functions we needed for list management, i.e. add, remove and print. However after careful consideration two functions that are not mentioned by Steiger were found. The first was the *UpdateSpace* function, which is called after every function that performs a change in the execution list. The second was the function *check_conflict,* which is used inside the *schedule* function, but we decided it would be better if implemented as a separate function, in order to have better classification of the functions in our scheduler.

Subsequently we began the development of the four above-mentioned functions. While Steiger fully explains the way the *schedule* function works*,* the absence of pseudo code for the *BestFit* function, required some improvisations. Although the implementation of the *BestFit* function is the same as the one in the Horizon Technique, the *UpdateSpace* and the *check_conflict* functions, were solely based in our ideas.

The *schedule* function's inputs are: the task to be scheduled, the tasks list, the execution and reservation lists and the free space list. The function returns a pointer to a structure consisting of the task's placement, the task's starting time, the task's number and a number representing whether the task has been accepted or not. The way this function works is exactly as described by Steiger, with no modifications made by us. Though some extra logic was added to this function, for example we constructed a new linked list, called *events* which holds the times of each event in the reservation and execution lists and also adds the finishing event time of a simulated starting task during scheduling.

The *BestFit* function's inputs are: the simulated free space list, i.e. the F_S *list* and the task's width. The function walks through the free space list trying to find intervals that can accommodate the scheduled task. The function returns the set of intervals found or an empty set if no such interval exists.

Next the *UpdateSpace* function, which was the most demanding function to implement as there was no description of it in Steiger's paper, is presented. The only information for the function, was its functionality:

- *The UpdateSpace updated the free space list with adding or removing intervals according to the current state of the execution list.*

From that first description of the function we determined its inputs; the execution list, the task list, in order to have specific information for every task, and the maximum width of the reconfigurable device. In the beginning, every one-width interval that fully partitions the spatial resource of the device, i.e. from interval [0,0] to [x, x], where x the maximum width of the reconfigurable area, is cross-checked with all the entries in the execution list. If any task in the execution list uses the interval, the interval is not added in the temporary free space list. After this check the temporary free space list consists of many one-width intervals, for every adjacent interval we perform a merge and then the merged entry is added to the free space list which is the output of our function. If no tasks exist in the execution list then the function returns a set, which contains the whole width of the device.

Finally the *check_conflict* function was implemented. The function's inputs are: the set of intervals produced by the *BestFit* function, the time during which the check occurs, the reservation list and the scheduled task. For each interval in the intervals list the function checks, if it conflicts in space and in time with any entry in the reservation list. If the above conflict occurs the interval is rule out, as a suitable placement option for the task. If more than one intervals are eligible for placement the choice between them is done with a best-fit fashion.

The other functions needed for the completion of the Stuffing Scheduler were trivial list management functions, for each list used in it. In order to examine the accuracy of our implementation, we created a task set same to the one Steiger uses in his paper, then we cross-checked the state of the reservation, execution and free space lists, at simulation times 1, 2, 3 and 21 with Steiger's. The task set we created is the same as the one used in Section 3.2. The task set produced the same results as Steiger mentions in his work. At Figure 4 a graphical representation of the reconfigurable area after the arrival and scheduling of all the tasks is shown.

Steiger introduced the Stuffing Technique in order to correct certain drawbacks he had in the Horizon Technique. One of them was the necessity for the Horizon Technique to consider intervals that were free must be appended to the horizon. That rule created high fragmentation in many tasks sets we created. When the input to the Stuffing Technique was the high fragmentation task set used in Section 3.2, all tasks were successfully scheduled and the device's space was utilized much better, as seen in Figure 5.

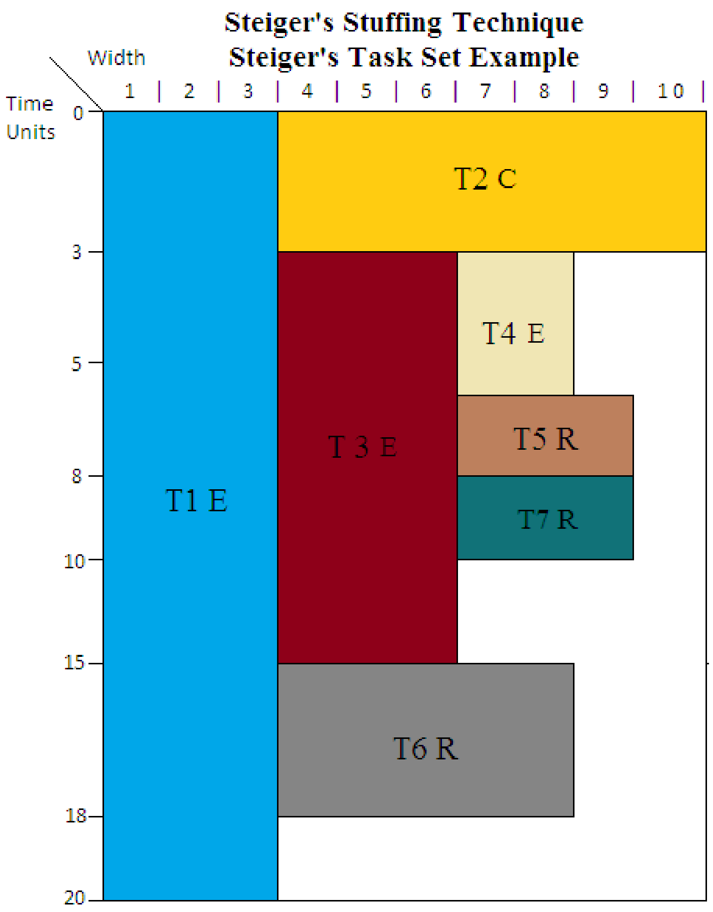


Figure 3.4: A graphical representation of the reconfigurable area at t_sim=3. The E means that the task is currently in execution, the R means it is reserved and the C means the task is completed

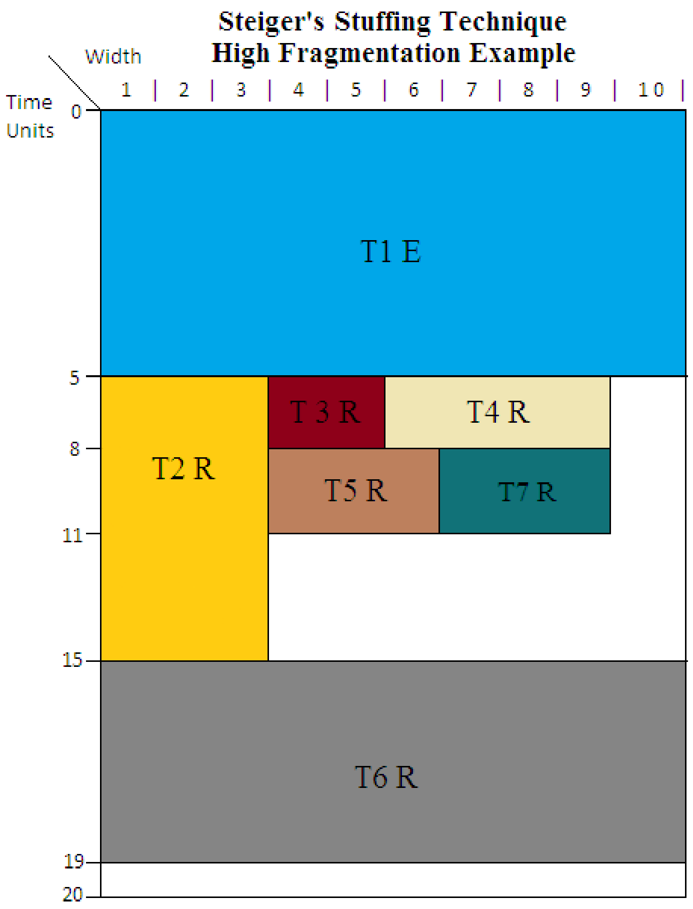


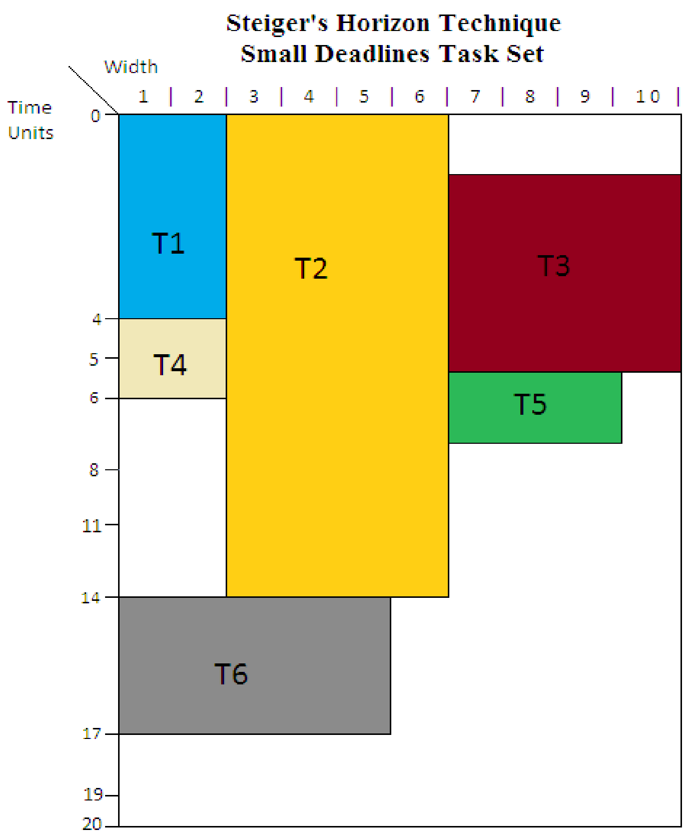
Figure 3.5: A graphical representation of the reconfigurable area at t_sim=2. The E means that the task is currently in execution, the R means it is reserved. In this example task 7 is accepted.

Another advantage found in Stuffing is that the scheduler can handle well tasks, with really small deadlines. In order to show that we created the task set show bellow.

Small Deadlines Task Set Table

Task	Arrival Time	Execution Time	Deadline	Width	Height
T1	0	4	10	2	3
T2	0	14	20	4	5
T3	1	4	15	4	5
T4	2	2	6	2	2
T5	2	2	10	3	4
T6	2	3	17	5	1
T7	3	2	8	2	2

In the above task set we can see that task 7 has a scheduling period of only 3 time units, while task 4 has a scheduling period of only 2 time units. When in this task set the Horizon Technique, is applied, task 7 is rejected, because although there is space in the left side of the device, the deadline of the task is not met. Also we can see that there is space in the right side of the device also, but the Horizon algorithm dismisses that space as it is not free in the entire time horizon.



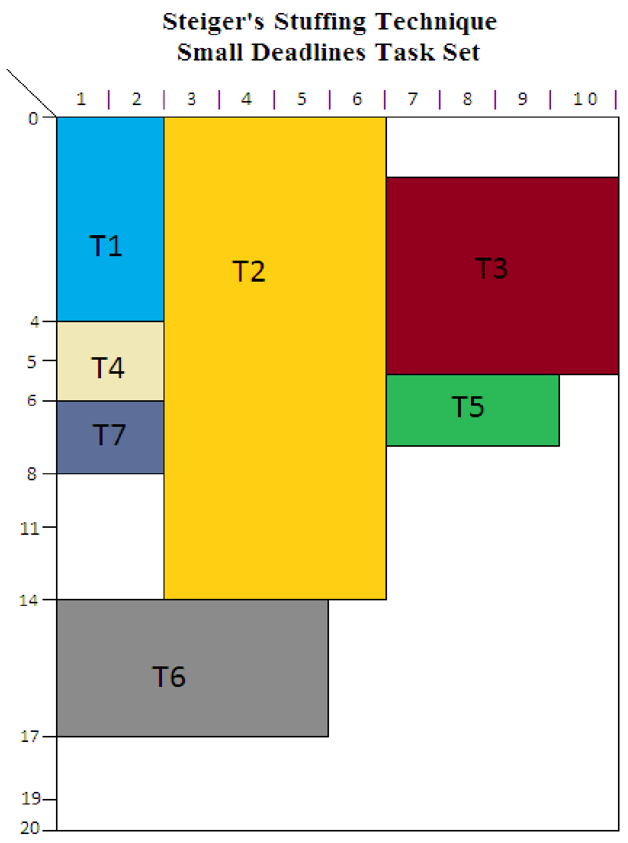


Figure 3.6: A graphical representation of the reconfigurable area after the arrival of all the tasks. We can see that although enough space exists for the planning of task 7, the Horizon Technique rejects it, while the Stuffing Technique accepts it.

When the Stuffing Technique is applied to that task set, all the tasks are scheduled successfully. The device after the arrival of all the tasks for the two algorithms is shown in Figure 6.

3.4 Expansion to the 2D model of Steiger's Stuffing Technique.

The algorithms described so far are designed for the 1D model of a reconfigurable device. However the space utilization achieved with the 1D model is not ideal, in order to achieve better results on resource utilization researchers directed their work on the 2D area model. In the previous section, the superiority of the Stuffing Technique against the Horizon, in the 1D model, is shown. For that reason the scheduler we choose for further expansion to the 2D area model is the Stuffing Technique.

During the expansion of the Stuffing Technique to the 2D area model, several problems must be considered. First it was clear that the execution flow of the algorithm would remain the same as the 1D one. Also it was observed that the main functions, i.e. *schedule, BestFit* and *check_conflict,* used for the 1D Stuffing Technique could be easily upgraded to support the 2D area model. Also the lists used for maintaining the data the scheduler needs, i.e. *execution, reservation* and *free space* lists, could also be used in the 2D area model with minor modifications. It is important to note that now the *free space list* entries, list a space rectangle like that [x_I, y_I, x_J, y_J]. However, the most important and difficult changes in the scheduler were relevant to the placement management.

When considering a two-dimensional plain, the problem of managing the space left becomes a problem of maintaining a list of free maximal rectangles in space and time. This problem can be compared to the 2D bin packing problem, which is known to be a combinatorial NP-hard problem. For this problem several solutions have been proposed, most of them consider different ways to partition the device, in order to create maximum rectangles.

In [10] Steiger et al. do not present the placer or the way the 2D Stuffing Algorithm manages the free space, instead the placer used is presented in [16]. The placer considers no permanent partitioning of the device, but instead creates overlapping rectangles, in order to have at each point the maximal rectangles as seen in Figure 7.

During the implementation process of the algorithm, we made, excessive efforts to recreate Steiger's placer. However, due to the lack of information provided by Steiger in

[16], our tries were unsuccessful. Also because of our studies, regarding the technology restrictions in FPGA devices, we were already aware of the infeasibility of Steiger's design. More on that matter will be presented later. Nonetheless this failure prompted us to search for other solutions.

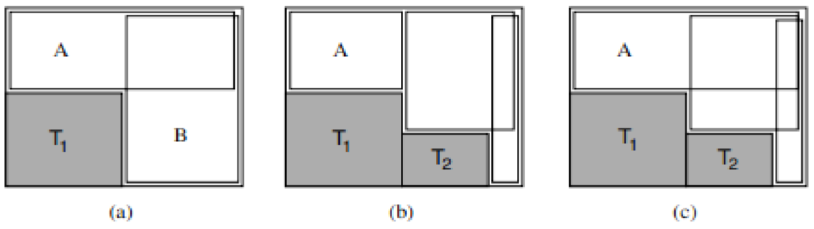


Figure 3.7: Here we see the place management technique presented by Steiger in [16]. We can observe that the splitting decision is delayed by the placer algorithm, thus creating two or more overlapping rectangles, representing the free space in the reconfigurable area.

In [16] it is clear that the main influence of the placer presented is Bazargan's work in [15]. In his work Bazargan presents many heuristics that will ultimately help the placer perform the best split possible, the results of the experiments showed that the best heuristic was the Shorter Segment Heuristic (SSEG). In that heuristic after the insertion of a task, the split is performed, choosing the shorter of the two segments created. A simple example of Bazargan's SSEG is shown in Figure 8. So we decided the *UpdateSpace* function used for place management to be created according to Bazargan's SSEG Heuristic.

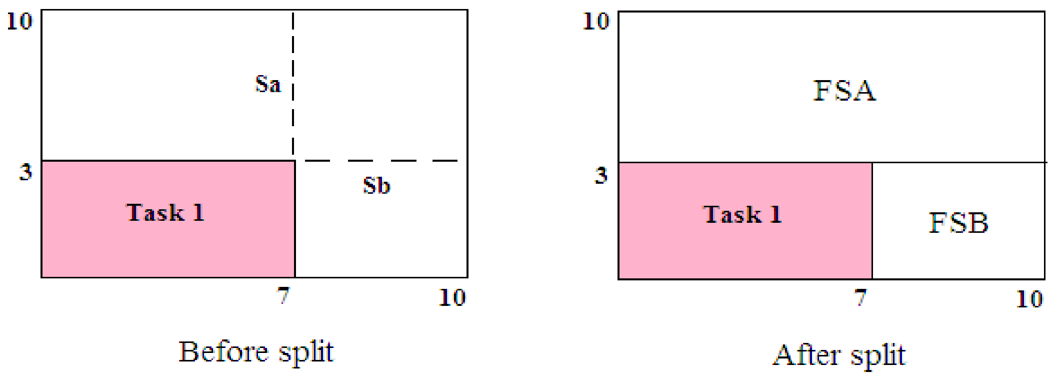


Figure 3.8: A representation of Bazargan's SSEG Heuristic. Before the splitting of the reconfigurable area the placer considers the length of the two segments *Sa* and *Sb* and splits the area along the shorter one, thus creating Free Space Area *A* and *B*.

Despite the fact that a solution was found and the expansion in the 2D model could be made, this solution introduced a huge disadvantage. More specifically after many task insertions, it was observed that the reconfigurable area would be split in many small rectangles. That meant that although enough area would exist on the device to accommodate a newly arrived task, the partitioning of this area would not allow the acceptance of said task.

An example of that can be seen in Figure 9. In this example, we see that the reconfigurable area is split in two Free Space rectangles, ([8,1], [10,3]) and ([8, 4], [10, 10]). The FSA rectangle is 3 space units wide and 7 space units high while the FSB rectangle is 3 space units wide and 3 space units high. The newly arrived task however is 3 space units wide and 10 space units high. Due to the current partitioning of the free space area this task would be rejected. However if a merging would occur between the two rectangles, enough space would exist for the task to be placed on.

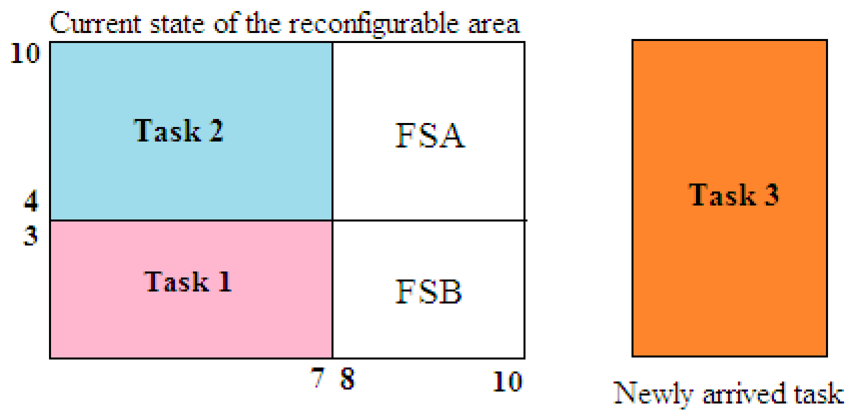


Figure 3.9: Example of task rejection due to the splitting of the reconfigurable area with Bazargan's SSEG Heuristic.

In order to overcome this disadvantage a new *mergeIntervals* function was created. In this function, when two adjacent rectangles are found, the algorithm merges them into one, while removing the merged intervals from the *free space list.* More specifically the function would merge adjacent rectangles that had at least one equal dimension. In the figure above the *mergeIntervals* function would merge the FSA and FSB rectangles, as they both have a width of 3 space units. into one Free Space rectangle, large enough to accommodate Task 3.

Another change needed for the expansion of the Stuffing Technique in the 2D area model was the creation of a new function called *removeExecUpdate.* After the completion of a task, that task would be removed from the device. After the removal, this function would add a Free Space entry, to the *free space list* with dimensions equal to the removed task's ones. The new *free space list* would then be given as an input to the *mergeIntervals* function in order to better utilize the free space created from the task removal, applying the appropriate merges.

After implementing all the changes needed for the expansion of the Stuffing Technique to the 2D area model, a task set was created to test the correct functionality of our algorithm. The task set created was Steiger's task set, used in all the previous evaluations, however, in order to present the better space management done by the 2D area model, we decreased the device's dimensions from 10×10 to 10 space units wide and 6 space units high. The final results produced by our program were the following:

Task 7 was rejected!

The number of declined tasks is: 1

The percentage of declined tasks is: 10.00

The results shown above are different, from the results Steiger showed in his work. Specifically this task set produced no rejected tasks when the 2D Stuffing Algorithm was applied in Steiger's. A graphical representation of the reconfigurable area before the arrival of task 7 is shown in Figure10.

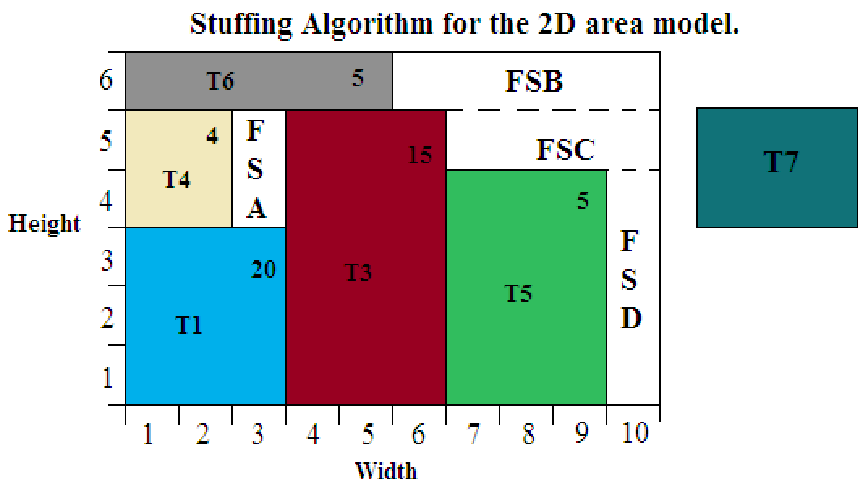


Figure 3.10: A graphical representation of the reconfigurable area before the arrival of task 7. Along with the entries of free space area listed in the *free space list.*

In the figure above it is clear that although there is enough space in the device for Task 7 to be accommodated on, the algorithm rejects this task. In order to understand why this happens the instance of the *free space list* before the arrival of Task 7 is presented.

```
******FREE SPACE LIST******
Free Space interval x1 3, y1 4, x2 3, y2 5.
Free Space interval x1 6, y1 6, x2 10, y2 6.
Free Space interval x1 7, y1 5, x2 10, y2 5.
Free Space interval x1 10, y1 1, x2 10, y2 4.
```

Now it is clear, why our implementation of the algorithm produces a rejected task. The chosen way of merging the intervals in the *free space list* after each update and the choice of conducting the split after the insertion of a task, opposing to Steiger's overlapping rectangles technique, led to this device instance, where the intervals are managed and listed in a way that is not optimal and leads to space fragmentation and finally to the rejection of Task 7. However as stated before this choices were inevitable.

It is important to note that, despite the fact, the same task set used in the 1D model, in 2D model produces one rejected task, ultimately the 2D model surpasses the 1D model in terms of space utilization, as we can see that the device's dimensions are clearly smaller, in the 2D area model evaluation.

With the extension of the Stuffing Algorithm to the 2D area model, the presentation of other scheduling algorithms implementations, comes to an end. All the work done above provided great insight, experience and understanding in the inner works and the design process of a scheduling algorithm for hardware tasks. This experience came to use when designing our scheduling algorithm. Also having a good understanding of the inner works of a scheduling algorithm combined with our knowledge about the technology restrictions induced by the FPGAs, is very important in deciding whether or not a scheduling algorithm could be implementable in a realistic Partially Reconfigurable FPGA.

3.5 The technology restrictions that prevent the implementation of schedulers on realistic FPGAs.

So far several scheduling algorithms have been presented by the research community, most important of which, were analyzed in Chapter 2 of the current thesis. However, it is important to make a more thorough analysis, of said algorithms in order to be able to decide whether or not these algorithms could work on a realistic Operating System targeting a FPGA device. In order to begin this analysis of the scheduling algorithms, first a presentation of the restrictions, involving FPGA technology is made.

Since the first time Xilinx Inc. released FPGAs with the ability of dynamic partial reconfiguration, around 1998, many research groups contemplated the construction of an operating system accommodated in a FPGA. The advantages of such an OS were numerous, first the hardware acceleration produced by the device would make certain processes run much faster that they would in software, also the portability of the device was a great asset, plus the fact that even without an embedded OS the FPGAs were experiencing a vast use amongst experienced and simple users.

However, around that time the Xilinx Company, due to the fear of competitive industries in the field, kept many FPGA technology specifications concerning dynamic partial reconfiguration "secret". This did not discourage the researchers, who made several assumptions and began the creation of an Operating System for Reconfigurable Embedded Platforms.

Be that as it may, many years later, since the dynamic reconfiguration feature was introduced, Xilinx Inc. released the Partial Reconfiguration Design Flow. In that Design Flow Xilinx stated that Reconfigurable Functional Units must be placed and routed, at design time, and configured during runtime on Partially Reconfigurable Regions, Figure 11.

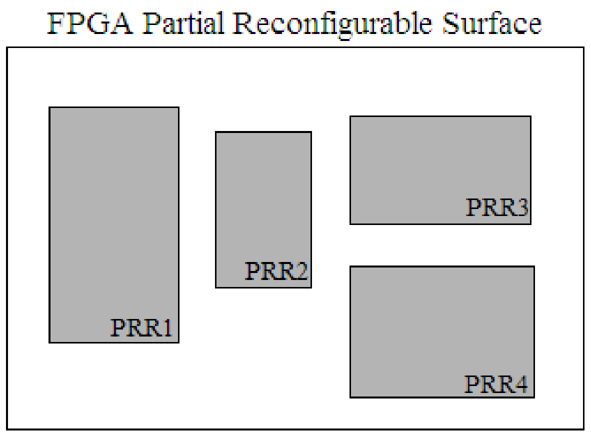


Figure 3.11: The reconfigurable surface of a FPGA can be partitioned into many reconfigurable regions only during compile time.

That was a huge hurdle in all the previously developed placement algorithms and some scheduling ones, which considered online partitioning of the device in order to construct the maximum rectangles. In addition, it also became clear that a PRR must at any point in time be used by only one PRM (Partial Reconfigurable Module). As a result, another

PRM cannot be configured in that same PRR even if there is enough space for it, this restriction is shown in Figure 12. The unusable area is shown with vertical white stripes.

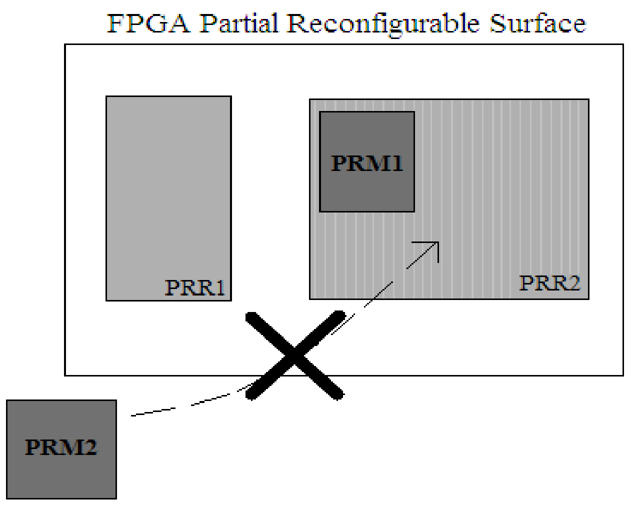


Figure 3.12: Even though there is enough space unused on PRR2 from PRM 2 to be configured into, the technology restrictions regarding PRR use, forbid this action

Finally it became clear that in order to download a PRM on the FPGA a bitstream of said PRM must exist, which is created during compile time and it is bound to a specific PRR. A graphical representation of this restriction is shown in Figure 13.

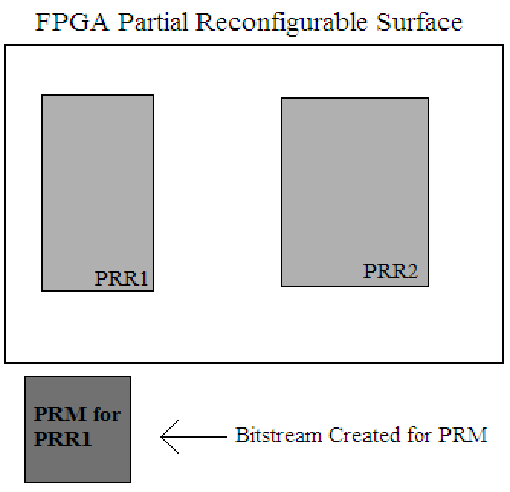

FPGA Partial Reconfigurable Surface
PRR1
PRR2
PRM for PRR1
Bitstream Created for PRM

Figure 3.13: Even though both PRRs have enough space to acccmmodate the PRM shown, the PRM can only be configured on PRR1. This is due to the fact that the bitstream created binds the PRM to PRR1.

Also very few of the scheduling algorithms, described in Chapter 2, have considered inner-task communication or I/O task communication. The way communication occurs in the FPGA devices was known from the very beginnings of research but through the years has been subject of many improvements and changes from Xilinx. The current technology used for task communication is hardware bus macros. According to the PR flow macros can be placed, during compile time, on the boundaries of RRs in order to define pins where RFUs can hook themselves. These macros are made with pairs of CLBs; one side of the CLB pair is connected to a RR signal, while the other is connected to a static logic signal. That means that each RFU can communicate with the static logic part of the device and through that and only that with each other.

Another technology restriction that no scheduling algorithm has considered is the devices' heterogeneity. This means that certain special resources (BRAM, dedicated multipliers, etc.) are in certain positions on the device and not scattered through the whole reconfigurable surface. This fact is not considered by any scheduling algorithms, which consider a heterogeneous design of the device. Moreover some tasks might require these certain resources, so their positioning must be narrowed down to the places on the FPGA, where those resources exist. Although this is a wrong assumption to be made and it was already known to the researchers that FPGA devices are of a heterogeneous design, we can consider some areas of the device to be in fact homogeneous.

Finally a fact that researchers in their works seem to neglect is the ratio between the reconfigurable resources needed by a module and the reconfigurable resources which are part of the PRR. Moreover in several of their works and simulations of them, developers of scheduling algorithms seem to pack the tasks in the reconfigurable region, as much tightly as they can, attempting to reach 100% use of the available reconfigurable area. However that may affect the tasks' communication. A "rule" that designers should use in this occasion is the "80%-90% Golden Rule", which states that, between the used CLBs on a FPGA device and the available CLBs should be at most 90% [27] and [28]. Following this rule; will have a more balanced use between the logic and communication resources provided by the device.

Next some scheduling algorithms alongside with their placement policies, are presented, along with the assumptions that these algorithms are based on and which of them can be potentially overlooked. It is important to note, that the technology restrictions analyzed earlier, are not mentioned in the works described below, a sign that researchers spent more time creating near-optimal algorithms than studying the technology restrictions:

- First the Horizon and Stuffing Techniques proposed by Steiger et al. in [10] are presented, the functionality and our implementations of these Techniques are presented in Sections 3.2, 3.3 and 3.4 of this Chapter. While in his work Steiger fully describes a model that has a communication channel, loosely based on the macros provided by Xilinx, he only made simulation experiments without communication between tasks, so we cannot be sure in which degree these Techniques would work if communication is taken into account. Additionally the system described by Steiger has the reconfigurable region partitioned into columns, at compile time, but the scheduling algorithm presented considers runtime partitioning, mapping and routing of the tasks on the device, which is a clear technology restriction violation. In Figure 14 we can see the system presented by Steiger.

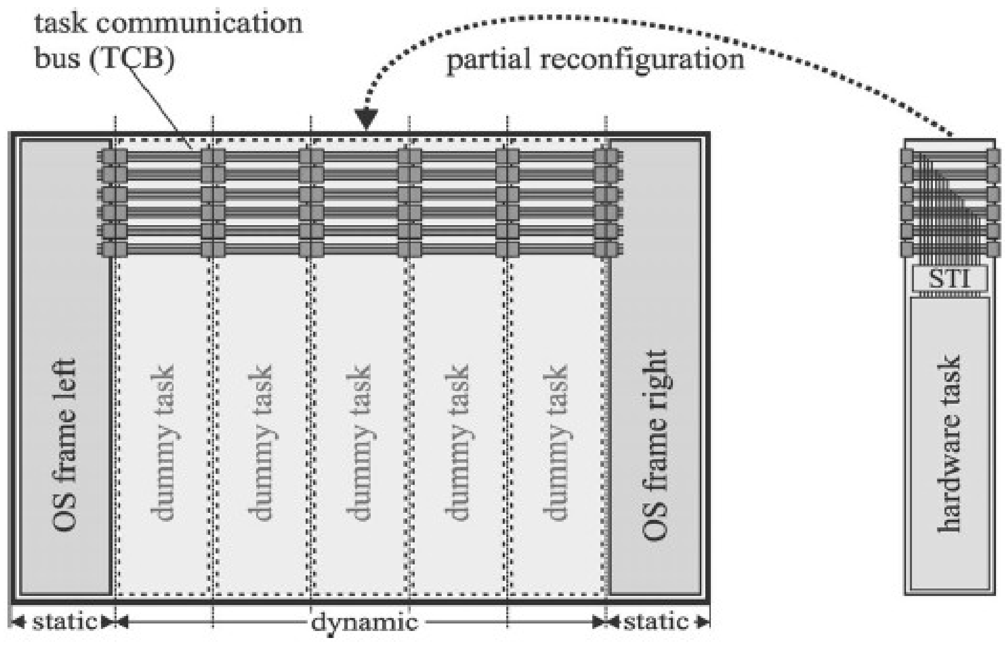


Figure 3.14: Here we see the system model presented by Steiger. In this Steiger considers pre-defined PRRs and a Task Communication Bus, based on hardware macros. However these ideas are disregarded during the development of the Horizon and Stuffing Techniques.

- Communication Aware algorithm by Lu and Marconi [14]. This algorithm partitions the device in design time first into columns and each of them into configuration blocks, which will be the PRRs (Partial Reconfigurable Regions) managed by the algorithm. Next the algorithm manages and configures tasks into the device according to their communication needs. Even though the algorithm pre-partitions the device, the scheduling algorithm uses a placer that partitions the device online and manages, applying further partitions or merges, the maximal free rectangles in an arbitrary way. All the communication buses Lu and Marconi suggest can be implemented with the Xilinx provided hardware macros, but they neglect the fact that these macros need to be placed also at compile time. The FPGA paradigm Lu and Marconi use is shown in Figure 15. It is important to not that the communication infrastructure proposed by Marconi is correct, according to the FPGA technology.

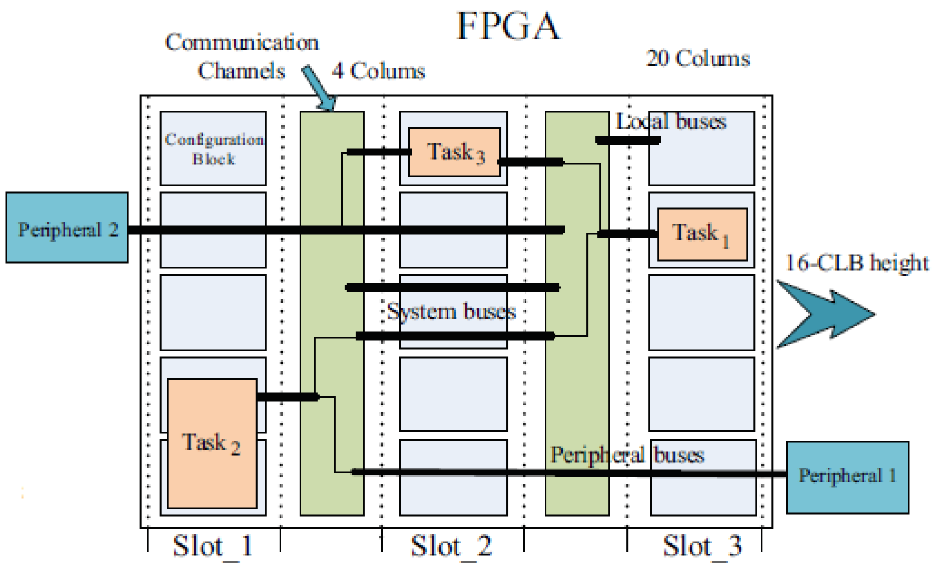


Figure 3.15: Here the system used by Marconi in [14] is presented. Even though Marconi pre-partitions the FPGA's Partial Reconfigurable Surfcace, first into columns and then in configuration blocks, he then assumes that the algorithm can perform further partitions and merges of these blocks at run time, which constitutes a technology violation.

- 3D Compaction by Marconi et al. [13], [30]. The algorithm uses the 3D Total Contiguous Surface Heuristic, which packs compactly the tasks by calculating the “touching” area between the new task and previously scheduled tasks or future ones and the device's boundaries. Their model does not support communication between tasks and considers a homogeneous design. The 3DTCS Heuristic may cause serious problems with the tasks' execution and communication as it violates the common 80%-90% “Golden Rule”. The 3DTCS Heuristic is also in violation of the fact that PRRs need to be predefined and that one PRR can be used from only one PRM at any time. Also Marconi et al. consider no partitioning of the device, leaving the scheduling algorithm to manage one big PRR, which it is then, partitioned and merged at runtime. Also, the concept of tightly packing the tasks in the partial reconfigurable surfaces considers a homogeneous design of the device. The functionality of the 3D Heuristic is shown in Figure 16.

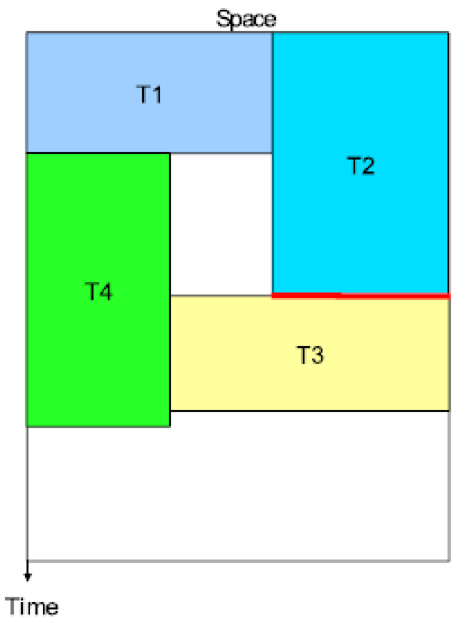


Figure 3.16: The basic idea of the 3DTCS Heuristic is to place the tasks in a way that will maximize the "touching" area between them.

- Most Frequently Used, Best Speedup, Multi-Constraint Knapsack by Fu and Compton [8]. These algorithms proposed decide on which kernels, i.e. hardware tasks, should be implemented in hardware, provided they fit in the available space. Here the algorithms work in an off-line scenario, where the user has many kernels and needs to decide, which of them is best to implement in hardware. Due to the offline scenario, the algorithms don't make any assumptions and can work with any of the restrictions provided by the device's technology. Although Fu and Compton do not present a complete system description in [8] we can assume that their algorithms can work with a pre-partitioned device. However, considering the simplicity of these algorithms it can be assumed, that they can also work at online scenarios.
- Intelligent Stuffing algorithm by Marconi and Lu [11]. The algorithm presented is based upon Steiger's Stuffing Technique. As Steiger did, Marconi also assumes an online scenario of his algorithm, where the device can partition the reconfigurable surface and place tasks in seemingly random areas on the device, during runtime. Apart from that Marconi takes no account in communication between tasks and I/O and considers a homogeneous device design. A paradigm of the Intelligent Stuffing algorithm is shown in Figure 17.

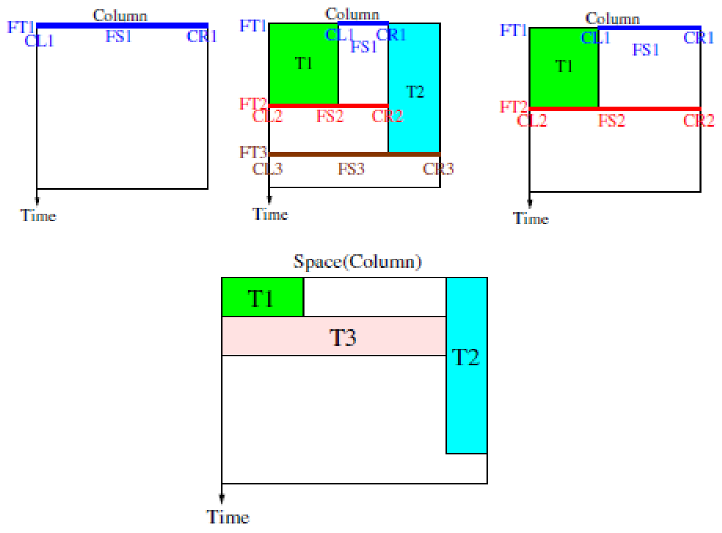


Figure 3.17: Here it is shown a paradigm of how the Intelligent Stuffing Technique works, presented by Marconi in [11]. It is clear that during the scheduling of a task the algorithm partitions the device into "Free Space" (FS) rectangles. As it is clear that this is an online scheduling algorithm, this partitioning is performed during runtime, which is impossible with current FPGA technology.

All the algorithms described above also consider the tasks as relocatable rectangles that can be placed anywhere in the 2D area model. This is a huge assumption that is made by almost anyone who wishes to develop a scheduling algorithm. The reason most of the researchers use this is, because they consider that with the use of relocation, they can override the restrictions inducted by Xilinx, considering the bitstream PRR-PRM binding and the necessity of the pre-definition of PRRs at compile time. The starting point for this assumption was the work done by Katherine Compton in [28].

Many researchers, referencing the work of Katherine Compton on task relocation, assume that tasks can become subject to many alterations concerning their placement and can overcome many footprint transforms, such as flips, both horizontal and vertical, rotations and vertical and horizontal offset movement. However, technology does not yet support the majority of these functions regarding task transformation, alongside with task relocation. The way task relocation is used by the researchers, is to override the fundamental requirement that partial bitstreams, for reconfigurable modules, are created in direct accordance with the Partial Reconfigurable Region, on which they will later be placed. Also the PRR in reference must have been stated during compile time.

In our opinion the research community needs to work towards more realistic runtime scheduling scenarios, regardless the fact that the limitations provided by the technology may reduce some of the system's performance. In simple terms a return to basics is required. Complex scheduling algorithms with online partitioning of the device are unfeasible and unimplementable at the moment. Simplest designs have more chances to be implemented and towards these kind of design our efforts will be focused.

3.6 Conclusion

In this Chapter we have presented the implementations we made of Steiger's 1D Horizon and Stuffing Techniques, as well as, the 2D Stuffing Technique using the Bazargan's *Shorter Segment* splitting heuristics. The results we gained were consistent with the ones provided by Steiger in his works. Through the development of those simple, but fundamental scheduling algorithms, we gained valuable knowledge about the development of our scheduling algorithm.

Moreover, in this Chapter we analyzed the reasons why, almost all of the current state of art algorithms cannot be implemented in a realistic FPGA device. To sum up the reasons were, the fact that developers, often made over-simplifying assumptions regarding the partial reconfiguration process, or neglected key restriction introduced by FPGAs. Next we will begin analyzing the development of our scheduling algorithm that was created with respect to the restrictions mentioned here.

Chapter 4

A Scheduling Algorithm targeting a realistic Partially Reconfigurable FPGA.

So far in this thesis we have presented many scheduling algorithms developed through the years. The main disadvantage of all those algorithms was their inability to be implemented in a realistic FPGA device. That is the main advantage of the algorithm we will present here. In this chapter we present the scheduling algorithm we developed in respect to the technology restrictions described in Chapter 3. The Chapter will be split in three sections:

- First we present the ideas used from other scheduling algorithms on the field, alongside with our ideas, which are focused on how, to make a scheduling algorithm, that will also be obedient to the technology restrictions analyzed before.
- Finally, we present and analyze our novel scheduling algorithm for hardware tasks, targeting a realistic PR FPGA.

4.1 Scheduling ideas used in our Scheduling Algorithm.

Whereas the limitations described in Chapter 3 might seem, and surely are, really restrictive, while creating an OS for reconfigurable devices a system has to be obedient in them in order to be applicable in a realistic partially reconfigurable FPGA. Furthermore the scheduling process must also obey in these restrictions.

Firstly, one of the most important restrictions a scheduler must obey to, is the pre-partitioning of the reconfigurable surface and the finality of this partitioning. Very few works have considered a pre-partitioned device, without later applying merge-split operations to utilize the remaining space, these are [3], [9] and [21]. So we had to be certain, that our scheduler would consider this restriction and pre-partition the device into several PRRs, but also would not perform any merge-split operations during runtime. In all the works mentioned here, the authors do not state clearly how the partitioning occurs or if a specific partitioning plan exists.

On the contrary, in the work done in [20] by T. Marconi et. al a clear partitioning plan is presented. Their work however, is deemed unimplementable due to the merge-split operations on these PRRs, performed during runtime. In their work, the FPGA is pre-partitioned into three different PRR sizes, small, medium and large, a partitioning plan, which we initially used. However, we found that the partitioning of the device with the use of only three different sizes, was restrictive. We reckon, it is really important to offer the scheduler several alternatives in the scheduling of tasks, for that reason we experimented with four or five different block sizes. Moreover the problem of initially positioning the PRRs on the device had to be addressed. For that matter we found useful the 2D Stuffing algorithm we have already implemented.

More specifically, with giving as inputs to the scheduling-placement algorithm many tasks of fixed sizes, the algorithm would provide a near-optimal positions for those tasks, moreover the whole process would be done offline. Eventually the placement of those tasks would provide the initial partitioning of our device. Furthermore, one might use a different scheduling-placement algorithm to obtain an even better initial partitioning.

Another important fact that is often neglected, as stated before, is that the remaining space in an occupied PRR is unusable. That is a major drawback in all of the placement managers, developed through the years and is not addressed to any of the scheduling algorithms presented in the current thesis. However, the majority of the works presented, are neglecting this restriction depending on [28], as stated in Section 3.5.

As we see in Figure 1 the PRM has been configured on PRR2. However we see that the space left in the PRR, marked with horizontal light gray lines is too much. In fact it could be large enough to accommodate another PRM. But since this PRR is in use this large amount of space is unusable and the choice of the scheduler to configure PRM1 on PRR2 should be considered bad scheduling. In order to avoid bad decisions, like the one presented here a Best Fit kind of policy was used for deciding the best PRR placement of a PRM. This placement decision can change depending on the choice made by the developer, who could chose a *Most Frequently Used* or *Best Speedup* policy, like the ones used in [8].

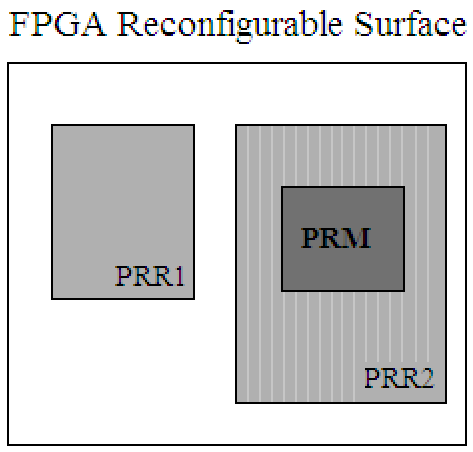


Figure 4.1: Example of bad scheduling and placement of the PRM. This could be avoided with the use of a *Best Fit* policy.

Also, we have stated in Chapter 3 of the current thesis, the absolute and necessary binding between the created bitstream of a PRM and the corresponding PRR. If only one bitstream per PRM would exist, then the scheduling process would be very straight-forward and very few alternatives would exist, for the scheduler to consider. As a result, like the researchers in [3] and [9], we consider a large "library of bitstreams", i.e. a PRM can have many implementations, that are routed to different PRRs. That way the scheduler can choose which implementation of this PRM will be placed on the device.

Even though in [3] and [9] the researchers create a large "library of bitstreams" they do not consider the advantage of creating one bitstream implementing two or more PRMs. In order to explain why, this consists an advantage in scheduling, we present the following example.

Let us consider a hardware AES fast encryptor, which takes up 342 slices [31] in a Virtex-5 device. In comparison with the total slices in a Virtex-5, i.e. 17,280 slices [32], the percentage of space that AES uses is 1.9%, which is pretty small, so a designer may choose to create a single bitstream that not only implements an AES encryptor but also a DES encryptor. It is observed a gain in space utilization and also we still have one PRR free for future use, while we have both AES and DES algorithms configured on the device, the example described above is shown in Figure 2.

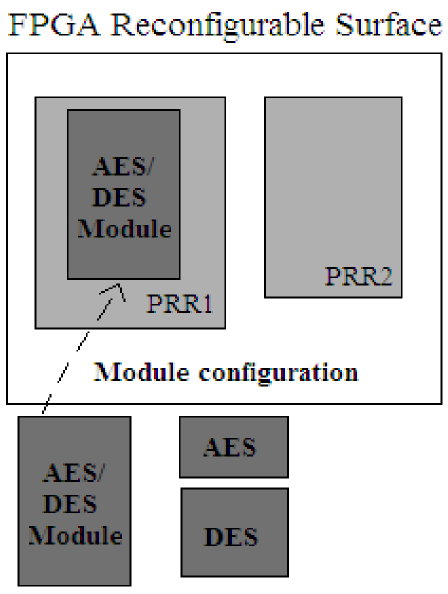


Figure 4.2: Here we can see that, despite the fact, the AES-only or DES-only fit in both PRRs, they leave a great amount of space unutilised. However, with creating a joint AES/DES bitstream, we can achieve better space utilization.

The designer must decide, which tasks will be joint and implemented in one partial bitstream. In order to simplify this decision, one might use the idea presented in [9]. In their works the authors introduce a source code annotation called OpenMP pragmas, which they use to state the level of parallelism between different tasks. Also the designer could create a task graph or tree to understand the parallelism between his tasks and then make the choice on which tasks should have a joint implementation. A great advantage of the joint bitstreams technique is that, when a designer has mutually exclusive tasks on his task set, with a careful coupling of tasks the scheduling process can become very easy.

It is a crucial feature for every scheduler to be able to plan the execution of the tasks. For example if a task cannot be placed immediately on the device the scheduler has to plan it for a latter execution, provided that the task will meet its deadline. The way we tried to integrate this feature in our scheduler, was the *reservation list* idea presented by Steiger in his work. So if a task is not able to be directly placed on the device, the scheduler will plan it for later execution.

The research done in [27] and in [28] has showed to the community that it is of little to no use trying to achieve 100% logic utilization of the resources provided. Moreover it would be more beneficial to try and achieve a good interconnect resources utilization in a design. The general rule one can obtain from these works is that the logic resources utilization may vary from 80%-90%. This is taken into account on our design as in order for a PRM to be placed on a PRR it must leave at least 20% of the PRR's logic resources unused.

Finally the FPGA technology offers certain advantages that could be used in a beneficial way, while designing a scheduler and have not yet be used by any work on the field that we are aware of. First is the possibility of a module running at a higher frequency depending on its placement on the device. In order to understand that, let us consider a device, which reconfigurable surface is partitioned in two PRRs. The designer wants to download a PRM and thus has created two partial bitstreams, each corresponding to a different PRR. Due to, the topology of the processing elements a PRR includes, or the wiring between them and/or the static part of the device, an implementation of the PRM might be able to run at a higher frequency in PRR1 than in PRR2.

In a highly advanced scheduler, that difference may prove crucial, as it would decide the completion or not of a task, provided that the task has a deadline it has to meet. However, in order to succeed this high-level functionality, the developer must know at compile time the frequencies at which a module can be executed in association with the PRR it will be placed, a feature that is not yet available. Also, another important advantage, offered by the FPGAs, is the relocation of PRMs, a feature that will be explained in details, in the following section.

4.2 Partial Reconfigurable Module Relocation on FPGA devices and its potential use in Scheduling Algorithms.

As we have stated before, the bitstream for configuring one PRR is tightly bound to the physical location of that region and cannot be used directly to reconfigure any other portion of the chip. In many cases though, it is possible to modify an existing bit-stream and adapt it to a different physical location. This process is termed bitstream relocation and can be performed statically by tools operating on the designer’s workstation, or dynamically by the agent that loads the bit-stream on chip. The key element is that a portion of the chip is reconfigured at runtime, without interfering with the operation of the rest of the chip.

So far, Xilinx has presented two forms of relocation. The first one, was the internal relocation. In this form the designer with the use of specialized software, could create a partial bitstream and after its placing on the device, could change the frame address of the bitstream and as a result change the “physical” placement of the module on the device.

The second, modern to the previous, states that when invoking relocation the device searches the host computer's memory or the internal FPGA memory for the bitstream that corresponds to the module, that is about to be relocated, and the PRR that this module will be relocated into. That not only increases the time need for relocation, but also defeats the true purpose of relocation, which is relocating a module to a PRR with no need of a previously generated bitstream for that combination of module-PRR. In this thesis when referring to relocation, we mean the first form of internal relocation with no use of the extra partial bitstream. A simple example of relocation is presented in Figure 3.

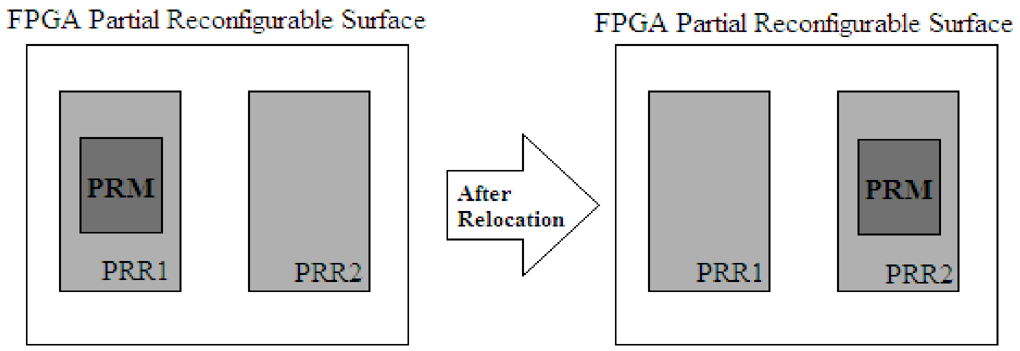

FPGA Partial Reconfigurable Surface
PRM
PRR1
PRR2
After Relocation
FPGA Partial Reconfigurable Surface
PRR1
PRM
PRR2

Figure 4.3: The *relocation* process. First the designer defines the two PRRs at compile time. Then at runtime the partial reconfigurable module "moves" from the one PRR to the other. The "move" does not require a seperate partial bitstream for the PRM bound to PRR2. Also the PRM needs to restart its execution as *relocation* is not a context switch type of operation.

Considering the above example the relocation process can become a powerful tool, with its limitations of course, in the scheduling process. An example of its use is shown at Figure 4. Let us consider a situation, where two PRMs are being scheduled with two PRRs to be configured into. One of the modules is already configured in the device (PRM1 to the PRR2) and a second one (PRM2) is waiting to be configured but does not fit in the free PRR1 or in another scenario a bitstream binding PRM2 to PRR1 might not exist. The scheduling algorithm would perform relocation of the first module, if possible, and configuration of the second one to the now empty PRR. We would like to note that these kinds of operations or this use of relocation are not supported, nor mentioned at any state of art scheduling algorithms that we know of or even in simple and complete runtime systems as in [3] and [9]. In order to perform this operation there is no need of a partial bitstream binding PRM 1 to PRR1.

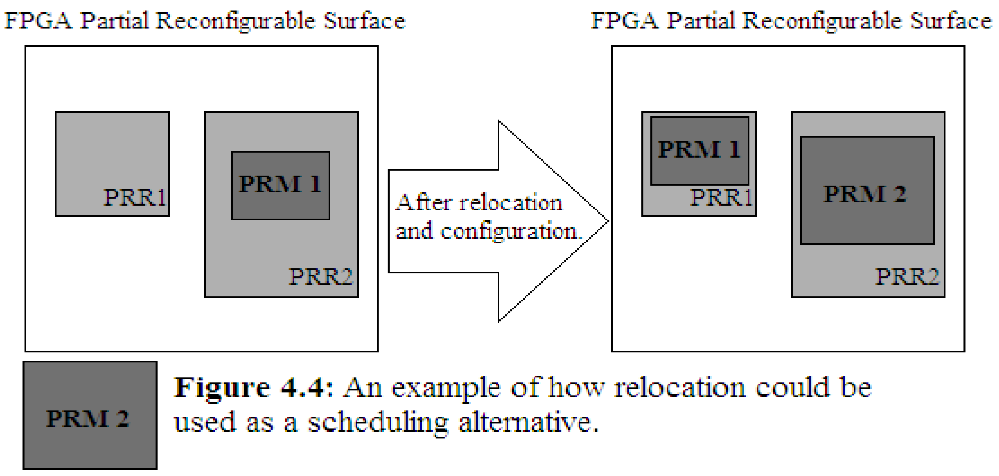

FPGA Partial Reconfigurable Surface
PRR1
PRM 1
PRR2
After relocation and configuration.
FPGA Partial Reconfigurable Surface
PRM 1
PRR1
PRM 2
PRR2
PRM 2

Figure 4.4: An example of how relocation could be used as a scheduling alternative.

We have shown that relocation, if applied correctly, can become a viable alternative for scheduling a PRM on a FPGA even though initially the scheduler could not come up with an appropriate placement for that module. In our scheduling algorithm we use the *Relocation Alternative* exactly with the way it was described here. A more detailed execution flow of the *Relocation Alternative* will be presented later.

4.3 The Partial Reconfigurable Module true FPGA size issue.

Another big mistake made on all of the existing scheduling algorithms and research regarding the creation of an Embedded Operating System is the true size of a PRM when that PRM is placed on the FPGA device. So far researchers assumed that the size of a PRM could be counted by the amount of reconfigurable units it uses when placed on the device. However that is not the case in a highly realistic scenario and in order to understand that, a deeper understanding of the FPGA technology is required.

The generic structure of a Virtex-II Pro FPGA is the one showed in Figure 5. The smallest addressable segment of the device's configuration memory space are called frames. A frame is one bit wide and stretches from the top edge to the bottom of the device, a frame does not directly map to any single piece of hardware; rather, they configure a narrow vertical slice of many and different physical resources. In order to perform reconfiguration of the device, one has to produce a partial bitstream that makes changes to one or more frames.

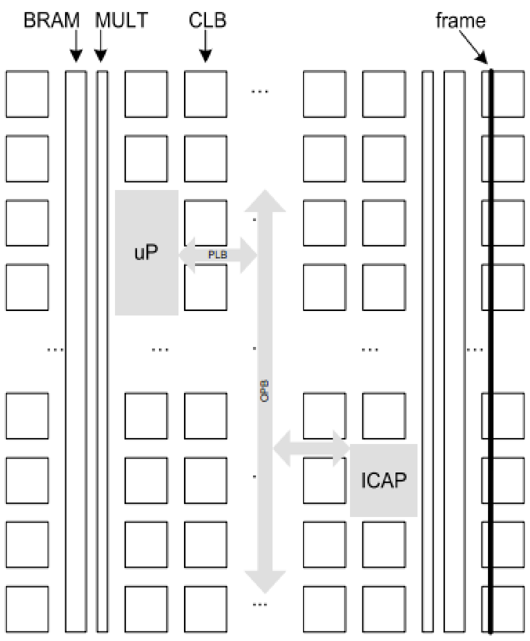


Figure 4.5: The Virtex II-Pro FPGA resources and configuration frames.

The important fact of all the above is that when defining a PRR the designer first must be absolutely sure that the partitioning of the device leads to the same amount of logic on each PRR. For example a designer could split the device shown in Figure 6 in two PRRs. However the first one contains a switch, whereas the second does not. Also even when the logic in a PRR is the same, the usage percentage of the logic can be different. In Figure 7 we see a design that uses 4 "number 1" resources and 4 "number 2" resources from the CLBs contained in a PRR. However in PRR1 the design uses 4 out of 5 CLBs and in PRR2 it uses 5 out of 5 due to a difference in routing of the resources.

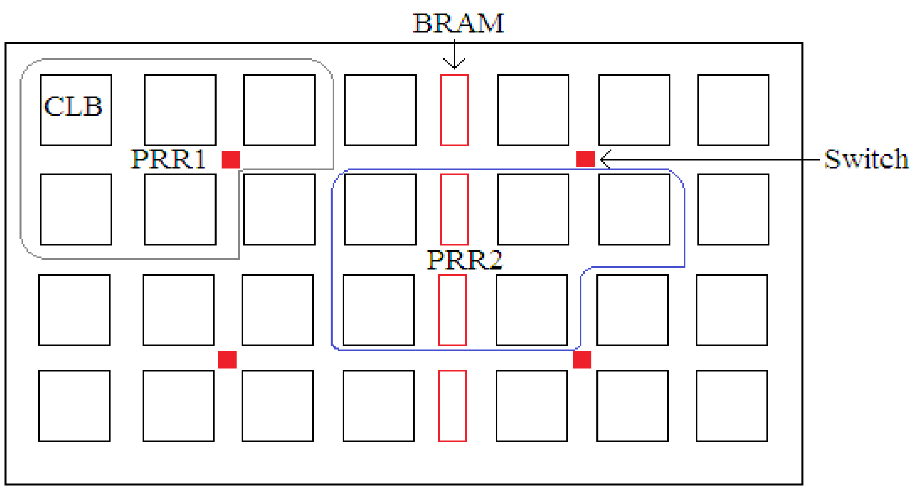


Figure 4.6: In the device paritioning process the designer must be very accurate as to what logic resources are contain in a PRR. Here eventhough the same amount of CLBs are contained in PRR1 and in PRR2, PRR2 also contains 2 BRAMs, whereas PRR1 contains a switch.

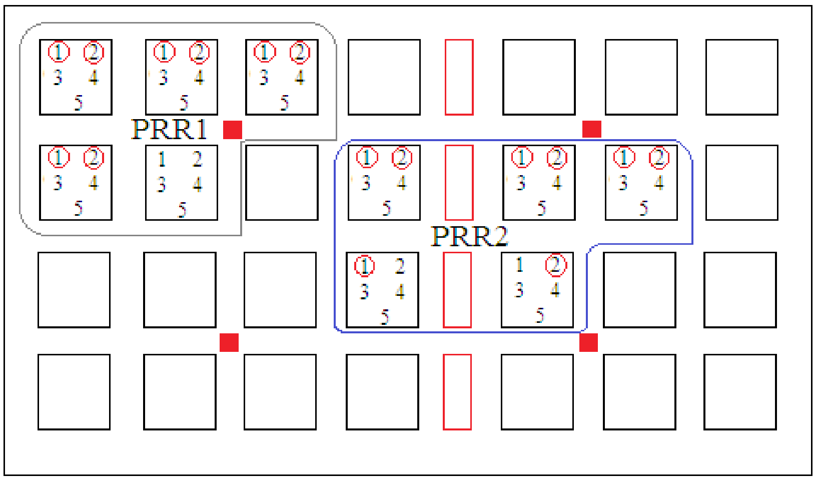


Figure 4.7: In this example PRR1 and PRR2 contain the same task in terms of functionality and logic requirements. The task needs 4 "number 1" and "number 2" resources. However due to different routing we see that in PRR2 5 CLBs are used instead of 4 in PRR1. The used resources are indicated with red circles.

This is a very thorough and detailed analysis of the FPGA technology and many might argue that this analysis and this level of detailed understanding of the FPGA is unnecessary. However we think that, in big designs this difference between the true PRM size and the one produced by the offline simulation tools could be crucial as to whether a PRM could be placed on a certain PRR. Also considering the above example we have an increase of CLB usage on the second PRR of 20%. If we add to this the rule presented in [27] and [28] then despite the fact that the analysis is very deep it is important to be done, in order to have a truly realistic system in terms of technology.

So based on this analysis, in our scheduler we create several partial bitstreams for each PRM, binding them to different PRRs. However despite having a fixed FPGA task size depending on the task's functionality, each bitstream is characterized by a different FPGA size.

4.4 The Scheduler Analysis.

After having analyzed the ideas we will use and implement in our scheduling algorithm we now present the scheduler we created. First we will sum up the ideas presented thus far in our work and will find use in our scheduler.

- *Reservation list,* initially presented by Steiger in [10].
- *Multiple Bitstreams per Task,* initially presented in [3] but also used in [9], the

only complete runtime systems that we have studied.

- *Initial device partitioning,* presented in [3], [9] and [21].
- *The Golden Rule,* this “rule” presented in [27] and [28] has not be taken in account by any of the researchers, who tried to create a complete runtime system in the best of our knowledge.
- *The PRM Size Issue,* is an idea used by us following a detailed analysis of the internal FPGA structure and the understanding of PRM implementation and routing.
- *The Relocation Alternative.* A novel technique that allows relocation to be used as a scheduling alternative.
- *Joint Bitstreams.* A novel technique that gives priority in downloading bitstreams, that implement one or more tasks in one bitstream file.

In order to efficiently implement our scheduler, several data structures for representation of available information, were used. More specifically, a list structure was used for representing the PRRs, the tasks, the mappings for each task and the reservations made from the scheduler. The three structures and their fields are mildly influenced by [9].

```
struct task_list{
   int TaskNum;
   int Task_exTime;
   int Task_arrTime;
   int Task_recTime;
   int Task_endTime;
   int Task_state;            //1=running, 0=not_running, 2=completed
   int Executed_PRR_num;      //0=if the task is not running in any PRR, x=the PRR number
   int implem_num;            //0=if no implementation is currently running, x=the implem_num
   int completion_time;
   struct task_list* next;
}task_list;
```

```
struct PRR_list{
   int PRR_num;
   int PRR_state;   // 1=in use, 0=not in use
   int reserved;    //1=reserved, 0=not reserved
   int PRR_releaseTime;
   int width;
   int height;
   int x_placement;
   int y_placement;
   int currentTask_num;
   int currentTask_implem;
   struct PRR_list* next;
}PRR_list;
```

```
struct mappings_list{
         int Tasknum;
         int implem_num;
         int PRR_num;
         int implem_width;
         int implem_height;
         int numOfDownloads;
         struct mappings_list* next;
}mappings_list;
```

```
struct reserv{
   int PRR_num;
   int PRR_time;
   int task_num;
   int task_implem;
   struct reserv* next;
}reserv;
```

Above we see the four main structures we used to implement our algorithm. These are, the task list, the PRR list, the mappings list and the reservation list.

Our scheduler begins its execution by accepting as input the initial partitioning of our device, the tasks to be scheduled and the different mappings of these tasks, i.e. different bitstreams corresponding to different implementations of one task.

More specifically the algorithm needs the execution, reconfiguration and deadline times of the task. Also the PRRs size is needed in order to use the Best Fit policy. Finally information about the bitstreams is needed, specifically the binding each bitstream creates between the task and the PRR and the size used by the bitstream to implement the task on the device.

At each point in time, our scheduler checks if there is a newly arrived task to be scheduled onto the device, then informs the user as to if this task was directly placed onto the device or a different scheduling alternative was chosen. After that, the scheduler checks if a task has completed its execution on the device and finally the scheduler checks, if there are reserved tasks that must start their execution on the device.

Algorithm 1: The pseudocode for the scheduler we created

```
/*Partition the device*/
PRR_list = initializeDevice(PRR_list);

/*Bitstream Creation*/
mapping_list = createBitstream(mapping_list);

/*Task input*/
task_list = initializeTasks(task_list);

/*Scheduling of newly arrived tasks*/
if (t == arrival_time) then
   schedule(task_list, PRR_list, mapping_list, reservation_list);
   Notify user for scheduling outcome

/*Completion of tasks*/
if (t == ending time of task) then
   Mark task as complete and PRR as empty.

/*Beginning of execution of reserved tasks*/
if (t == starting time of task) then
   Begin execution of task, mark PRR as occupied
```

One of the main functions of our program is the scheduling function. Inside that, we try to find a way to either place the newly arrived task on the device or schedule it for a later start. In order to schedule the task on the reconfigurable device, the scheduler first creates a list of the available mappings for the newly arrived task. According to the PRRs this mappings have been created for, a list of PRRs is also created. In essence this list contains the PRRs, the task has mappings for. If this list contains more than one PRR a Best Fit policy is used to decide, which PRR the task will be placed on. Inside the function that implements the *Best Fit* policy it is also checked whether or not the PRR is free at that moment. The *Best Fit* policy will place the newly arrived task on the PRR, which will produce the less unused area.

After applying the *Best Fit* function if no suitable PRR could be found the scheduler will perform the *Relocation Alternative.* With that method the scheduler tries to relocate a previously placed task on another PRR so that the newly arrived task can be placed also on the device. If this proves to be unsuccessful the scheduler tries to make a reservation for the newly arrived task, in order to be executed on the device at a later time. Finally if none of the above works the scheduler informs the user that the task's software implementation should be executed. The execution flow of the process described above is shown in Figure 8.

An important function in our scheduler is the relocation function, which is used as a first alternative, if no immediate placement for the task is found. Specifically the relocation alternative is performed when none of the PRRs, for which the task has mappings for is free. The relocation alternative first considers the available mappings for the newly arrived task and chooses the first occupied PRR, for which a mapping already exists. Afterwards the scheduler tries to find for the task currently occupying the PRR a new placement, considering again a Best Fit policy. If no new placement can be found the scheduler considers the next occupied PRR and tries to find a new placement for the task currently occupying this. If all the mappings for the newly arrived task are considered and no relocation could be done for any of them the relocation alternative is considered as unsuccessful and the scheduler tries to perform reservation of the newly arrived task.

The *Relocation Alternative* execution flow is shown in Figure 9. As stated before if the *Relocation Alternative* cannot find a solution to place immediately the newly arrived task on the device, the scheduler will try to make a reservation for it in a currently occupied PRR, in order to start execution once the PRR is free. A task can be reserved in a PRR that is not currently reserved for another task, also a mapping corresponding to that PRR must exist. Finally the PRR that will be chosen for a reservation is that with the shortest release time, without considering a best fit policy, that way the task will start its execution, as soon as possible.

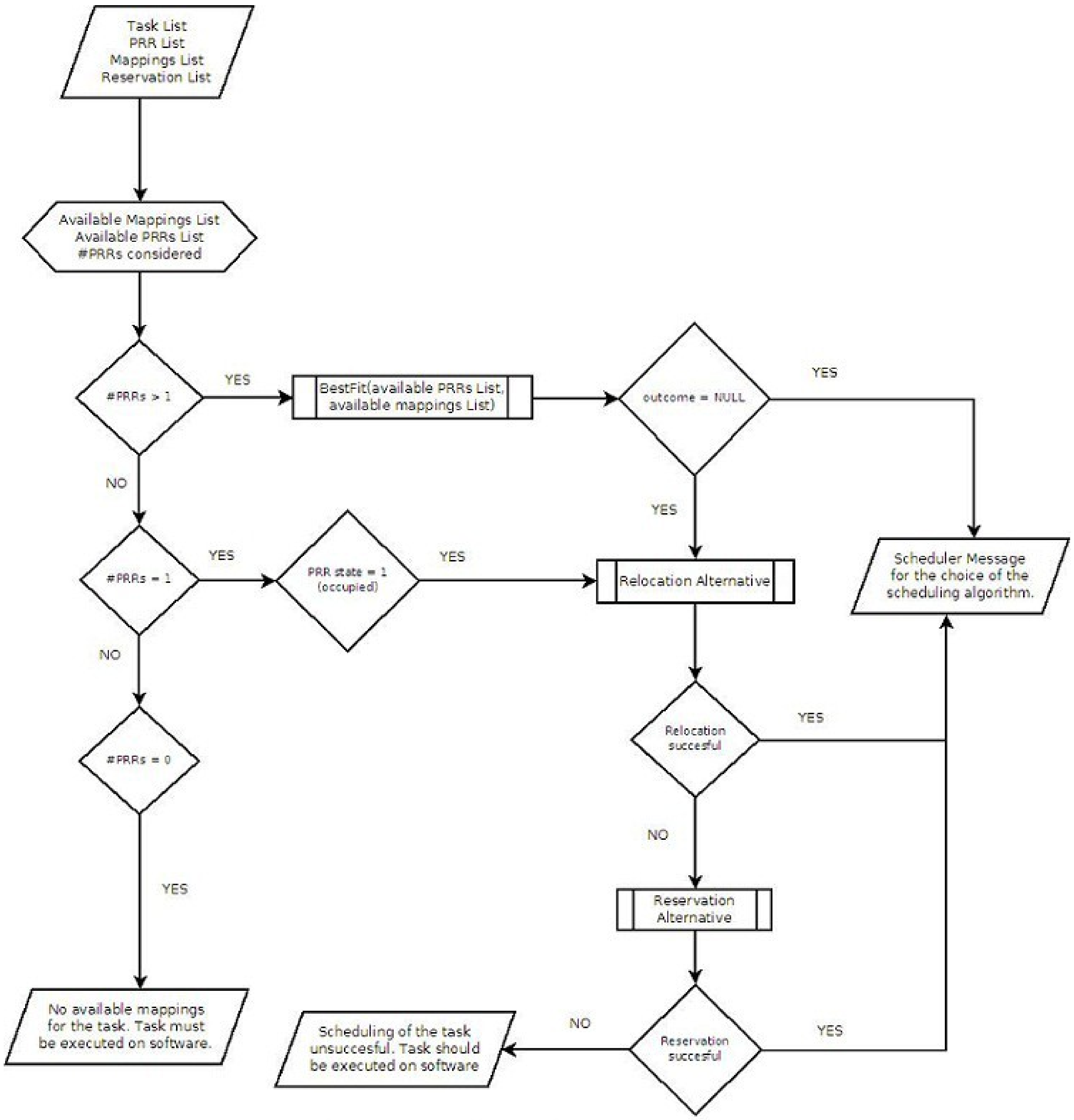


Figure 4.8: The execution flow of the scheduler we created.

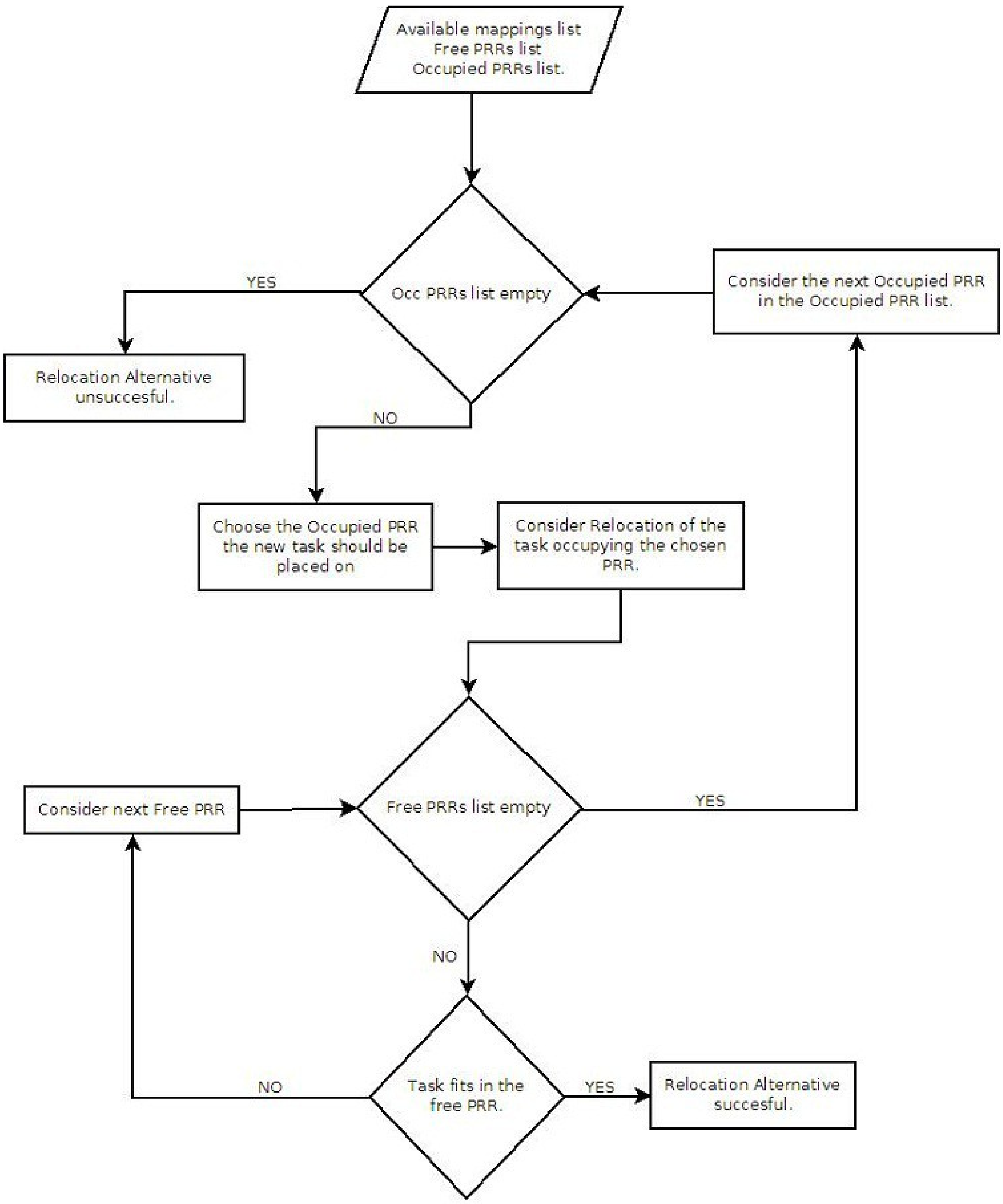

Available mappings list
Free PRRs list
Occupied PRRs list.
Occ PRRs list empty
YES
Relocation Alternative unsuccesful.
NO
Consider the next Occupied PRR in the Occupied PRR list.
Choose the Occupied PRR the new task should be placed on
Consider Relocation of the task occupying the chosen PRR.
Consider next Free PRR
Free PRRs list empty
YES
NO
Task fits in the free PRR.
NO
YES
Relocation Alternative succesful.

Figure 4.9: The execution flow of the *Relocation Alternative.*

Of course for all the operations done regarding the task scheduling and placement on the device, the algorithm considers the deadline, set for each task. If an alternative, either relocation or reservation is available but the deadline is not met then the task is either rejected or executed on software.

4.5 Conclusion

In this Chapter we presented a hardware task scheduling algorithms that targets a realistic partially reconfigurable FPGA device. The development of the scheduler was done, with respect to the technology restrictions analyzed on Chapter 3 of this thesis. In the implementation process several ideas were used. Some were derived from works presented earlier in our thesis, mainly [3], [9], [10], [21], [27] and [28], while others were novel ideas, of ours, regarding task relocation and its use in scheduling, the joint bitstreams and several restrictions that have never been taken in account, in the best of our knowledge, even in more complete works, like the ones done in [3] and [9].

Chapter 5

Evaluating our Scheduling Algorithm.

After the presentation we made, of our novel scheduler in Chapter 4, here we will attempt a thorough evaluation of our design. Following the above-mentioned flows and guidelines a C-language implementation of our scheduler was made. We have built a discrete-time simulation framework in C to evaluate the proposed algorithm. The framework was compiled and run under Windows 7 operating system on an Intel Core i3 CPU @ 3.10GHz PC. The evaluations were made, with simulating a FPGA device and synthetic tasks. as well as, real application tasks too.

5.1 Evaluations with our task sets.

One of the most important aspects in evaluating our algorithm is to successfully determine the inputs our system needs in order to work properly. If we refer to Chapter 4 and the representation of the structures that are used in our algorithm, we can derive the inputs needed. First it is necessary that the designer provides the data of the pre-partitioned FPGA. Mainly the algorithm needs the number of the PRRs that were made, their placement on the device as a pair of [x, y] coordinates and the PRR's size, as it is defined by its placement on the FPGA, e.g. a PRR might have a placement [1, 1] and its size be 4×5 reconfigurable units.
Second the user needs to provide certain data, regarding the tasks that will be scheduled by the device. At this state of our algorithm, those data need to be defined before executing the algorithm. However it would be possible in future versions that this data would be given during runtime of the algorithm, mainly the arriving time of the tasks.

The data needed for each task are, its arrival, execution and reconfiguration times along with its deadline.

An important advantage of our algorithm is the library of bitstreams that should be created from the user in order to improve the quality of scheduling and increase the acceptance rates. So for each task the user must provide the algorithm with the available mappings for this task. Each mapping must come with information regarding, which task(s) it implements, which PRR is bound to and the size it occupies when placed on the PRR.

Once the designer has supplied the algorithm with these information, then the scheduling process can begin. First we feed our algorithm with a simple task set we created and a partitioning plan. The task set consists of four tasks and 1 or 2 implementations per task, the device is partitioned in two PRRs. The main purpose of this task set is to show a simple scheduling process, using both the *Relocation Alternative* and the *Reservation Alternative.* In the tables that follow we present the data regarding, the tasks, the mappings and the PRRs, which are the inputs of our scheduler.

Task Table

Task Number	**T1**	**T2**	**T3**	**T4**
Execution time	12	4	7	3
Configuration time	3	1	2	5
Arrival time	1	2	3	3
Deadline	20	10	20	25

Mappings Table

Task number	**T1**	**T1**	**T2**	**T3**	**T3**	**T4**	**T4**
Implementation number	1	2	1	1	2	1	2
PRR number	1	2	1	1	2	1	2
Implementation width	2	3	2	3	2	3	3
Implementation height	2	2	2	2	3	3	4

PRRs Table

PRR number	**1**	**2**
PRR width	3	4
PRR height	3	4
PRR x placement	1	5
PRR y placement	1	1

Considering the above we can produce a graphical representation of our device, which can be seen in Figure 10.

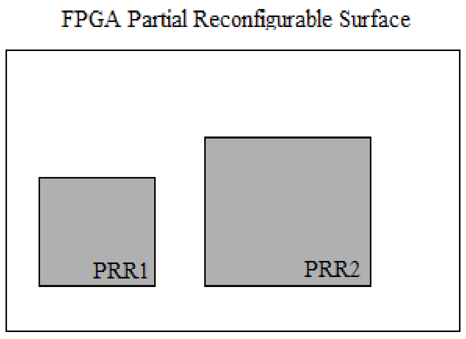
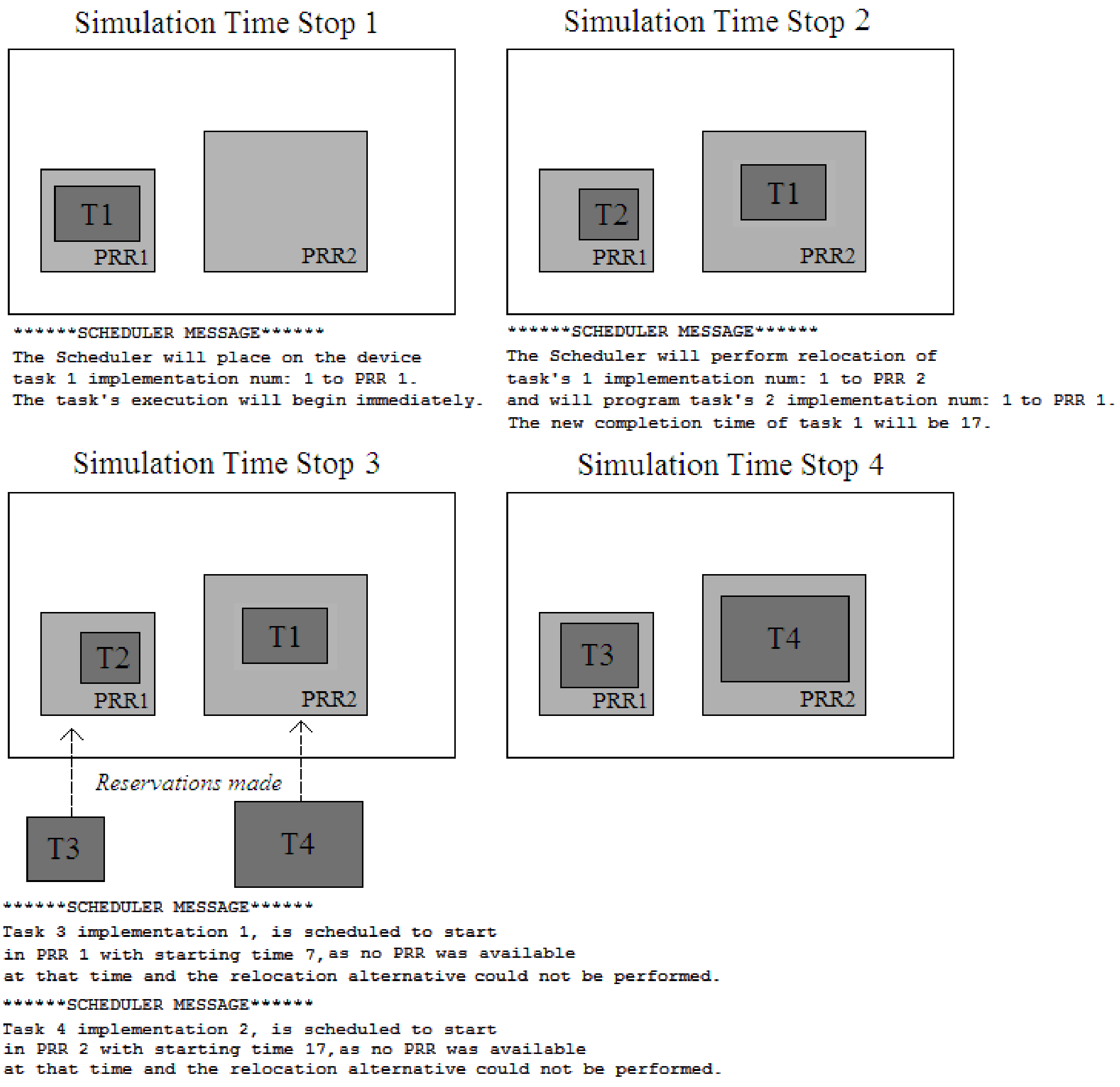


Figure 5.1 : Graphical representation of our device.



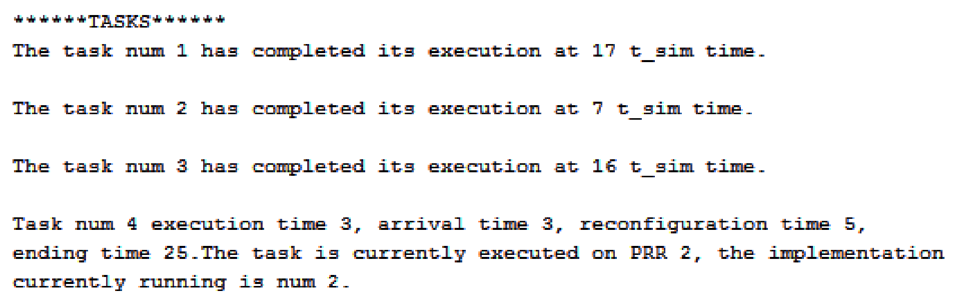
```
******SCHEDULER MESSAGE******
The Scheduler will place on the device
task 1 implementation num: 1 to PRR 1.
The task's execution will begin immediately.
```

```
******SCHEDULER MESSAGE******
The Scheduler will perform relocation of
task's 1 implementation num: 1 to PRR 2
and will program task's 2 implementation num: 1 to PRR 1.
The new completion time of task 1 will be 17.
```

```
******SCHEDULER MESSAGE******
Task 3 implementation 1, is scheduled to start
in PRR 1 with starting time 7,as no PRR was available
at that time and the relocation alternative could not be performed.
******SCHEDULER MESSAGE******
Task 4 implementation 2, is scheduled to start
in PRR 2 with starting time 17,as no PRR was available
at that time and the relocation alternative could not be performed.
```

Figure 5.2 : A graphical representation of the device state at each major Simulation Time Stop, accompanied with the Scheduler Message for each scheduling decision.

After the specification of the scheduler inputs we begin the execution of our program. In order to make it more user friendly we have several simulation time stops, where the program informs the user for the current device representation. In Simulation time Stop 1 we see the configuration of Task 1 on PRR 1, next in Simulation Time Stop 2 we see the application of the *Relocation Alternative* for the configuration of Task 2, in Simulation Time Stop 3 we see the reservations the scheduler made for Tasks 3 and 4 and finally we see the configuration of these tasks on the device. A graphical representation of the process described above is shown in Figure 11. The final message from the scheduler is shown in Figure 12.



```
******TASKS******
The task num 1 has completed its execution at 17 t_sim time.

The task num 2 has completed its execution at 7 t_sim time.

The task num 3 has completed its execution at 16 t_sim time.

Task num 4 execution time 3, arrival time 3, reconfiguration time 5,
ending time 25.The task is currently executed on PRR 2, the implementation
currently running is num 2.
```

In order to check the deadlines parameter on our scheduler we have to perform two kinds of new checks. First we check the deadline parameter regarding the reservation alternative and then for the relocation. For the first check the only thing we have to do is put the deadline for task 4 earlier that t_sim=25. Let us assume we put the deadline at t_sim=12. The now changed task table is shown bellow, with bold we show the change made.

Task Table

Task Number	**T1**	**T2**	**T3**	**T4**
Execution time	12	4	7	3
Configuration time	3	1	2	5
Arrival time	1	2	3	3
Deadline	20	10	20	**12**

After successfully scheduling the first three tasks, the message we receive from the scheduler during the planning of task 4 is the following.

```
******SCHEDULER MESSAGE******
The scheduler could not find appropriate placement for task 4. Also the
relocation alternative haven't found a viable solution nor the task
could be reserved for execution at a later time. The task's software
implementation should be executed.
```

Now we have to check the deadline parameter regarding the relocation alternative. In order to have the relocation alternative rejected, the deadline of the task to be relocated will have to be earlier than the new completion time of the task when that is moved to a new PRR and restarts its execution. So, to achieve that we change the deadline for task 1 from 20 to 16. Without that change in Figure 11 we have seen that after the relocation the new completion time of task 1 would be 17. However applying the change mentioned task 1 is forbidden to be relocated as the new completion time exceeds its deadline. In order though to show the successful reservation of task 2 we have to change its short deadline from 10 to 25. The new task table is shown bellow, with bold we show the change made.

Task Table

Task Number	**T1**	**T2**	**T3**	**T4**
Execution time	12	4	7	3
Configuration time	3	1	2	5
Arrival time	1	2	3	3
Deadline	**16**	**25**	20	25

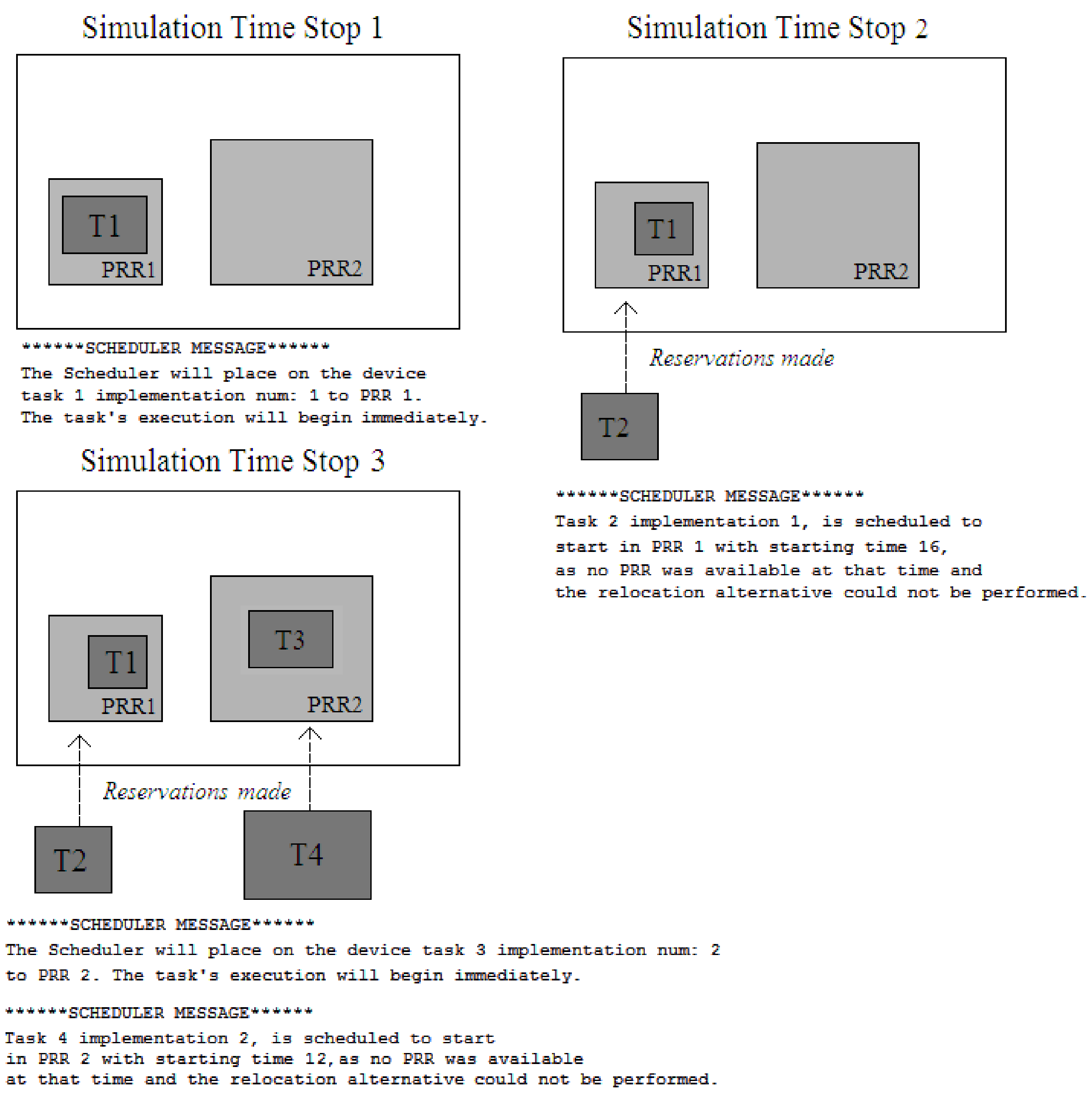
The result is that the scheduler now tries and successfully manages to reserve task 2 for later execution on PRR 1. That means that the whole scheduling process of task 3 changes accordingly to that fact. Now that a PRR is free task 3 can begin its execution immediately. Next the scheduler reserves task 4 for later execution and succeeds, after it secures that its deadline will be met. A graphical representation of the process described above is shown in Figure 13. The final message of the scheduler, which shows the completion times for each task is shown bellow.

```
******TASKS******
The task num 1 has completed its execution at 16 t_sim time.

Task num 2 execution time 4, arrival time 2, reconfiguration time 1,
ending time 21.The task is currently executed on PRR 1, the implementation
currently running is num 1.

The task num 3 has completed its execution at 12 t_sim time.

Task num 4 execution time 3, arrival time 3, reconfiguration time 5,
ending time 20.The task is currently executed on PRR 2, the implementation
currently running is num 2.
```



```
******SCHEDULER MESSAGE******
The Scheduler will place on the device
task 1 implementation num: 1 to PRR 1.
The task's execution will begin immediately.
```

```
******SCHEDULER MESSAGE******
Task 2 implementation 1, is scheduled to
start in PRR 1 with starting time 16,
as no PRR was available at that time and
the relocation alternative could not be performed.
```

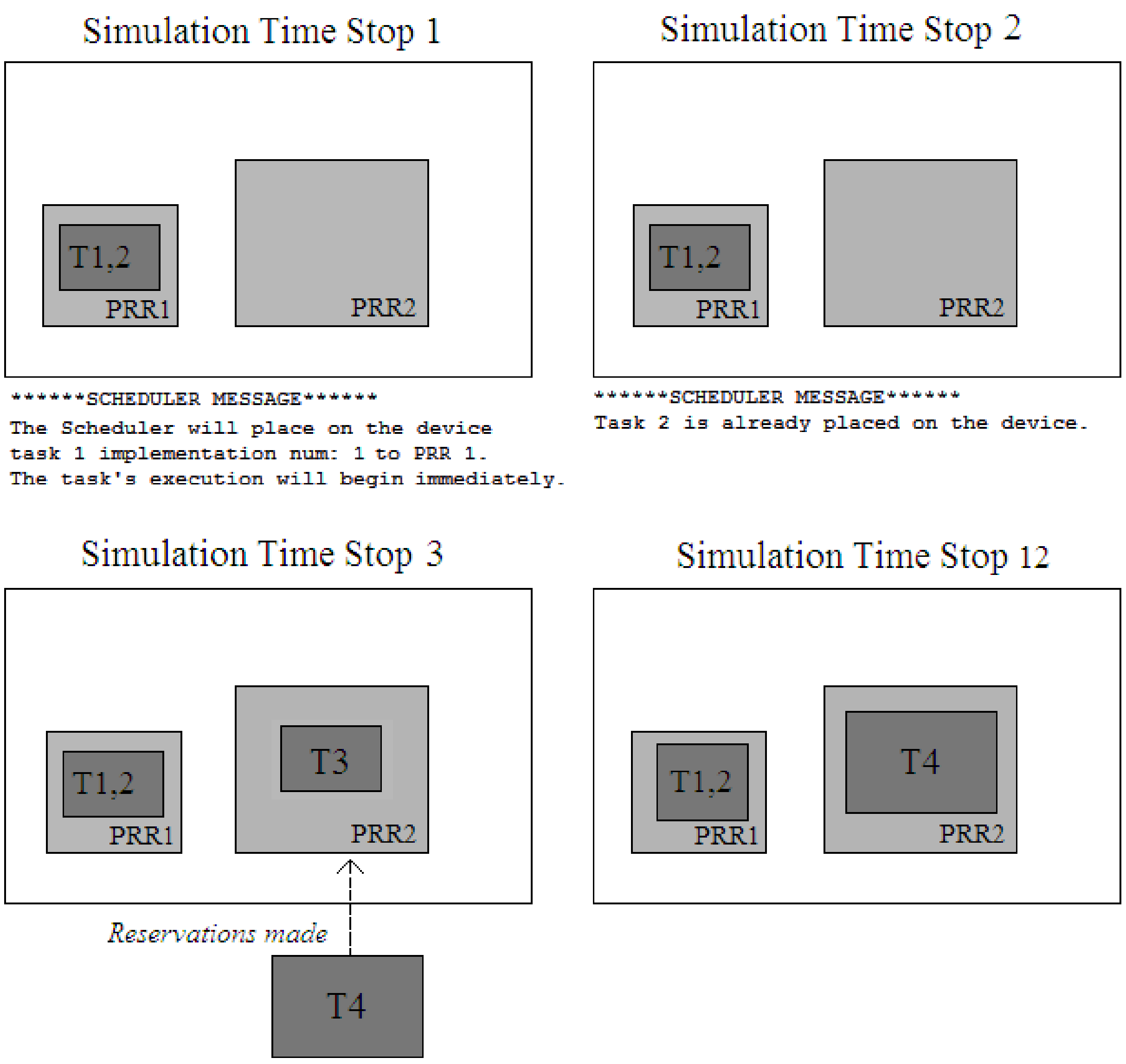
```
******SCHEDULER MESSAGE******
The Scheduler will place on the device task 3 implementation num: 2
to PRR 2. The task's execution will begin immediately.
```

```
******SCHEDULER MESSAGE******
Task 4 implementation 2, is scheduled to start
in PRR 2 with starting time 12, as no PRR was available
at that time and the relocation alternative could not be performed.
```

Figure 5.3: Here we present the graphical representation and the messages shown by the scheduler when the first relocation of task 1 in order to immediately place task 2 is rejected due to the deadline of task 1.

The next step in evaluating our design is to check the joint bitstreams. As stated before a great advantage in the scheduling process could be the creation of bitstreams that could not only implement one task but two. With that premise, when such a bitstream is placed on the device the scheduling of the accompanying task is useless regardless the time this task would normally arrive, as it already place on the device.

One of the tweaks, we have inserted in our algorithm, is to always give priority in placing a joint bitstream on the device. It is our belief that it is more important to be able to place one more task on the device, than having better space utilization via the *Best Fit* function. In order to check our algorithm we provide a joint bitstream. The joint bitstream implements both task 1 and task 2 and is bind with PRR1. So when the scheduler places that bitstream on the device task 2 will also be placed on the device. A detailed representation of the device in that case is shown in Figure 14.



```
******SCHEDULER MESSAGE******
The Scheduler will place on the device
task 1 implementation num: 1 to PRR 1.
The task's execution will begin immediately.
```

```
******SCHEDULER MESSAGE******
Task 2 is already placed on the device.
```

```
******SCHEDULER MESSAGE******
The Scheduler will place on the device
task 3 implementation num: 2 to PRR 2.
The task's execution will begin immediately.
```

```
******SCHEDULER MESSAGE******
Task 4 implementation 2, is scheduled to start
in PRR 2 with starting time 12,as no PRR was available
at that time and the relocation alternative could not be performed.
```

Figure 5.4: Here we see that because of the joint bitstream we created implementing task 1 and task 2 task 2 is already place on the device at t_sim=2 and PRR2 is empty.

5.2 Comparing our scheduler with other works.

Apart from applying our task sets and evaluating our scheduler with tasks sets we created, we continued with tests and evaluations of our algorithms with task sets taken from other works of the field.

However there were many difficulties in the course of achieving this. First and foremost our scheduling algorithm is the only one that considers almost every technology restriction in the best of our knowledge. Also our algorithm needed a pre-partitioned device in order to work, so every work done by T. Marconi was almost impossible to compare to.

First, because the way our scheduler works the pre-partitioning of the device and the bitstreams creation process are part of the scheduling process. Also the lack of technology restrictions in Marconi's work gave outstanding results to the acceptance rates achieved by the algorithm. However, those results are ideal and do not correspond to a realistic FPGA device.

The only work on the field that was so careful with realizing the technology restrictions and with the system prototype we used was [9]. In their paper Santambrogio et al. test their system by simulating two algorithms by the imaging process field. More specifically they consider a Canny Edge Detection Filter and a Motion Detection Filter. The Canny Edge Detection Filter, performs a gray scale conversion, a noise removal filter, the edge detection and finally it applies a threshold. The Motion Detection Filter performs the same actions, except the edge detection one that is substituted by a motion detection filter.

After setting up their framework the researchers give specific information and data regarding the execution times of each of the tasks and their size. The device considered in the experiments of the Italians is a Virtex 5-LXT110 with 17.280 slices of reconfigurable area. Here we present the same data on the following table:

Task	Execution time [clock cycles]	Reconfiguration time [clock cycles]	Area [slices]
Gray Scale	88.713	390.745	5.120
Noise remove	90.434	390.745	3.613
Edge Detection	78.234	390.745	2.723
Motion Detection	52.348	390.745	3.354
Threshold	47.688	390.745	3.224

Although in our set we have made different assumptions regarding the task's execution time and area. The execution and reconfiguration time are measured in microseconds and the area is measured in CLB “tiles”. The conversion of cycles to microseconds was easy because the clock frequency of the experiments was documented at 100MHz. In order to convert the slices to CLB, we need to make some assumptions. First for each task we find the usage percentage of the total reconfigurable area. Then we apply that to the total amount of CLBs on our framework and find an approximation of the CLBs each task uses.. In order to make the calculations and the simulations simpler we consider an 18×6 device with a total of 108 CLBs.

The table showing the tasks information with our metrics is shown bellow.

Task	Execution time [μs]	Reconfiguration time [μs]	Area [CLBs]
Gray Scale	887	3.907	32
Noise remove	904	3.907	23
Edge Detection	782	3.907	17
Motion Detection	523	3.907	21
Threshold	476	3.907	20

After converting the task information to the ones we use, we continued with studying the pre-partitioning of the device. Unfortunately no specific data was provided in [9], apart from the fact that the experiments were held with varying PRR numbers from 1 to 6. However if we assume that the all PRRs must cover the area of the largest task, then there is no way to partition the device in more than 3 PRRs of 36 CLBs each. Despite that we initialized our device first with 1 PRR of 42 CLBs size, then with 2 PRRs one 42 and one 36 CLBs size and finally with 3 PRRs of 42, 36 and 30 CLBs size each. Also when making experiments with more than one PRRs, the bitstreams we created, bound every task to almost every PRR, with the exception of the Gray Scale conversion.

Giving this setup as an input to our scheduler and after disabling the deadlines parameter (the Italians did not consider task deadlines) we saw that in all the cases the scheduler managed to successfully place all the tasks on the device, while applying the *Best Fit* function for achieving better space utilization. For each test we measured the times a Canny Edge Detection followed by a Motion Detection Filter would be performed per second. The results we obtained were the following:

- For 1 PRR; the execution time of one CED + MDF sequence, was 37.696 microseconds, i.e. 26 times per second.
- For 2 PRRs; the execution time of one CED + MDF sequence, was 18.878 microseconds, i.e. 52 times per second.
- For 3 PRRs; the execution time of one CED + MDF sequence, was 14.072 microseconds, i.e. 71 times per second.

However the results the Italians documented in their paper showed the throughput in each of the cases with providing the frames per second achieved. The image they had as an input was a 640×480 BMP image. Also the Italians did a partitioning of each task mentioned before into four subtasks, however it is not clear whether each subtask was of the same size as the "parent" task or a fraction of it. Also several other problems arise with the division of tasks. For example, in order for a task to be complete, all the subtasks have to be complete. Nonetheless, in a case where two task complete first their execution and the other two complete theirs much later, how does the system store the intermediate results?

We have come to the conclusion that even though it is not mentioned in their work, the researchers do not built a real system, but they perform simulations of their system after first obtaining realistic data regarding the execution times and the sizes of their tasks. The comparison of our scheduling algorithm with their task sets, confirmed its proper execution. Finally we would like to add that the advantages of our scheduling algorithms are not visible with this task set. First the tasks do not have deadlines and second the advantages offered by the *Relocation Alternative* are better shown in cases of different PRR sizes and fewer bitstreams per task. More thorough examinations with real life tasks and applications are scheduled to be done in the near future.

Chapter 6

Conclusion and Future Work.

For the purposes of this thesis, we first studied excessively the most notable works done so far in the field of hardware task scheduling and Embedded Operating Systems targeting a FPGA device. Following the studies made we evaluated every one of these scheduling algorithms, with respect to the technology restrictions introduced by the FPGA devices and the Partial Reconfiguration process, in terms of their ability to be implemented in a realistic FPGA device.

Then we continued with the development of a novel scheduling algorithm targeting a realistic FPGA device. In this scheduling algorithms we included ideas taken from previous work on the field, but also some novel ideas of ours that have not been used so far, in the best of our knowledge, such us numerous technology restrictions and Partial Module Relocation as a scheduling alternative.

Afterward, we evaluated our algorithm by creating numerous synthetic task sets and gathering the scheduling results. These results proved to be quite promising. Despite the fact that, they were slightly worse compared to the ones other researchers have published, their high advantage is that these results refer to a realistic FPGA device with no simplifying assumptions.

As part of future work, we first of all, plan to create a complete Runtime System that will use this Scheduling Algorithm, in order to measure execution times and overheads, introduced by the scheduling process. Also one might explore different placing policies besides the *Best Fit,* that has been used here. Some of them could be, *Most Frequently Used, Power-constrained scheduling policies, Best-Fit Not Ready, First Fit* and many more.

We have mentioned this before, but it is important for the community to understand that the scheduling process does not begin and end with the execution of a scheduling algorithm. In fact the scheduling process begins with the bitstreams creation and the pre-partitioning of the device, as well as with the task graph creation and analysis, and ends with the efficient and quality placement of the tasks on the FPGA device.

As a result, a significant amount of research could be spent on methods and heuristic that will help the designer in the process of bitstream creation and the initial device partitioning. Some of that research could be towards the creation of efficient task graphs that exploit the technique of Joint Bitstreams and multitasking. A good start towards that could be [25] and [26]. We also plan to add a priority constraint to our tasks, in order to signify a task that must be executed. Finally, as we have stated earlier, the advantage of different clocking frequencies depending on the placement of a task could be exploited.

Bibliography

[1] C. Kao (2005): “Benefits of Partial Reconfiguration”. In: Xcell Journal. Fourth Quarter.

[2] Software-defined radio. http://en.wikipedia.org/wiki/Software-defined_radio

[3] Burns, J.; Donlin, A.; Hogg, J.; Singh, S.; de Wit, M.; "A dynamic reconfiguration run-time system" *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, vol., no., pp.66-75, 16-18 Apr 1997

[4] Esam El-Araby, Ivan Gonzalez, Tarek El-Ghazawi (2009): “Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing”. In Journal ACM Transactions on Reconfigurable Technology and Systems Volume 1 Issue 4, January 2009 Article No. 21.

[5] Lysaght, P.; Blodget, B.; Mason, J.; Young, J.; Bridgford, B.; , "Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs," *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, vol., no., pp.1-6, 28-30 Aug. 2006

[6] H. Walder and M. Platzner (2004): “A Runtime Environment for Reconfigurable Hardware Operating Systems,”. In: Proc. Int’l Conf. Field-Programmable Logic and Applications (FPL).

[7] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott Mcmillan, Prasanna Sundararajan (2003): “A self-reconfiguring platform”. In: Field Programmable Logic and Application Lecture Notes in Computer Science Volume 2778, 2003, pp 565-574

[8] Fu, W.; Compton, K.;, "An execution environment for reconfigurable computing," *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, vol., no., pp. 149- 158, 18-20 April 2005

[9] Durelli, G.; Pilato, C.; Cazzaniga, A.; Sciuto, D.; Santambrogio, M.D.; "Automatic run-time manager generation for reconfigurable MPSoC architectures," *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop*

on, vol., no., pp.1-8, 9-11 July 2012

[10] Steiger, C.; Walder, H.; Platzner, M.; "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *Computers, IEEE Transactions on*, vol.53, no.11, pp. 1393- 1407, Nov. 2004

[11] Thomas Marconi, Yi Lu, Koen Bertels, and Georgi Gaydadjiev: "Online Hardware Task Scheduling and Placement Algorithm on Partially Reconfigurable Devices". In: ARC '08 Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications Pages 306-311

[12] Chen, Y., Hsiung, P. (2005): "Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC". In: Yang, L.T., Amamiya, M., Liu, Z., Guo, M., Rammig, F.J. (eds.) EUC 2005. LNCS, vol. 3824, pp. 489–498. Springer, Heidelberg.

[13] T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev (2010): "3D Compaction: a Novel Blocking-aware Algorithm for Online Hardware Task Scheduling and Placement on 2D Partially Reconfigurable Devices". In: Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC), pp. 194-206, Bangkok, Thailand.

[14] Y. Lu, T. Marconi, K.L.M. Bertels, G. N. Gaydadjiev (2010): "A Communication Aware Online Task Scheduling Algorithm for FPGA-based Partially Reconfigurable Systems". In: FCCM '10 Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines Pages 65-68.

[15] K. Bazargan, R. Kastner, and M. Sarrafzadeh (2000): "Fast Template Placement for Reconfigurable Computing Systems". In: IEEE Design and Test of Computers, vol. 17, no. 1, pp. 68-83.

[16] Herbert Walder, Christoph Steiger, Marco Platzner (2003): "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing". In: IPDPS '03 Proceedings of the 17th International Symposium on Parallel and Distributed Processing Page 178.2

[17] Herbert Walder, Marco Platzner: "Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform". In: Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA).

[18] Thomas Marconi, Yi Lu, Koen Bertels, Georgi Gaydadjiev (2008): "Intelligent Merging Online Task Placement Algorithm for Partial Reconfigurable Systems". In: Proceedings of the conference on Design, automation and test in Europe Pages 1346-

1351

[19] T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev (2009): "A Novel Fast Online Placement Algorithm on 2D Partially Reconfigurable Devices". In: Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT), pp. 296-299, Sydney, Australia.

[20] Y. Lu, T. Marconi, G. N. Gaydadjiev, K.L.M. Bertels, R. J. Meeuws (2008): "A Self-adaptive on-line Task Placement Algorithm for Partially Reconfigurable Systems". In: Proceedings of the 22nd Annual IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp. 1-8, Miami, Florida, USA.

[21] Montone, A.; Santambrogio, M.D.; Sciuto, D.; "Wirelength driven floorplacement for FPGA-based partial reconfigurable systems," *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, vol., no., pp.1-8, 19-23 April 2010

[22] K. Papadimitriou, A. Anyfantis, A. Dollas (2010): "An Effective Framework to Evaluate Dynamic Partial Reconfiguration in FPGA Systems". In: IEEE Transactions on Instrumentation and Measurement (TIM), vol. 59, no. 6, pp. 1642-1651.

[23] K. Papadimitriou, A. Dollas, S. Hauck (2011): "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model". In: ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 4, no. 4.

[24] Papadimitriou, K.; Vatsolakis, C.; Pnevmatikatos, D.; "Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era," *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, vol., no., pp.1-5, 9-11 July 2012

[25] K. Papademetriou, A. Dollas (2006): "A Task Graph Approach for Efficient Exploitation of Reconfiguration in Dynamically Reconfigurable Systems". In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 307-308.

[26] K. Papadimitriou, A. Dollas (2006): "Performance Evaluation of a Preloading Model in Dynamically Reconfigurable Processors". In: IEEE International Conference on Field Programmable Logic and Aplications (FPL), pp. 901-904.

[27] André DeHon (1999): "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)". In: FPGA '99 Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays Pages 69-78.

[28] R. Tessier, H. Giza (2000): “Balancing Logic Utilization and Area Efficiency in FPGAs”. In: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications Pages 535-544.

[29] Compton, K.; Zhiyuan Li; Cooley, J.; Knol, S.; Hauck, S.; "Configuration relocation and defragmentation for run-time reconfigurable computing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.10, no.3, pp.209-220, June 2002

[30] Marconi, T.; Mitra, T.; "A novel online hardware task scheduling and placement algorithm for 3D partially reconfigurable FPGAs," *Field-Programmable Technology (FPT), 2011 International Conference on*, vol., no., pp.1-6, 12-14 Dec. 2011

[31] Overview Datasheet for High Performance AES (Rijndael) cores for Xilinx FPGA – Helion Technology

[32] Xilinx, Virtex-5 Family Overview, DS100 (v5.0) February 6, 2009