2009

Task scheduling and placement for reconfigurable devices

Mostafa Elbidweihy
Louisiana State University and Agricultural and Mechanical College, mkadry76@gmail.com

TASK SCHEDULING AND PLACEMENT FOR RECONFIGURABLE DEVICES

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

In

The Department of Electrical and Computer Engineering

by
Mostafa Elbidweihy
B.S., Cairo University, 1999
M.S., Cairo University, 2003
May 2009

To my LORD
Who created me and created every thing

To my Father, my Mother, my Brother, my Sister, my Beloved Wife and my Son

Who supported me by everything they can

ACKNOWLEDGEMENTS

The completion of this dissertation has been a monumental accomplishment in my academic career. I am grateful for this opportunity to acknowledge and thank those that have helped me throughout this process. If it were not for my father, mother, wife, brothers and sisters who have supported me and prayed for me throughout all of my different interests, I would not have finished.

I would like to express my sincere gratitude to my advisor Dr. Jerry Trahan for having accepted me as his student in this department and for his continuous help throughout my research by giving me ideas reading through my papers and chapters and correcting my mistakes. Without his help and support this work would never have been.

I would like also to thank my committee members: Dr. Suresh Rai, Dr. Ramachandran Vaidyanathan, Dr. J. Ramanujam, and Dr. Sukhamay Kundu. Thanks for their patience in reading this dissertation and their comments. Deep thanks to Dr. Dr. J. Ramanujam and Dr. Ramachandran Vaidyanathan for discussing some issues that I had in my research and suggesting solutions and ideas for me.

I would like to express my sincere gratitude to my father Dr. Kadry Elbidweihy for his continuous support. I would like to express my sincere gratitude to my mother, my brother and my sister for keeping me in their prayers. I would like to express my sincere gratitude to my wife for supporting me with her love and taking care of everything I need during my study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Partially reconfigurable devices allow the execution of different tasks at the same time, removing tasks when they finish and inserting new tasks when they arrive. This dissertation investigates scheduling and placing real-time tasks (tasks with deadline) on reconfigurable devices.

One basic scheduler is the First-Fit scheduler. By allowing the First-Fit scheduler to retry tasks while they can satisfy their deadlines, we found that its performance can be enhanced to be better than other schedulers. We also proposed a placement idea based on partitioning the reconfigurable area into regions of various widths, assigning a task to a region based on its width. This idea has a similar rejection rate to a First-Fit scheduler that retries placing tasks and performs better than the First-Fit that does not retry tasks. Also, this regions-based scheduling method has a better running time.

Managing how the space will be shared among tasks is a problems of interest. The main function of the free-space manager is to maintain information about the free space (areas not used by active tasks) after any placement or deletion of a task. Speed and efficiency of the free-space data structure are important as well as its effect on scheduler performance. We introduce the use of maximal horizontal strips and maximal vertical strips to represent free space. This resulted in a faster free space manager compared to what has been used in the area.

Most researchers in the area of scheduling on reconfigurable devices assumed a homogeneous FPGA with only CLBs in the reconfigurable area. Most reconfigurable devices offered in the market, however, are not homogeneous but heterogeneous with other components between CLBs. We studied the effect of heterogeneity on the performance of schedulers designed for a homogeneous structure. We found that current schedulers result in worse performance when applied to a heterogeneous structure, but by simple modifications, we can

apply them to a heterogeneous structure and achieve good performance. Consequently, the approach of studying homogeneous FPGAs is a valid one, as the scheduling ideas discovered there do carry over to heterogeneous FPGAs.

CHAPTER 1: INTRODUCTION

A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT, or more complex combinational functions such as decoders or simple math functions. Besides the programmable logic components, some FPGAs include memory elements, which may be simple flip-flops or more complete blocks of memory.

FPGAs are generally slower and more expensive than their application-specific integrated circuit (ASIC) counterparts. Also, FPGAs draw more power and cannot handle complex designs like analog designs. FPGAs have several advantages, however, such as a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs. FPGAs are more flexible than ASICs as they can be modified after the design is committed. For many digital designs, the design is first developed and tested on an FPGA chip. If any changes need to be done on the design, the designer will make the changes to the design and reprogram the chip for the new design. This process continues till the design is finished and working with no errors. At this step, the design can be developed on an ASIC for mass production. With lower prices of FPGAs now and when no mass production is needed, designers in some cases choose to go with an FPGA instead of an ASIC.

1.1. Reconfigurable Device

Reconfigurable devices are devices such as FPGAs that allow changing the configuration on the chip. This means changing the contents of the memory and setting the switches and look up tables. Partially reconfigurable devices allow changing part of the configuration while keeping the remainder of the configuration without change. A dynamically reconfigurable device is a partially reconfigurable device that allows a part of the chip to continue working while the

configuration for the other part is changing [X04-1, X04-2, X05, X07, X08, X09]. In principle, any RAM or FLASH memory-based FPGA can be dynamically reconfigured. In the currently known reconfigurable devices, the main performance overhead is the speed of reconfiguration. There are several ways to reduce the time taken for reconfiguration.

- When changing only part of a chip, the reconfiguration time in a partially reconfigurable chip is less than the reconfiguration time for a fully reconfigurable chip in which the whole chip has to be reconfigured. In partially reconfigurable devices, the reconfiguration time is usually proportional to the area being reconfigured.
- In non-dynamically reconfigurable devices, the entire chip halts during reconfiguration. In dynamically reconfigurable devices, only the part that needs to be changed halts while the rest of the chip is executing. This can reduce the reconfiguration time in dynamically reconfigurable devices as the reconfiguration time can overlap with the computing time of the unchanged part.
- Multiple-context configuration memory maps successive configurations into multiple contexts of the configuration memory. The reconfiguration is performed by swapping a selected inactive configuration memory context into the active context. The configuration in the active context controls the programmable switches on the reconfigurable device. This "context swap" can be performed quickly across the entire configurable array, so these devices have the shortest dynamic configuration times. Multiple contexts can be applied in fully, partially, and dynamically reconfigurable devices.

An FPGA can execute several tasks in parallel. The ability to reconfigure a chip (fully or partially) allows removing tasks from the chip and replacing them with other tasks. This opens some opportunities for researchers to study several aspects of operating systems for these devices, designing some system services that will help in utilizing chip resources and supporting

multiple applications [DW99]. These system services form a "reconfigurable operating system" [B96, WP03-2]. Designing an operating system for a reconfigurable device is similar to designing an operating system for embedded real-time operating system (RTOS) kernels for microprocessors [SWP04].

1.2. Scheduling Problem

The task scheduling problem for a partially reconfigurable device involves three key activities: determining when to execute tasks and where to place the tasks on the reconfigurable area, and managing free space to efficiently use the resources. Given a task *t* that is to be executed, a scheduler uses information about the currently free space to find whether a placement exists for *t*. If so, then the scheduler places the task and begins its execution. If not, then, depending on the specific problem, the scheduler may reject the task or attempt to schedule it to execute at a specific future time or retain the task to try to place it in the future. The scheduler's job also includes deleting finished tasks. When a task begins or finishes executing, the scheduler is to update the free space representation accordingly. To achieve these goals, we divide the job among three modules, where the scheduler is the main module, the placer is a subroutine of the scheduler, and the free space manager is a subroutine of the placer. To simplify the explanation, we will describe the actions of the modules for new tasks, saving for a later time the discussion of retrying tasks that did not find an initial placement. These modules are as follows.

- **Scheduler**: The scheduler manages the task scheduling process. It decides on a starting time for each task based on the information given by the other modules about the status of the chip (free space, occupied space, future status, etc.). To get this information, it passes task requirements to the placer module. Also, the scheduler module keeps track of executing tasks, and when a task finishes

executing, the scheduler passes this information to the placer to remove (delete) the task from the chip.

- **Placer**: When the scheduler passes information about a new task, the placer tries to find a location at which to place the task. If it finds a placement, then it requests the free space manager to update the free space. When the scheduler passes information about a task to be deleted, the placer deletes the task from the chip and requests the free space manager to update the free space.
- **Free Space Manager**: This module maintains a free space data structure to keep track of free (unoccupied) space. If the scheduler only places tasks to start immediately, then the free space manager needs to maintain information on only the currently free space. If the scheduler can assign tasks to start at some time in the future, then the free space manager must also maintain information about free space as far in the future as tasks are scheduled.

Different variations of the scheduling problem exist, for example, in the offline version, all tasks are given initially along with their release times (first time at which each task can start executing), while in the online version, the scheduler has no information about tasks to arrive in the future. Different variations of the scheduling problem will be discussed in detail in the next chapter.

In part of this work, we will design a scheduling system that includes the three modules. We are interested on solving different scheduling problems. The first problem we solved is scheduling real-time tasks with deadlines. For ease of discussion, we will call the scheduling system as a scheduler. The results generated by proposed schedulers will be compared to results for basic schedulers and also will be compared to schedulers proposed in the area. The second objective is to construct data structures to manage free space information and that can provide

quick and efficient ways for a generic online placer to find a required area. These data structures should also be able to efficiently update the free space records when a task finishes. The third objective is to investigate how the heterogeneous structure of current FPGAs can affect the performance of schedulers. Also, one of our objectives in this part is to design a new scheduler for heterogeneous structures that has a performance close to the performance of schedulers on homogeneous structures.

1.3. Outline

Chapter 2 outlines a background on reconfigurable devices and also defines problems of interest in this research area and the irrelevant parameters. Prior work is reviewed in Chapter 3. In Chapter 4, we explain in detail schedulers that have been used in the area by other researchers and the enhancement we added to these schedulers. Also, in Chapter 4, we will explain the first main contribution of our research which is the regions-based scheduler. Results that compare all presented schedulers will be described at the end of Chapter 4. In Chapter 5, we will discuss different data structures in the area, and we will examine a widely used free space data structure (maximal empty rectangles) and detail the update process for the data structure each time a task is inserted or deleted. In Chapter 6, we will present the use of maximal horizontal and vertical strips (MHVS) as a free space data structure. Results comparing existing data structures with MHVS set will be presented in Chapter 6. Chapter 7 will study how the heterogeneous structure of an FPGA chip can affect the performance of the scheduler, and we will suggest some recommendations on how to reduce this effect. We will present a scheduler designed to work on heterogeneous FPGAs. Results that show the performance of current schedulers on heterogeneous structures compared to the proposed scheduler will be shown in the same chapter. In Chapter 8, we will present some open problems.

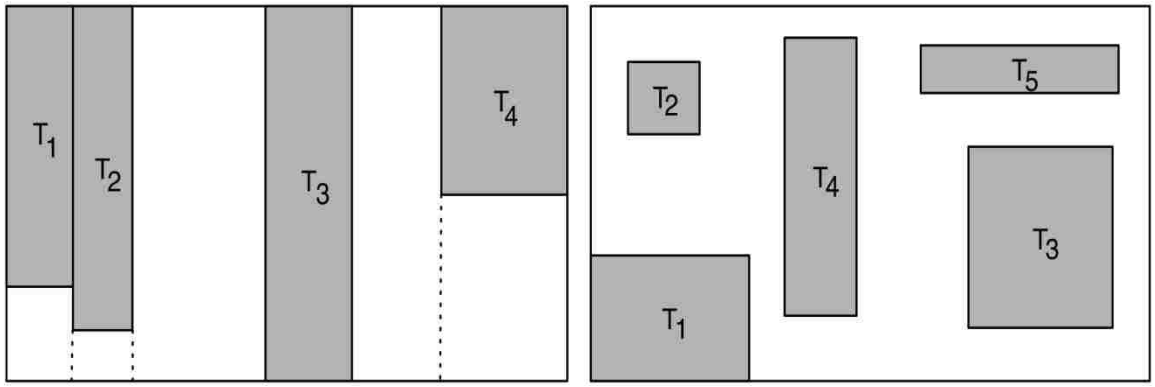
CHAPTER 2: BACKGROUND

In this chapter we will cover the characteristics of the problems with which we are dealing and also we will give a background description on reconfigurable devices and their structure. The problem characteristics include the characteristics of the tasks on which the online scheduler works and also include various parameters and capabilities of online schedulers.

2.1. Reconfigurable Device Models

One can classify reconfigurable devices based on the structure of the chip and how tasks can be placed on the chip. A reconfigurable chip consists of a number of reconfigurable units (RCUs) arranged in a rectangular grid of area $r \times c$ where r (c) is the number of rows (columns). The placer can model this 2D area as a 2D space or a 1D space [SWPT03] (Figure 2-1). In a 2D model, a task can have size $h \times w$ for any $h \leq r$ and $w \leq c$. A placer can assign a task to any region of free space as long as this free space has a width greater than or equal to w and a height greater than or equal to h. In a 1D model, each task will be treated as if its height is spanning the height r of the chip regardless of its original height and can have any width $w \leq c$. This model corresponds to the partial reconfiguration in some FPGAs, such as the Xilinx Virtex-I, Virtex-II, and Virtex-II Pro families [X05], in which the smallest unit of partial reconfiguration is a column of RCUs. In a 1D model, tasks can be placed only along the horizontal device dimension. After the task is placed, the remaining free space in the columns used cannot be used to accommodate any other tasks before the current task finishes. The fine-grained reconfiguration of the 2D model is not available in the market. Some new chips available in the market fall between 1D and 2D and allow configuration to be done on blocks less than a column height: Virtex-4 and Virtex-5 [X08, X09]. Some recent research targeted these capabilities of the Virtex-4 [LBM06, SBB06] and Virtex-5 [CGG08, HPLD08]. Conger *et al.* [CGG08] used this ability and proposed partial

reconfiguration (PR) design flow methodologies to help designers to use the capabilities of PR without having to deal with many internal details. Also, Huang *et al.* [HPLD08] created a structure that allows the smallest unit of reconfiguration to be 16 CLBs in a column. They used this structure to propose an FPGA-based scalable architecture for DCT computation.



(a) 1D model (b) 2D model

Figure 2-1 Different models of reconfigurable devices

Most of the work in the area of reconfigurable design assumes a homogeneous FPGA area in which the task can be placed anywhere on the chip with no constraints. Many FPGA chips are not homogenous but there are some buses, RAM modules and some multipliers between the CLBs, as shown in Figure 2-2. The assumption of using homogeneous structure was made for several reasons. One of the reasons is that the scheduling problem on 2D model is a hard problem and it is useful to study an easier version of the problem first as assuming heterogeneous structure will make it harder.

The problem of scheduling and placing a set of tasks on an FPGA chip is similar to the classic bin packing problem that has been attacked by many researchers in the scheduling field [C83, CW98] where the goal is to pack a collection of objects into the minimum number of fixed-size "bins". Some of the designed online schedulers are based on similar ideas to bin-packing algorithms [BKS00]. Researchers have proposed many algorithms for placement under a

variety of assumptions about tasks and the reconfigurable device [ABT03, BKS00, SWP04, SWPT03, WP03-1].

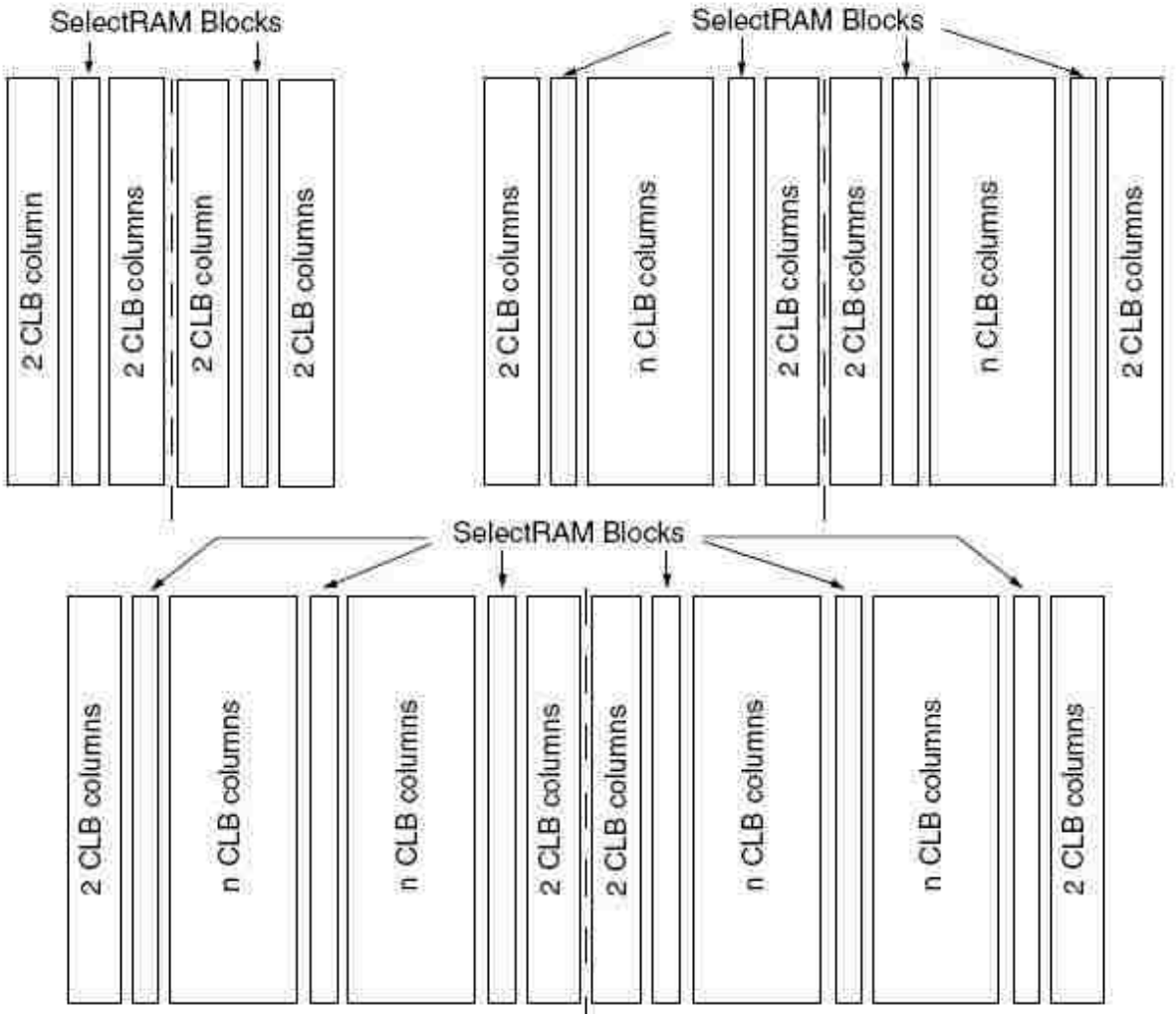


Figure 2-2 Chip structure in Xilinx XCV series [X07]

The problem can be different based on whether the chip is a 1D model or a 2D model. The difference between these two models makes a difference in how to design the placer algorithm as the way to handle the free space in the two models is different. Schedulers have been designed for both the 1D model [AT03, BJS04, DP05-1, KPR04, SWP04, WP03-1], for placing 2D tasks on 1D structure as [FVA08, HSB06, SBHB08, SKHB08] and for the 2D model [ABBT04, ABF04-1, ABT03, BKS00, SWPT03, TFS01, TSMM04]. The scheduler for the 2D

model is more complicated as the amount of data that needs to be known about the free space is more than the amount of data that needs to be known in the case of the 1D model.

Researchers are also interested in different problems by having different assumptions about the set of tasks they are handling. One main difference is handling real-time tasks (tasks with deadlines) or handling tasks without deadlines. These variations are not all explored and there is still room for improvement on problems already studied; some problems have not yet received much attention such as real-time tasks (tasks with deadlines). Most of these ideas will be discussed in the prior work review in Chapter 3 and some will be covered in Chapter 4 in some detail.

Scheduling and placing problems come in offline and online varieties. Offline problems assume that all information about all tasks is available initially, and the algorithm is looking for the best placements for all of them to satisfy certain criteria. As has been studied before [BKS00, FKT01], the offline placement problem for a general set of tasks (tasks with no constraints) is NP-complete, so it is very unlikely that an algorithm exists to obtain an optimal solution to the problem in less than exponential time. For online problems, on the other hand, the algorithm is not aware of tasks that will arrive in the future. Decisions of the algorithm are based on the current information on hand about already arrived tasks.

Tasks have different characteristics including: arrival time, execution time, shape, and size.

- **Arrival time** is the time the task arrives. Generally, this is the first time at which a placer receives information about a task and also is the first time at which the task can begin execution.
- **Execution time** is the total time needed by the task to finish and includes any reconfiguration or loading time.

- **Task shape**: Almost all placers assume that a task has a rectangular shape with the size of the task defined by width (*w*) and height (*h*).
- **Task size**: Most work in the area assumes that a scheduler cannot change the size of a task even if the task will keep the same area, but some work assumes that a task can be rotated [AT03].

These parameters are the basic parameters; other parameters exist depending on the problem at hand.

- **Deadline** is the time by which the task must finish. If a task is not able to satisfy its deadline, then it has to be rejected.
- **Laxity** expresses the amount of slack time available between the current time and the deadline, in which a task can wait before being placed and still satisfy its deadline: laxity = deadline – current time – execution time.
- **Precedence** means a certain task has to finish before another task can start. Problems with precedence have been attacked by some researchers. For some specific problems with constraints, optimal solutions can be found. Fekete *et al.* [FKT01] found an optimal solution for a placement problem with precedence between tasks.
- **Priority level** for tasks is another parameter. Priority among tasks may arise at different levels in problems. Priority can be considered between tasks that arrive together so that, based on the priority, the placer decides which task to start placing first. Priority also can be considered on a higher level in which the placer can stop (pre-empt) a currently executing task to place a higher priority task.

The final parameter that classifies a problem is the set of choices for how the placer responds to arriving tasks. When a task arrives, the response of the online scheduler and the placer can have two or more options. The two options common to all schedulers are:

- Yes: place immediately, and
- No: reject the task.

For tasks that cannot be placed immediately, sometimes the placer can choose from the following options.

- Maybe: the placer found no placement now, but, while laxity remains, the algorithm will try again in the future to find a placement that will still satisfy the deadline for the task.
- Reserve: the placer has identified a known placement in the future and reserved the area for the task.

2.2. Performance Measurement

Different ways exist to measure the performance of a scheduling algorithm. The particular problem we are trying to solve determines which are relevant.

- **Rejection ratio**: ratio of the number of rejected tasks to the total number of tasks arrived during a certain time period.
- **Routing cost** [ABBT04]: cost to route the communication required by each task. This communication can be between the task and other tasks or the boundary of the chip (I/O). The routing cost is the sum of the weighted distance between the communicating points. Weighted distance represents the distance between the two communicating points multiplied by a number that depends on the width of the communication channel and amount of time for which it needs to be established. If there is no communication

between two modules or between the task and the boundary of the chip, then the routing cost between the two tasks or between the task and the boundary is zero.

- **Total execution time**: the total time taken by all tasks to finish executing starting from when the first task begins executing until the last task finishes.
- **Average waiting time** [WSP03]: waiting time is the time that a task has to wait until space becomes available. Average waiting time is the total waiting time divided by the number of tasks.

The particulars of a problem determine task characteristics and which performance criteria are relevant. For instance, for tasks with deadlines, the rejection ratio is important, while for tasks without deadlines, this is irrelevant.

2.3. Standard Approaches in Schedulers

Several basic ideas have been implemented in the area of offline and online schedulers. Each method has its advantages and disadvantages. One of these basic ideas is *First Fit* (FF). It is obvious from the name of the algorithm that FF places a task in the first place found to accommodate the task. There are different versions of FF based on where to start searching the chip. FF may search the chip from the bottom-left corner going right then to the top of the chip. Other ways of searching free space exist depending on how the free space is represented. One of the main advantages of FF is its easy implementation. One of the disadvantages of FF is the use of the first found location to place the task without looking at how this decision will affect the placement of tasks in the future and how this decision will affect the chip utilization.

Best Fit (BF) is the second standard algorithm for this type of problem. BF searches all possible locations on the chip for the best location in which to place the task. The definition of "best" location depends on minimizing or maximizing certain criteria. For rectangles with sufficient width and height, some of these criteria are:

- smallest area,
- smallest width,
- smallest height, and
- least routing to the I/O pins.

The time needed for BF to find the best placement varies based on the way that free space is represented. A search can involve anything from searching the whole chip to just searching the first few items in an ordered data structure. BF has an intuitive advantage over FF in that by selecting the best location to place the task, it leaves free space that is less fragmented after placing the task than that left by FF. On the other hand, BF is slower than FF because FF can schedule a task as soon as it finds a sufficiently large free space, while BF must examine all possible locations.

CHAPTER 3: PRIOR WORK

In this chapter we will present some of the work done in the area of reconfigurable devices' operating systems. There are several problems of interest in this field and they are slightly different. As mentioned before, some of the differences between the problems come from the different task parameters and algorithm parameters. These different problems make it hard to compare the work in the area to other researchers unless they deal with the same problem. Some different problems and the questions raised in this area were presented by Diessel and Wigley [DW99]. They identified several opportunities and aspects of task placement on an FPGA. Their technical report laid the foundation for most of the work in the area. They stated the reason behind the need for an operating system to control the function of reconfigurable devices, and they outlined most the functions to be performed by the operating system. These functions include, but are not limited to: design capture (the method of specifying the application), partitioning, placement, routing, and scheduling. They also raised some questions and issues on scheduling and placing such as task sizing, task placement, pre-empting tasks, and inter-task communication.

Most of the work done in this area targets the reconfigurable devices made by Xilinx, particularly the Virtex family of FPGAs [X04-1, X04-2, X05, X07]. Some recent work targeted Virtex-4 and Virtex-5 [X08, X09] chips. Because of the different operating system functions of interest and the different problems of interest to researchers, this chapter is organized as follows. The first part of the chapter will deal with the offline problem. In the second and the third parts, we will review the work done in the area about 1D online schedulers and 2D online schedulers. Then we will present some ideas of the work done on two main aspects of the operating system: defragmentation and free space management.

3.1. Prior Work on Offline Problem

The general offline problem for scheduling a general set of tasks on an FPGA, without any dependence among tasks, is NP-complete. For the problem "sequencing with release times and deadlines," a set of tasks is given, each with a length, release time, and deadline. You want to determine whether there exists a one-processor schedule that satisfies release time constraints and satisfies all deadlines. This offline problem is NP-complete [GJ79]. The offline placer for a general set of tasks on an FPGA generalizes to this problem and that is why this offline FPGA scheduling problem is NP-complete. Since a solution to the online scheduling problem implies a solution to the offline problem, we can conclude that the online problem is also NP-complete.

Fekete *et al.* [FKT01, TFS01] presented algorithms to solve the problem of offline scheduling for a set of tasks with precedence constraints. They were able to solve two problems:

- find the minimum execution time of the given problem on an FPGA of fixed size, and
- find the FPGA of minimum size to accomplish the tasks within a fixed time limit.

3.2. Prior Work on 1D Schedulers

As described before, the problem of scheduling a set of real-time tasks on a reconfigurable device is similar to bin-packing problems. There are several heuristics that solve bin-packing problems. Chazelle [C83] implemented a bottom-left heuristic for two dimensional bin-packing. He proved that his algorithm can pack (insert) n rectangles in an infinite height vertical bin of fixed width with the bottom-left heuristic in time $O(n^2)$, using $O(n)$ space. This algorithm complexity is for insertion only without deletions.

Steiger *et al.* [SWP03, SWP04] presented heuristics for two online schedulers: the horizon and the stuffing techniques. They first presented the schedulers to work on the 1D model, and then they showed how to change these algorithms to work on the 2D model. They

found that the 1D model schedulers depend on the task aspect ratio. They also discussed design issues for reconfigurable hardware operating systems and the problem of online scheduling for real-time tasks with deadlines to partially reconfigurable devices.

Walder and Platzner [WP03-1] discussed different area models for reconfigurable devices. They also introduced a 1D block-partitioned model and devised an online scheduling system that schedules tasks according to several non-preemptive and preemptive policies. The block partition is similar to our regions method (Chapter 4), but the regions method targets a 2D model. They designed their scheduler to schedule real-time tasks with deadlines, but in their simulations they used average response time as a performance measurement. They later [WP03-2], continued their earlier work [WP03-1] and implemented a case study based on the XESS prototyping board (based on the Virtex XCV-800 FPGA). The case study executes a control/data-flow application that performs networking and multimedia tasks in an OS environment.

On the same trend, Baskaran *et al.* [BJS04] enhanced the work done by Walder and Platzner [WP03-1]. They introduced an efficient partitioning algorithm for the placement of real-time tasks on a 1D reconfigurable model. Instead of just using a fixed block size, their algorithm allows merging and dividing blocks to generate blocks with different sizes.

Kalte *et al.* [KPR04] presented a "system integration approach" that enables online, fine grained, 1D placement of modules on a Xilinx Virtex FPGA. They focused on designing a system that can work on current FPGA tools. They focused on the hardware details and explained the details of the hardware implementation. For communication between modules, they proposed a tri-state bus that spans the whole chip in a horizontal manner and through this

tri-state bus they allowed the communication between modules and between modules and the chip border.

Chen *et al.* [CH05] proposed a fragmentation-based scheduler for a 1D structure. They defined a metric composed from the area and the execution time of a task then used this metric to choose the best placement for the task on hand using a best-fit strategy.

Hubner *et al.* [HSB06] discussed in detail the memory configuration and the structure of the Virtex-II architecture. Also they showed how communication will be done in a column-oriented manner. They divided the reconfigurable area of the design into vertical configuration slots. A module can be placed in any of these slots regardless of its vertical position within the slot. Also they proved that within a slot one can place different modules on top of each other as long as the sum of the module heights does not exceed the slot height.

Some researchers presented ideas to schedule 2D tasks on a 1D model. The main problem they faced is the way the reconfiguration is done in a 1D reconfigurable device, as the whole column has to be reconfigured and so the task height has to span the entire height of the chip. Ahmadinia *et al.* [AT03] presented an online placer for a 1D model of a reconfigurable device. Their algorithm selects a placement for a task as if in a 2D model. To actually place the task on the FPGA, the algorithm preempts tasks that share a column with the new task and places all of them together. As the algorithm has to preempt some tasks, the algorithm schedules the new task in the location that has the least interference with the currently placed tasks. Interference between tasks in their work means sharing the same column. Also, they allowed task rotation mainly to make $h \geq w$ as their algorithm is column oriented. They also suggested that when a task finishes, the algorithm will not delete the task from the chip but will keep it as long as the space is not otherwise needed, in case it is needed in the future.

Sedcole *et al.* [SBALB05] proposed two methods for implementing dynamic partial reconfiguration on the Virtex chip also with a goal of placing 2D tasks on a 1D model. The proposed methods enable the modules to be placed on arbitrary areas of the FPGA. Other researchers have also studied placing 2D tasks in 1D slots [FVA08, HSB06, SBHB08, SKHB08].

Periodic tasks are tasks that must be executed at regular intervals. The deadline for these periodic tasks can be defined as the start of the next interval (period). Danne and Platzner [DP05-1, DP05-2] presented two scheduling algorithms for the placement of periodic real-time tasks. The first algorithm is an adaptation of the well known Earliest Deadline First (EDF) heuristic. The second algorithm uses the concept of servers that reserve area and execution time for other tasks. Tasks are successively merged into servers, which are then scheduled sequentially. Danne and Platzner [DP06-1] extended their work by applying a partial scheduling approach. Also they assumed that the same task can be implemented in different variants. The variants of tasks are differing in area and speed characteristics, for example, fast but resource-consuming and slower but resource-sparing versions. By extending one of the heuristics for the 2D level strip packing problem, their partitioned-EDF scheduling approach achieves acceptable device utilization. Danne *et al.* have continued working on periodic tasks [DMP06, DP06-1, DP06-2].

3.3. Prior Work on 2D Schedulers

In this section we will discuss some of the work done in the area on 2D online scheduling. As current FPGA devices do not allow fine-grained 2D placement, researchers have made some assumptions and designed their 2D schedulers based on them. Ahmadinia *et al.* [ABT03] presented two clustering methodologies that cluster real-time tasks before placing them on the reconfigurable device. In their research they targeted a fully reconfigurable FPGA. The

main idea behind the clustering is to collect some tasks that have similar features to be placed all together at the same time. They had two types of clusters: the first one is based on tasks with similar run time, and the second one aims at reducing communication among clusters by using a trade-off function for inter-task communication. They used FF as the placement heuristic to place these clusters on the chip. The results showed significant reduction in communication cost. They handled real-time tasks with deadlines. When the algorithm constructs the clusters, it checks that each task within a cluster will satisfy its deadline when placed at the cluster start time.

To minimize the communication costs among running tasks and between running tasks and the outside of the chip, Ahmadinia *et al.* [ABBT04, ABF04-1, ABF04-2] presented a new idea for an online scheduler. To reduce the communication cost, first, they found the optimal placement of a new task to minimize the communication cost with the currently executing tasks. Second, they tried to find the nearest possible position to the optimal point to place the task without overlapping with any of the currently executing tasks.

Tabero *et al.* [TSMM04] presented several heuristics for selecting task location based on a vertex list structure which they had developed [TMS03]. They applied their algorithm on a real-time set of tasks with deadlines and used the rejection ratio as a performance measurement. They compared the classical FF with the bottom-left heuristics to the FF algorithm operating with their vertex list approach. Also they compared the same for the BF algorithm and they proved that the vertex list enhanced the performance of the classical FF and BF algorithms.

Handa and Vemuri [HV04-3] used an area matrix of size equal to the size of the reconfigurable device. Each location in this matrix represents one CLB on the FPGA and indicates whether the CLB is occupied or free. They designed three different hardware architectures to be placed on a small portion of the FPGA to achieve ultra-fast placement. The

three hardware architectures search in the area matrix to find a free space to place the task on hand.

Handa and Vemuri [HV04-4] proposed a task queue management data structure for in order and out of order task scheduling strategies. The in-order task set means that tasks in the input priority queue may be data dependent upon each other, so the tasks need to be processed in order. In out of order processing, the tasks are not data dependent upon each other, so they can be executed in any order. They maintained empty area as a list of maximal empty rectangles.

As a different approach to solve the 2D scheduling problem, Bazargan *et al.* [BKS00] presented the idea of mixing the hardware and software execution of tasks. They assumed that when a task has no free location to be placed on the chip, then it can be executed in software at a time penalty. They combined their offline placement with a scheduling algorithm. The offline algorithm gives estimates of the available reconfigurable functional unit area, and the scheduler can use this information to schedule the tasks. They tried different free space search heuristics and applied them on the two basic algorithms: First Fit and Best Fit. The penalties applied when a task has to be executed on the software were the comparison parameter they used to compare these algorithms. In their simulation they found the SSEG (shorter segment) and LSQR (larger square empty rectangle) are the best among partitioning heuristics that keep only $O(n)$ empty rectangles.

Walder and Platzner [WP02] introduced a new operation, footprint transform, to be used with the current online schedulers. A footprint transform changes the shape of a task. This process is more complex than relocation. The idea behind the footprint transform is that they assumed that task shape does not have to be rectangular. They assumed that tasks are coarse-

grained non-rectangular and that they consist of rectangular subtasks that can be placed all separately at the same time and save space.

Steiger *et al.* [SWPT03] designed two online placers to deal with tasks that have arbitrary arrival times and synchronous (periodic) arrival times. Their algorithm uses an enhanced version of the non-overlapping empty rectangles to manage free space. In their work they represented the empty rectangles as tree nodes, and they formed an algorithm that allows insertion, deletion, and merging of rectangles from the tree. They used the First Fit algorithm as a reference point with which to compare their work. They dealt with real-time tasks with deadlines, so they used rejection ratio as a performance measurement. Also, they used the run time of the schedulers as a way to compare their work to FF. We compare our work on the region-based method to their results in Chapter 4.

Other recent research exists in the area of 2D scheduling [ABF07, CGLD07, CDHG07, TSMM06, TNY06, ZWHP07].

3.4. Prior Work on the Defragmentation Problem

Diessel *et al.* [DE97, DE01, DEMSS00] proposed effective heuristics to solve the problem of fragmentation by moving tasks on the chip to free more space to accommodate the new incoming tasks. They called this process as task compaction, and they have two types: full compaction and partial compaction (all tasks must be moved or just some of them). The task compaction is done by reloading the task configuration and state.

Kalte *et al.* [KPR04] discussed the issue of defragmentation. They studied 1D placement in which defragmentation is less complex than in the 2D case. To place a new module on the chip, the algorithm removes (some) existing modules from the chip and places them again on the chip

in different places in order to build one or a few coherent free areas that can be used to place new arriving tasks. The reallocation process means placing tasks again from scratch.

One of the good works presented in the area of defragmentation is the work done by Septien *et al.* [SMMT06]. They presented a fragmentation metric to estimate when the FPGA fragmentation status is critical. Also, they presented several heuristics to perform the de-fragmentation process when the chip reaches a critical status. Their heuristics are based on using the vertex set list presented by Tabero *et al.* [TMS03] to represent the free space on the reconfigurable chip.

The work done by Van der Veen *et al.* [VFM05] extends the work of Teich *et al.* [TFS01]. They presented a defragmentation algorithm for an FPGA. Their algorithm is based on the idea of packing classes and uses an optimal solution for the 2D strip packing problem.

While most researchers are trying to solve the problem of defragmentation by designing algorithms to help reducing the defragmentation and increase the chip utilization, Compton *et al.* [CLC02] decided to attack the problem from a different direction. In their work, they proposed a new architecture designed specifically to exploit the benefits of run-time relocation and defragmentation. They referred to their architecture as the Relocation/Defragmentation (R/D) FPGA.

Handa and Vemuri [HV04-1] proposed a fragmentation metric different than Septien *et al.* [SMMT06]. They assumed that if an empty rectangle can accommodate a task as large as twice the average size of the task being placed, then the area inside that rectangle is not fragmented. They build a fragmentation matrix based on this assumption and used it to design a fragmentation-based placement strategy. Their proposed placement strategy works better than FF and BF placement strategies.

Gericota *et al.* [GASF02] proposed an idea to solve the problem of defragmentation by making a copy of all CLBs without interrupting the currently executing tasks. The idea is to then apply the defragmentation technique on the copy and transfer all the state information and routing resources to the new copy. When everything is ready, stop the original and put the copy in active status. Other researchers have continued investigating fragmentation measurements and fragmentation-based schedulers [EES07-1, EES07-2, FVA08, KKP06, SMMT08].

3.5. Free Space Management Prior Work

Bazargan *et al.* [BKS00] proposed one of the main techniques in the area of free space management. Most of the work done in this area is related to their work. They proposed the idea of representing the free space as a set of non-overlapping rectangles, and showed how this representation can help in finding a placement for a newly arrived task and also they showed how this representation can be updated when a task is to be deleted from the chip. Free space stored as non-overlapping rectangles does not recognize all the empty rectangles. This can lead to some tasks being rejected even though there is enough space to accommodate them but this space is divided between two non-overlapping rectangles. To solve this problem, they presented the idea of allowing overlapping of the empty rectangles, specifically overlapping maximal empty rectangles (MERs). They call this method as Keeping All Maximal Empty Rectangles (KAMER). For n tasks, we can have $O(n)$ non-overlapping rectangles and in the case of MERs we can have $O(n^2)$ rectangles. They proved that searching the non-overlapping empty rectangles to find those that can accommodate the module can be done in $O(n \log n)$ space to store the empty rectangles and $O(\log n + K)$ time to check which empty rectangle can accommodate a task, where K is the number of reported candidate rectangles (rectangles that have enough space to accommodate the task).

Ahmadinia *et al.* [ABBT04] proposed an idea of managing the occupied space rather than the free space on the chip. They proved that by managing the occupied space, their placement algorithm has a complexity of $O(n)$ compared to $O(n^2)$ for the KAMER proposed by Bazargan *et al*. [BKS00], where n is the number of running tasks. To test the performance of their scheduling algorithm that keeps track of occupied space, they compared their work to the classical FF and BF, establishing that their algorithm can achieve better results.

Steiger *et al.* [SWPT03] developed an enhanced version of the partitioner presented by Bazargan *et al.* [BKS00]. Bazargan *et al.'s* placer uses heuristics to decide whether a free space rectangle is split vertically or horizontally upon task placement. The key to enhance this partitioner is to delay the decision of which way you split the free space rectangle till the new task arrives.

Walder *et al.* [WSP03] presented three partitioning algorithms based on the Bazargan *et al.* [BKS00] approach. They enhanced the non-overlapping rectangles technique. Two of these approaches are similar to other work done by them [SWPT03]. The third is based on a 2D hashing table to find a feasible task placement with a run time complexity of $O(1)$, but they did not account for reconfiguration time and also they did not account for the update time needed to update the hashing table.

Tabero *et al.* [TMS03] presented an area management algorithm to manage the free space on the chip. Their techniques derive from bin-packing heuristics. They represented free space by vertex sets; each one describes the contour of an unoccupied area fragment in the reconfigurable device. Some of these vertices may be used as candidate locations to place tasks. Their approach has a reasonable complexity of $O(n^2)$ time, where n is the number of running tasks, and gives good results in terms of device fragmentation.

Handa and Vemuri [HV04-2] designed an algorithm to find the free space on an FPGA and represented this as a list of overlapping maximal empty rectangles. They converted the representation of free space on the chip to staircases, and they used the staircase data structure to find the overlapping maximal rectangles list.

3.6. Applications Using Reconfigurable Devices

Partial reconfiguration at run-time can save hardware in several types of applications. For example, Becker *et al.* [BHH07] presented run-time reconfigurable systems in the automotive domain. Their motivation to use one FPGA in the place of several chips was the increasing use and cost of computer hardware in automobiles. Also, Sedcole *et al.* [SBB06] designed run-time reconfigurable solutions for software-defined radio and video image processing. In all these instances, hardware reuse was prominent.

Braun *et al.* [BHB07] used the Xilinx Virtex-II as an adaptive circuit-switched Network-on-Chip. They implemented a new switch that resulted in reducing the controlling logic. Controlling the switch can be done by reconfiguring the content of specified look-up tables. Anderson *et al.* [AFK08] designed an adaptable signal processing prototype based on a run-time reconfigurable system. Huang *et al.* [HPLD08] created a structure that allows the smallest unit of reconfiguration to be 16 CLBs in a column. They used this structure to propose an FPGA-based scalable architecture for DCT computation. By using partial reconfiguration, the elements of the DCT architecture can be changed at run-time without the need to stop the chip while changing the configuration. Some applications will benefit from the improvement in time like audio and video applications.

3.7. Scheduling on Heterogeneous FPGAs

Very few researchers designed schedulers for heterogeneous structures. Fekete *et al.* [FKS08] proposed a new method based on dynamic relocation of module positions during run time. In their research they considered a 1D model chip. They proved that they had an improvement in the quality of the module layout over static layout by 50%.

CHAPTER 4: ONLINE SCHEDULERS AND PLACERS

Different problems in the area of scheduling tasks on reconfigurable devices require different schedulers. In this dissertation, we are interested in solving the online problem in which the scheduler receives tasks online and tries to place them as soon as space is available and before the laxity of the task reaches zero. In Section 4.1, we will describe how the first fit (FF) scheduling algorithm will operate in the Yes-No-Maybe (YNM) mode. Many papers in this area designed algorithms to operate in YNM mode but compared their results to FF operating in Yes-No (YN) mode [ABBT04, AT03, BKS00, HV04-1, SWP04, SWPT03, TSMM04, TMS03, WP02]. We decided to make a fair comparison by comparing our algorithm operating in YNM mode to FF operating in YNM mode (FF_YNM). The interesting result that we found is that YNM mode gives FF a significant performance improvement over YN mode. This was our first contribution in this research. In Section 4.2, we will describe the main idea about our region-based scheduler and in Section 4.3 compare the performance of our idea to FF_YNM and other algorithms and show why we expect our scheduler to give better performance. The comparison will be from different points: rejection ratio, ease of implementation, search time, and chip utilization.

4.1. FF_YNM Algorithm

We focus on two main modes of operation for online scheduling algorithms. In the first mode, YN mode, the placer will try to place a newly arrived task immediately and if no space is available, then the scheduler rejects the task. In this mode, the placer completely ignores the task's laxity. The second mode is YNM mode in which the placer will try to place the task immediately, and if the scheduler finds no placement and the laxity of the task is greater than

zero, then the scheduler will keep it in a pending queue and attempt to place it as long as laxity remains.

The algorithm for scheduling real-time tasks consists of three modules: scheduler, placer, and free space manager. The placer and the free space manager are the same for the two modes YN and YNM, but the scheduler is different. In the following paragraphs we will describe first the modules common to the two modes and then after that we will describe the difference between the two schedulers. *FF_Placer* (Figure 4-1) is the first common module that the scheduler module will call to delete a task that finished executing and to find a placement for a task. The placer has no output in case of deleting a task, but in the case of placing a task the output will be the location found, if any.

The second common module is the free space manager, and this module is responsible for keeping the free space data structure updated every time a task is placed or deleted. This module will be called by the placer. The function of the free space manager is integrated with the functions of the placer and the end result of the two modules is to be able to search for placement and update the data structure for the free space.

The placer performs a search in the free space area to find a large enough rectangle to accommodate the task. The order in which the free space rectangles will be searched depends on the implementation and on the data structure used to represent the free space. This search will be done based on the information passed to the placer from the free space manager.

Some common parameters will be passed between the modules. The variables that we use to represent a task contain all the required information about the task such as task size (h_t: task height, w_t: task width), execution time, arrival time, deadline, and laxity. Also, this variable contains a slot called *location.* The location will be empty when the task arrives and will contain

the location at which the task is placed when a placement decision has been made. *Placed* is a variable by which the placer indicates if it found a placement. If *placed* is zero after the placer module has been called, then this means it found no placement. If *placed* is one after the placer module has been called, then this means it found a placement for the task. Note that the above algorithm assumes bottom-left corner placement.

Placer:

Inputs: task *t*, *option*
Output: update *location* slot in *t*, *placed*
Runs when: called by scheduler
FF_Placer (*t, option*)
    If *option* = 1
        Delete task *t* from the chip
        FF_free_space_manager(*t*, 1)
    Else
        placed = 0
        r = 1 (assume the free rectangles are indexed from 1)
        While *placed* =0 and *r* ≤ number of empty rectangles
            %% Check free rectangle *r* on FPGA %%
            If $h_r \geq h_t$ and $w_r \geq w_t$ Then
                place *t* in bottom left corner of *r*
                placed = 1
                t (*location)* = bottom left corner coordinates of *r*
                FF_free_space_manager (*t*, 0)
            r = *r* + 1
****
Inputs: task *t*, *option*, free space data structure
Output: updated free space data structure
Runs when: called by placer to update the free space
FF_free_space_manager (*t, option*)
    If *option* = 1
        Add the space occupied by *t* to the free space data structure
    Else
        Delete the space occupied by *t* from the free space data structure

Figure 4-1 Placer and free space manager for task scheduling problem

The *FF_YN_scheduler* (Figure 4-2) and the previous two common modules form the FF_YN algorithm. *FF_YN_scheduler* has simple functions to perform. The scheduler runs every

unit time to check if any task finishes executing, and if so, then it will call the placer module to delete the task(s). It also checks every unit time if any tasks arrived and, if so, then it will try to place them immediately by calling the placer module, and if no placement was found, then it rejects the unplaced tasks. The yes-no name of the scheduler derives from the actions of either placing (yes) or rejecting (no) a newly arrived task.

We assume that the scheduler module has the following inputs. The arrived task array includes the tasks that have arrived at this time step. The executing task array includes all currently executing tasks. The output of the scheduler module is the executing task array.

On the other side, *FF_YNM_scheduler* tries to place a task as long as it has some laxity. In Figure 4-3, we describe the steps of this module. Again, as in *FF_YN_scheduler*, this module runs every unit time to remove tasks that have finished executing. Next, if any executing task finished, then it tries to place the tasks in the pending queue. Also, on every unit time the module checks if any tasks arrived and tries to place newly arrived tasks by calling the placer module. If no placement was found for a new task at the current unit time, then the module puts the task in the pending queue if it has laxity. When the task is added to the pending queue, this means the module will try to place it again when more space is available or until its laxity has expired. This module has the inputs and outputs of *FF_YN_scheduler* plus the pending queue.

When an executing task finishes (thereby freeing up space on the chip), the YNM scheduler attempts to place tasks from the pending queue in priority order. The pending queue is a data structure that orders waiting tasks based on some criterion, so it is not a FIFO queue. Possible criteria (orders) according to which the algorithm can retry to place these tasks include:

- ED: earliest deadline,
- LD: latest deadline,

- EA: earliest arrival,
- LA: latest arrival,
- SRL: shortest remaining laxity, and
- LL: longest laxity.

FF_YN scheduler:

```
Inputs: arrived task array, executing task array
Output: executing task array
Runs: every unit time
FF_YN_scheduler (arrived task array, executing task array)
        For all executing tasks v
                If a task v finished executing
                        Remove v from the executing task array
                        FF_placer (v,1)

        For all arrived tasks t do the following
                Remove t from the arrived task array
                FF_placer (t, 0)
                If placed = 1
                        Then add t to the executing array
                Else
                        reject t
```

Figure 4-2 FF_YN_Scheduler

In simulations, the ED criterion gave the best performance among them.

With the FF scheduler operating in YNM mode, the implementation is still easy as it runs the same placer as YN mode. In YNM mode, trying multiple times to place tasks in the pending queue will make it slower than FF_YN but at the same time will result in a better rejection ratio as the scheduler will not reject a task unless the task cannot satisfy its deadline.

In the following section we will describe a new scheduler idea that depends on splitting the chip into regions and deals with each region as if it is a different chip.

FF_YNM scheduler:

Inputs: arrived task array, executing task array, pending queue
Output: executing task array, pending queue
Runs: every unit time

***FF_YNM_scheduler* (*arrived task array, executing task array, pending queue*)**

```
finished = 0
For all executing tasks v
    If a task v finished executing do
        Remove v from the executing task array
        FF_placer (v, 1)
        finished =1

If finished =1
    For each task t in the pending queue
        if laxity(t) < 0
            Delete task t from the pending queue
        else
            FF_placer (t, 0)
            if placed = 1 then
                Delete t from the pending queue
                and add t to the executing task array

For all arrived tasks t do the following
    Remove t from the arrived task array
    FF_placer (t, 0)
    if placed = 1 then
        add t to the executing task array
    If placed = 0 and laxity (t) ≠ 0
        Then insert t in pending queue
    If placed = 0 and laxity (t) = 0
        Then reject t
```

Figure 4-3 FF_YNM_Scheduler

4.2. Regions-Based Method

Most of the work proposed in the area of scheduling real-time tasks on an FPGA deals with the chip as a whole. Therefore, a search for free space involves searching in a data structure that represents the entire chip. This way is time consuming and motivated our idea of dividing

the chip into regions to allow a reduction of the search time. We call the new idea as the *regions method*. The free space manager represents the space in each region separately from other regions. The placer searches by regions to find a placement, still using the FF technique for searching inside a region.

The regions-based idea derives from the problem of stacking boxes in an empty room. Stacking the same size boxes on top of each other efficiently uses the available space. In contrast, by stacking different sizes on top of each other the remaining space might be broken into many small pockets with no sufficiently large space for other boxes to be placed. The idea of our algorithm is to divide the room into regions of different sizes and assign each region to carry a certain box size. We call the size of a region as its *base*.

4.2.1. Description of the Regions-Based Idea

From this concept of stacking boxes, we decided to divide the chip into regions such that each region has a base (value). The base determines the width of a task that can be placed within this region. A task will be placed in the region that has a base corresponding to the task width. In this initial version of the method, we need to have regions with bases equal to all the expected task widths. Because of the chip size limitation and the variety of task sizes, it is not feasible to have all possible bases on one chip. To accommodate more widths, we permit a region to hold tasks with width equal to a small multiple of the base. For a region with base b, any task of width equal to b or multiple up to α of the base b will be placed in this region. Consequently, one of the design parameters is the value of α. For $\alpha = 3$, for example, only tasks with widths $\in \{7, 14, 21\}$ will be placed within a region with base 7. This still does not cover all task widths, so if the task width does not match any multiple up to α of a base, then our online scheduler adjusts the task width by adding some dummy columns to match the closest larger base multiple.

Figure 4-4 shows a 96 × 64 chip divided according to following bases ‹3, 6, 7, 7, 9› and α = 3, so the numbers of columns reserved for the bases are ‹9, 18, 21, 21, 27›. The borders between adjacent regions are all virtual borders with no physical divisions on the FPGA.

To explain the idea of readjusting the task size to match the regions we will use the base assignment of the Figure 4-4 example and explain in detail how this step is performed. Let the set of bases be {3, 6, 7, 7, 9} and α = 3. If a task t_k arrives with width w_k = 10, then the algorithm resizes the task to the first possible size allowed, which is 12 in base 6. The algorithm then tries to place it within the base 6 region with a width of 12. This process does not mean reshaping or changing physically the size of the task. It is a virtual increase in the size to match a base multiple, and the main idea behind adding these columns is to keep the remaining free space within the region as a multiple of the base.

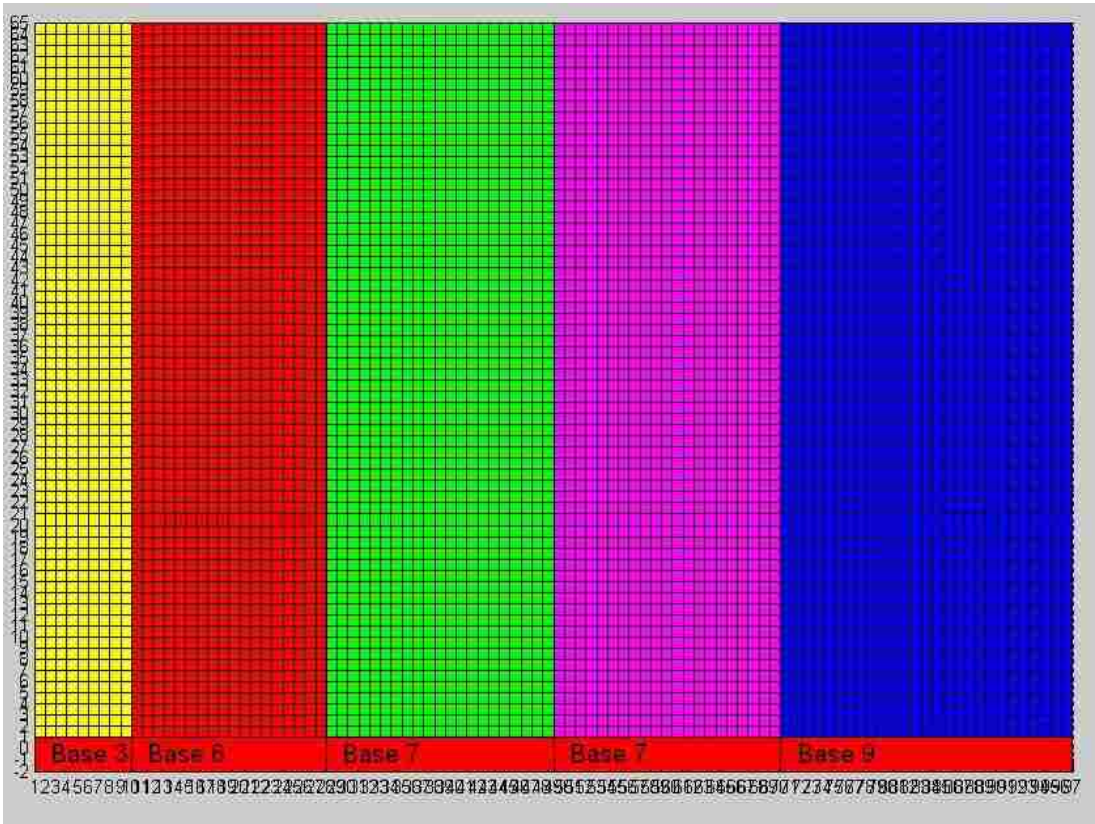


Figure 4-4 Empty chip with regions

In the above example, the maximum width that can be accommodated on the chip is 27. As the maximum task width expected might be more than what the largest region can accommodate (in the example, 27 columns), we have created an area called the *joker* area at the top of the chip, and it has some rows of the chip reserved exclusively for it. The joker area is mainly used for two purposes:

- accommodate tasks with width more than the largest region width, and
- accommodate tasks with width that can fit in a region but at the time of arrival there was insufficient free space for that task in the required region.

Figure 4-5 shows a chip with regions and with a joker area reserved at the top of the chip. Our regions-based online scheduler allows the joker exclusive use of its reserved area and also allows the joker to place tasks in the regions area. Basically the placer searches just a region area when it attempts to place a task in that region, but searches the entire chip (starting from the joker area) when it attempts to place a task in the joker area. If the task width is larger than what any region can accommodate, then the task is marked to be placed within the joker area. If the task width fits in a region, but the region has no space for it, then the placer attempts to place the task in the joker with original task width (no adjusting required).

In the regions-based method we assume that the bases are chosen before the scheduler starts and stay fixed till the last task finishes executing. The criteria on which the scheduler can decide on the bases can vary from one problem to another, but for best performance, bases should satisfy the following conditions:

- span all columns, that is, for a set of bases $B = \{b_1, b_2, \ldots, b_r\}$ and chip width w, for each base b_i the width reserved will be $\alpha \cdot b_i$ and

$$\sum_{i=1}^{r} \alpha \cdot b_i \leq w,$$

- cover as many expected task widths as possible, and
- cover widths consistent with task width distribution.

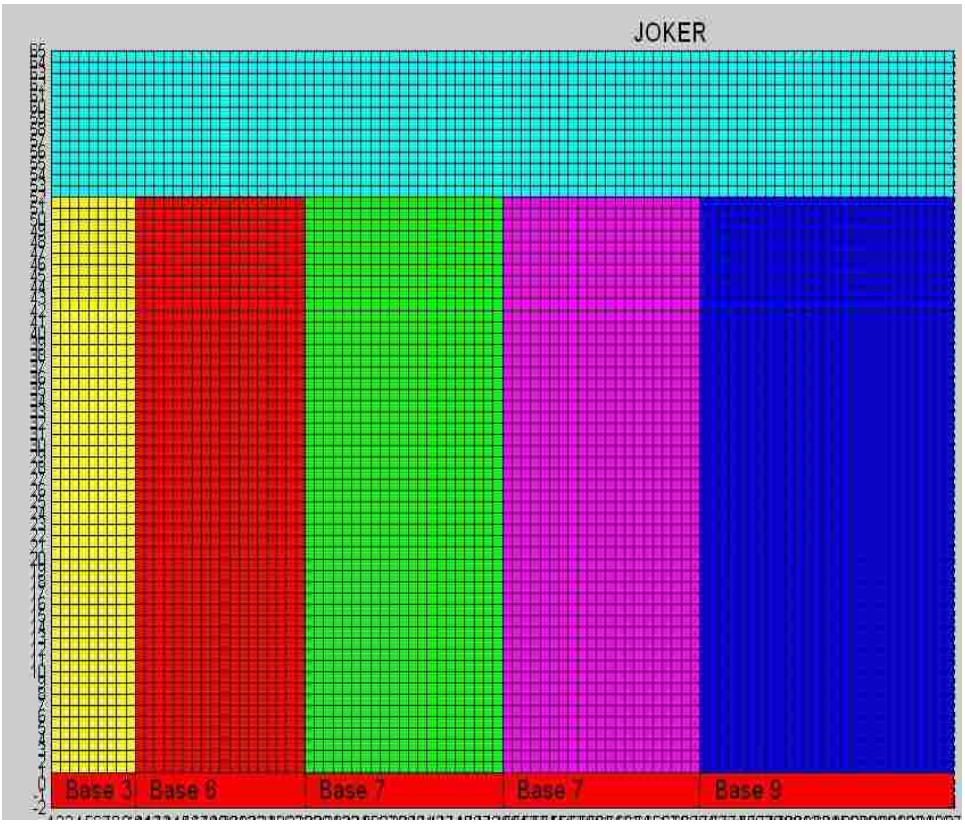


Figure 4-5 Empty chip with regions and joker area

One of the open problems for research in this idea is creating an algorithm to select the best set of bases as a function of FPGA size and expected incoming task widths. Also, changing the bases during the operation of the chip dynamically to be able to reduce the use of the joker area and to be able to use the regions to fit as many tasks as possible is another open problem.

4.2.2. Algorithm Steps

In this section we will describe the steps of our region-based scheduling algorithm. We describe first the placer's actions in attempting to place a task within an assigned region, whether

it is a newly arrived task or from the pending priority queue. Following that, we describe the higher level, which is the scheduling algorithm.

Consider a task t for which the scheduler has earlier assigned a base of b_k .

1. Search within base b_k*'s* region using the FF technique to find a placement for the task.
2. If no placement is found, then try to place the task within the joker area using FF starting from the top row of the joker region and proceeding downward.
3. If a placement was found in Step 1 or 2, then update the free space and skip Step 4.
4. No placement was found: if laxity remains for t, then insert t into the pending priority queue (according to the chosen ordering criterion); otherwise, reject t.

At a higher level, the scheduling algorithm must handle newly arrived tasks and tasks in the pending priority queue, assign regions to new tasks, and define an order for attempting to place tasks. This part of the code is responsible for assigning each task a region, adjusting the task width if needed. The steps are as follows.

Assume that we have partitioned the FPGA into r regions S_m, for $1 \leq m \leq r$, where b_m is the base of S_m. Let α denote the multiplier used to determine region width (so, for example, the width of the region reserved for S_m is $\alpha \cdot b_m$).

1. Check currently executing tasks and delete any task that has finished executing. Call the placer to update the free space data structure to free the space used by completed tasks.
2. If at least one task finished executing in Step 1, then for each task in the pending priority queue in order, execute the placement procedure.
3. If any new task t_i has arrived, then do the appropriate one of the following.

- If w_i equals $j \cdot b_m$ for some $1 \leq j \leq \alpha$ and some base value b_m, then assign t_i to the first such region S_m. Execute the placement procedure for new task t_i.
- If $w_i \neq j \cdot b_m$ for all b_m and $1 \leq j \leq \alpha$ but $w_i < \alpha \cdot b_k$ for some b_k, then adjust w_i to the next larger value of $j \cdot b_k$, where $1 \leq j \leq \alpha$, $1 \leq k \leq r$, and assign t_i to region S_k. Execute the placement procedure for new task t_i.
- Otherwise, if $w_i > \alpha \cdot b_m$ for all b_m, then execute the placement procedure for t_i starting from Step 2.

The algorithm works in YNM mode. In the simulation section we examine several possible ordering criteria for the pending priority queue (Figure 4-11). The structure of this algorithm follows that of FF_YNM in Figures 4-1 and 4-3 with the difference of dealing with regions instead of dealing with the whole chip at once.

In the above algorithms, we assumed that the placer found no placement for the task in the assigned region, then the placer will try to place it in the joker. In the simulations, we tried different versions of the algorithm. One of the versions is that if the placer did not find a placement for the task in the assigned region, then it will report to the scheduler that no placement is found and the scheduler readjusts the task size to match a different region and calls the placement procedure again to try to place the task in the new assigned region. Each time no placement is found, the placer reports to the scheduler. If the scheduler finds no placement in any of the stated number of regions it is allowed to try, then the scheduler will call the placement procedure to try to place the task within the joker area. Also, one of the versions we tested in our simulations was a chip with no joker area.

Figure 4-6 shows an example for a chip with some tasks placed in region areas and one task placed in the joker area. The task placed in the joker area has a width of 30 which is bigger

than what the biggest region can support (region base 9 with biggest width to support = 27). When this task arrived, the scheduler placed the task in the joker area. Figure 4-7 shows several tasks in the joker. Task number 85 has a width of 17 and at the time it arrived there was no placement available within the regions (region base 6), so the scheduler placed it in the joker area.

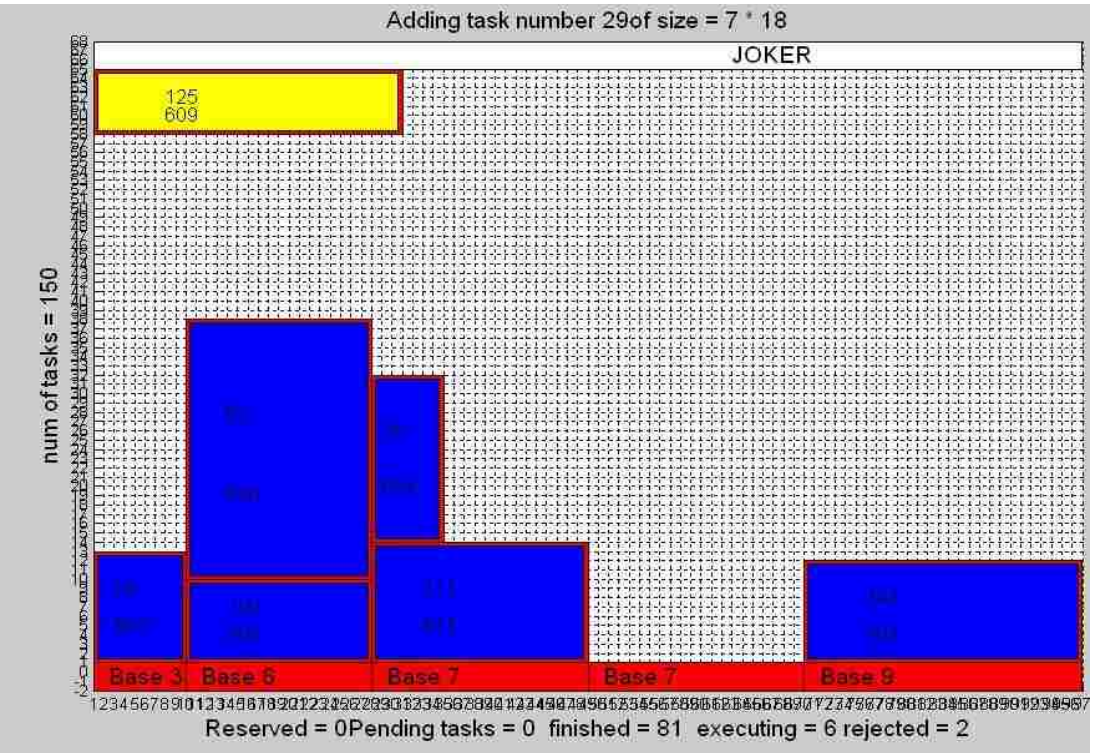


Figure 4-6 Regions example 1

4.3. Experimental Evaluation

In this section we will present the results of simulations comparing FF in YN mode, FF in YNM mode, and the regions-based method. We also compare different criteria for ordering tasks in the pending queue.

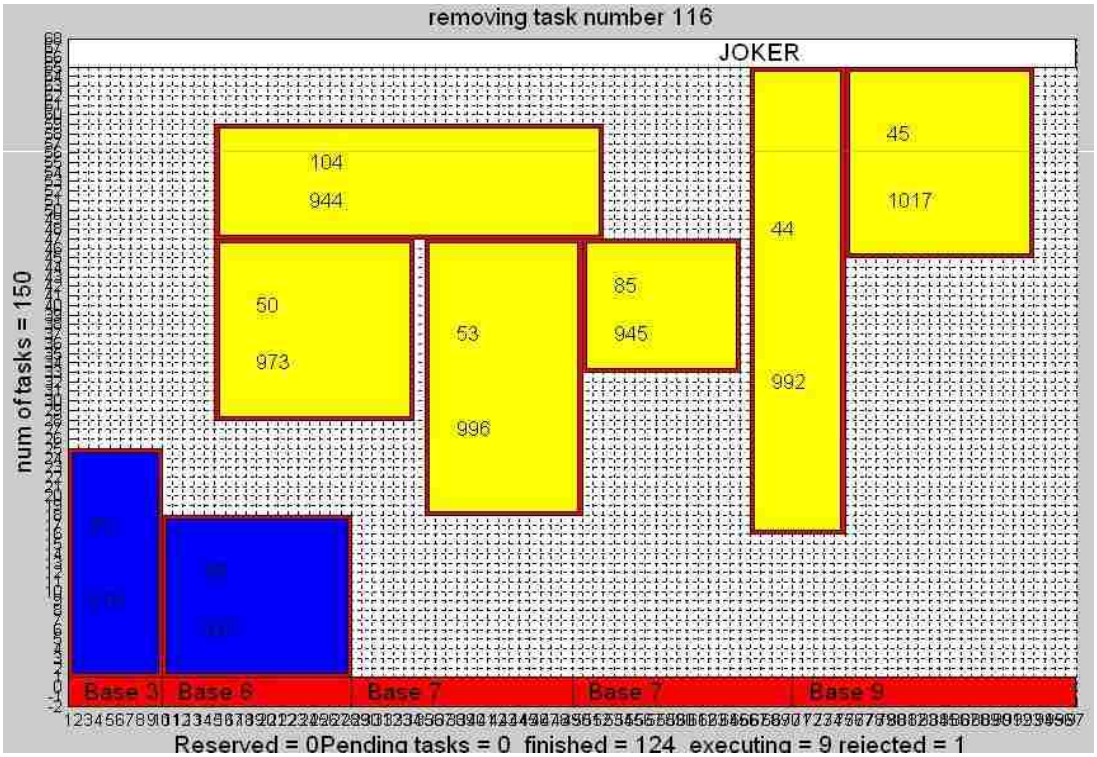


Figure 4-7 Regions example 2

4.3.1. Simulation Parameters

We performed simulations with sets of tasks with several parameters randomly generated and uniformly distributed over certain ranges. The following parameters and ranges have been used.

- The simulated device matches the size of the Xilinx XCV1000 FPGA, which consists of $96 \times 64 = 6144$ reconfigurable units. Also, we performed simulations for size 200×200.
- Task area is in the range of $[50, x]$ reconfigurable units, where $x \in \{100, 300, 500, 1000, 2000\}$.
- The aspect ratio used in generating the tasks is in the range of [0.2, 5]. Half of the generated tasks will be uniformly distributed in the range of [0.2, 1] and half will be uniformly distributed in the range of [1, 5].

- The arrival time is uniformly distributed over the period [1, t_{max}], where t_{max} is the latest time a task is expected to arrive.
- The execution time of each task is in the range of [5, 100] time units.
- Laxity is in the range [y, z] time units, where [y, z] $\in$ {[1, 50], [50,100], [100,200]}.
- The deadline of a task is the sum of its arrival time, execution time, and laxity.
- No information is known in advance about the maximum task width expected.

The mentioned parameters are the same simulation parameters used in the work done by Steiger *et al.* [SWPT03]. We will compare our results with theirs.

4.3.2. Performance Measure

As the problem we are interested in is online scheduling for a real-time set of tasks, we use the rejection ratio as a performance measure. We also use the chip load [SWPT03] as a way to measure how much demand a task set places on the FPGA in each simulation. The chip load is a function of the area time product for each task:

$$chip\_load = \frac{\sum_{all\_tasks} w_i \cdot h_i \cdot e_i}{w \cdot h \cdot t_{\max}},$$

where

- w_i , h_i are the width and the height of task i,
- e_i is the execution time of task i,
- w is the chip width,
- h is the chip height, and
- t_{max} is the latest time a task is expected to arrive.

The simulations will measure the rejection ratio for different chip loads. We are interested in chip load in the range [0.3, 1.2] because chip loads more than that will not actually indicate the

performance of the code as the chip is overly loaded. To cover the chip load in which we are interested, the simulations vary chip load range by changing the number of tasks under simulation. We had several runs at different chip loads, in other words, at different numbers of tasks. To evaluate the performance of the online scheduler, we performed 210 simulations with 30 simulations for each of the following seven different numbers of tasks: 150, 200, 250, 300, 350, 400, and 500. Because the task parameters are randomly distributed, the chip load does not have to be exactly the same for the same number of tasks. To evaluate the performance of the scheduler, we average rejection ratio for any run that has a chip load between 0.65 and 0.75, for example, and report it as the rejection ratio at chip load 0.7. For each range of chip load, we do the same. Also, the simulations will vary the value of t_{max} according to different area classes used to keep the chip load in the same range.

4.3.3. Simulation Results

In this section we report different results for regions-based schedulers varying the simulation parameters explained in the previous section. We examine two existing methods for comparison with our work. The first one is the basic FF scheduler and the second one is the scheduler designed by Steiger *et al.* [SWPT03]. The regions, FF_YN, and FF_YNM schedulers have been implemented with Matlab. The results for the work done by Steiger *et al.* were taken directly from their paper. All the codes have been tested with the same simulation parameters.

Tables 4-1 through 4-7 summarize representative simulation results. Each entry in these tables shows average rejection ratio over the 210 simulations across a range of chip loads on a 96 × 64 FPGA. The set of parameter values in each row of regions simulations indicates the number of rows exclusively allocated to the joker, the number of regions the placer will try before checking the joker, the multiple α of base value to fix region width, and the set of bases. The

results reported in this table assumed earliest deadline (ED) as the criterion on which the pending queue is sorted. At the end of the chapter we will show the effect of using different criteria for sorting the pending queue.

In Table 4-1, experiment 2 (E2) displays results for the core set of parameter values: 14 rows reserved for the joker, the placer tries one region, α = 3, and set {3, 6, 7, 7, 9} of bases. Compared to FF_YN (E1), E2 shows substantial improvements in rejection ratio for each task area range. Figure 4-8 details this comparison of E1 and E2 across chip loads for the area range [50, 500].

Table 4-1 Average rejection ratio for FF_YN and regions-based

different area, laxity = 1:50, chip 96 × 64				
task area range		50:100	50:300	50:500
E1	FF_YN	0.084	0.141	0.214
E2	joker rows = 14, try = 1, α=3, bases = {3, 6, 7, 7, 9}	0.037	0.081	0.143

Experiments in Table 4-2 show the effect of trying different regions to find a placement for the arrived task. In this case the placer will try the first assigned region and if it finds no placement, the placer will try to place it in other regions after readjusting the size of the task. In Table 4-2, "*try*" indicates the number of regions the placer is allowed to try to find a placement. In experiments E2, E8, and E3, we exclusively reserved 14 rows for the joker area and we allowed the placer to try 1, 2, and 4 regions, respectively. The results in these three experiments produce very little change in the rejection ratio. In experiments E9, E10, and E5 we canceled the joker area completely so the placer places tasks only in the regions (note: tasks with width larger than what a region can accommodate will be rejected), and we allowed the placer to try 2, 3, and 4 regions, respectively. The results in these three experiments show some enhancement in performance when trying more regions to place tasks. Also, by comparing the results in E2, E8,

and E3 with E9, E10, and E5, we can conclude that using the joker area is dominant over trying multiple regions.

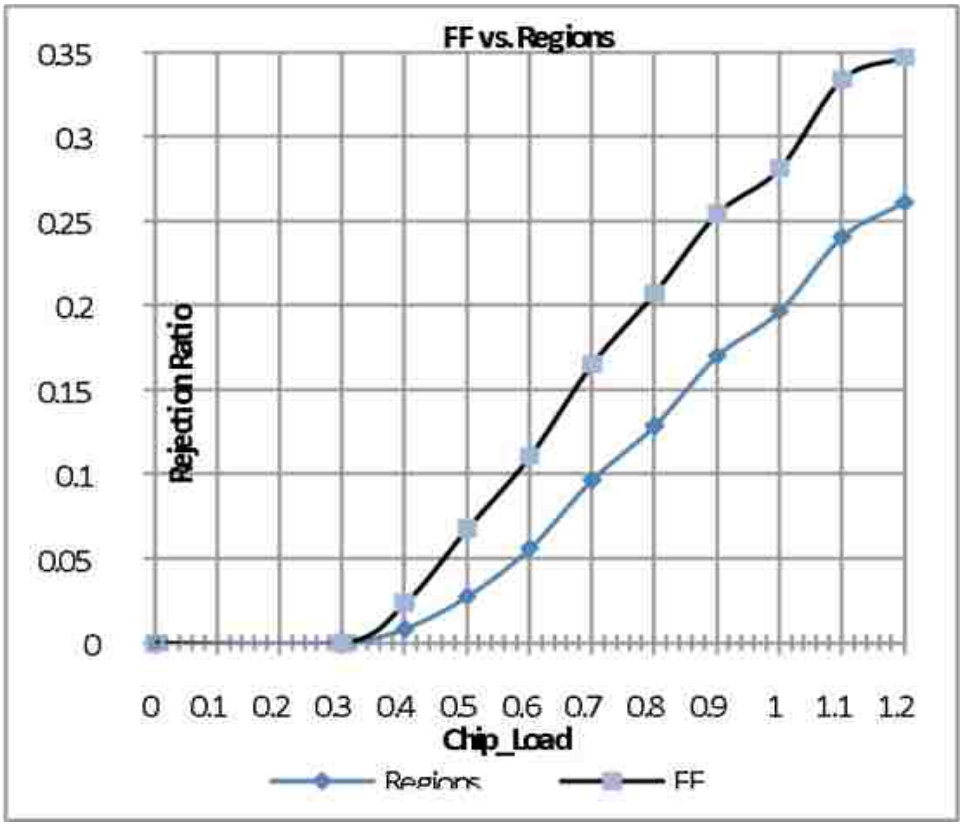


Figure 4-8 Rejection ratio vs. chip load for FF_YN (E1) and regions method (E2) with area range [50,500].

Table 4-2 Effect of trying multiple regions

different area, laxity = 1:50, chip 96 × 64				
task area range		50:100	50:300	50:500
E2	joker rows = 14, try = 1, α=3, bases = {3, 6, 7, 7, 9}	0.037	0.081	0.143
E8	joker rows = 14, try = 2, α=3, bases = {3, 6, 7, 7, 9}	0.038	0.084	0.145
E3	joker rows = 14, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.049	0.088	0.150
E9	joker rows = N/A, try = 2, α=3, bases = {3, 6, 7, 7, 9}	0.092	0.185	0.308
E10	joker rows = N/A, try = 3, α=3, bases = {3, 6, 7, 7, 9}	0.053	0.172	0.294
E5	joker rows = N/A, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.054	0.170	0.295

Table 4-3 shows the effect of using the joker area. To show only the effect of the joker area, we fixed the following parameters: bases {3, 6, 7, 7, 9}, the placer tries four regions before trying to place in the joker, and α = 3. In experiments 3 to 5 (E3 to E5), we changed the number of rows exclusively reserved for the joker area between 14, 0, and no joker at all. From the results shown in the table, not using a joker at all (E5) has a large performance penalty because any task wider than the widest region will always be rejected. Not using a joker is an option only if all tasks fit within some region. Also, the table shows that in the case of task area 50:100 (relatively small tasks) the use of the joker produces little change in the results (E3 and E4 vs. E5). Also, from the results shown in the table, changing the number of exclusively reserved rows produces little change (E3 vs. E4).

Table 4-3 Effect of using different joker area sizes

different area, laxity = 1:50, chip 96 × 64				
task area range		50:100	50:300	50:500
E3	joker rows = 14, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.049	0.088	0.150
E4	joker rows = 0, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.051	0.091	.0155
E5	Not using joker rows = N/A, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.054	0.170	0.295

Experiments in Table 4-4 show the effect of using different sets of bases. In this set of experiments the placer will try to place the task in all regions before trying to place it in the joker area. Also, we reserved 14 rows exclusively for the joker area. The value of α was adjusted for different bases so that the regions span all FPGA columns. These results indicate little sensitivity to the exact choice of bases. Knowledge of a specific task set, however, can profitably guide base assignment.

Table 4-5 illustrates that, for a larger chip (200 × 200), in experiment E12, the regions method greatly enhances performance when the joker was used. In experiment E13, the scheduler was tested without a joker at all and performance is almost the same as E11 even though in this case some tasks were rejected because their sizes are larger than what the regions can accommodate.

Table 4-4 Effect of using different sets of bases

different area, laxity = 1:50, chip 96 × 64				
task area range		50:100	50:300	50:500
E3	joker rows = 14, try = 4, α=3, bases = {3, 6, 7, 7, 9}	0.049	0.088	0.150
E6	joker rows = 14, try = 4, α=3, bases = {6, 7, 8, 11}	0.050	0.088	0.148
E7	joker rows = 14, try = 7, α=2, bases = {3, 4, 6, 7, 8, 9, 11}	0.044	0.084	0.145

Table 4-5 Average rejection ratio with chip size 200 × 200.

area = 50:500, laxity = 1:50, chip 200 × 200		
E11	FF_YN	0.084
E12	joker rows = 0, try = 7, α =3, bases = {6, 7, 8, 9, 11, 12, 13}	0.011
E13	not using joker, try = 7, α=3, bases = {6, 7, 8, 9, 11, 12, 13} (does not cover all task sizes)	0.085

We also compared the regions method to the algorithm of Steiger *et al.* [SWPT03]. Table 4-6 compares the rejection ratio at several chip loads for E2 against results from their work.

Table 4-6 Rejection ratio at different chip loads compared to Steiger *et al.* [SWPT03].

area = 50:500, laxity = 1:50, chip 96 × 64				
chip load		0.5	0.75	1.0
E2	joker rows = 14, try = 1, α=3, bases = {3, 6, 7, 7, 9}	0.02	0.09	0.19
	Steiger *et al.* [SWPT03]	0.10	0.20	0.27

Table 4-7 illustrates that the performance of FF_YNM (E14) is better than FF_YN (E1). Figure 4-9 shows the performance of FF_YNM using the earliest deadline ordering criterion (FF_YNM_ED) compared to FF_YN. At chip load = 1, the rejection ratio of FF_YN is approximately 0.28 and the rejection ratio of FF_YNM_ED is approximately 0.19 which indicates a reduction in the rejection ratio by about 36%. The results in Figure 4-9 show the importance of retrying tasks that were initially rejected.

Table 4-7 Rejection ratio at different chip loads for FF_YNM, FF_YN, and regions

area = 50:500, laxity = 1:50, chip 96 × 64				
chip load		0.5	0.75	1.0
E2	joker rows = 14, try = 1, α=3, bases = {3, 6, 7, 7, 9}	0.02	0.09	0.19
E14	FF_YNM	0.02	0.09	0.19
E1	FF_YN	0.07	0.17	0.28

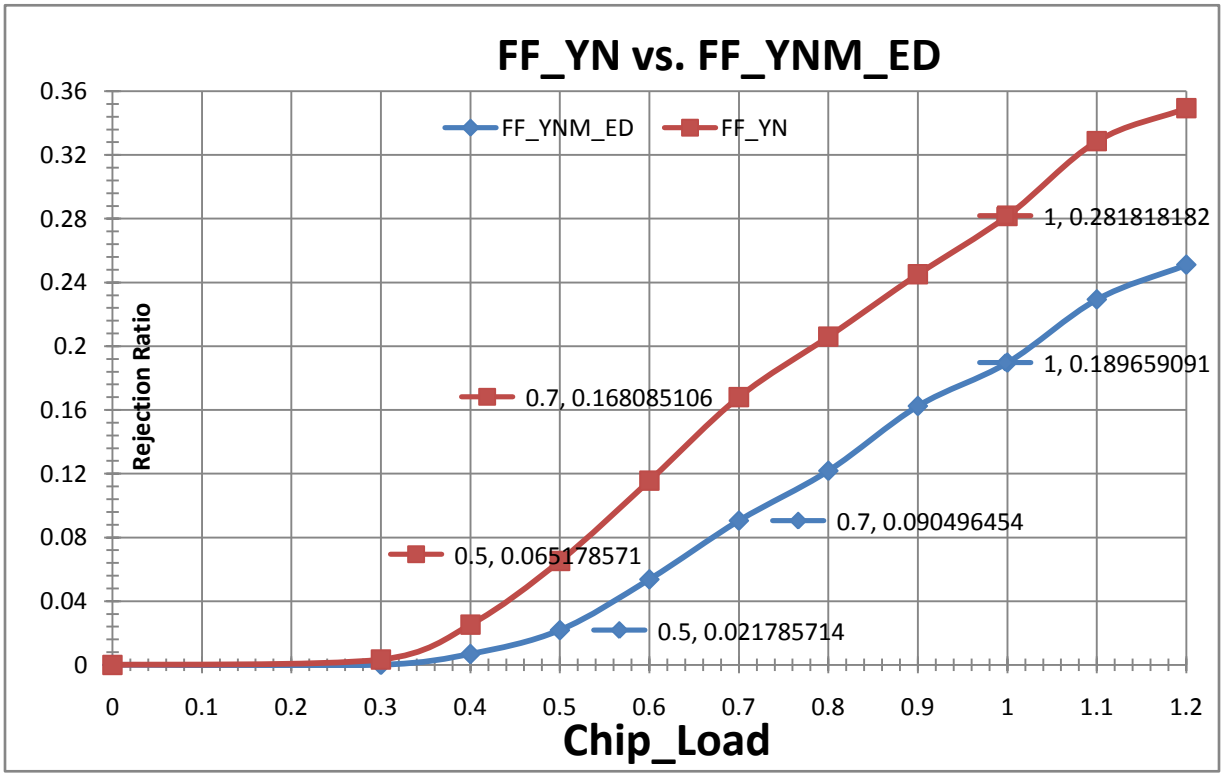


Figure 4-9 FF_YN vs. FF_YNM_ED

Figure 4-10 compares the performance of the regions-based method to the FF_YNM algorithm. Figure 4-10 shows the comparison between them using the earliest deadline ordering criterion; results with other ordering criteria are comparable. From Figure 4-10 we can see that the two methods return nearly identical results. Even though the regions-based scheduler achieves the same rejection ratio as FF_YNM, the regions-based idea has other features that make it better than FF_YNM. As described above, the regions-based scheduler has less search time as the algorithm does not have to search the whole chip to check if there is free space for the task or not.

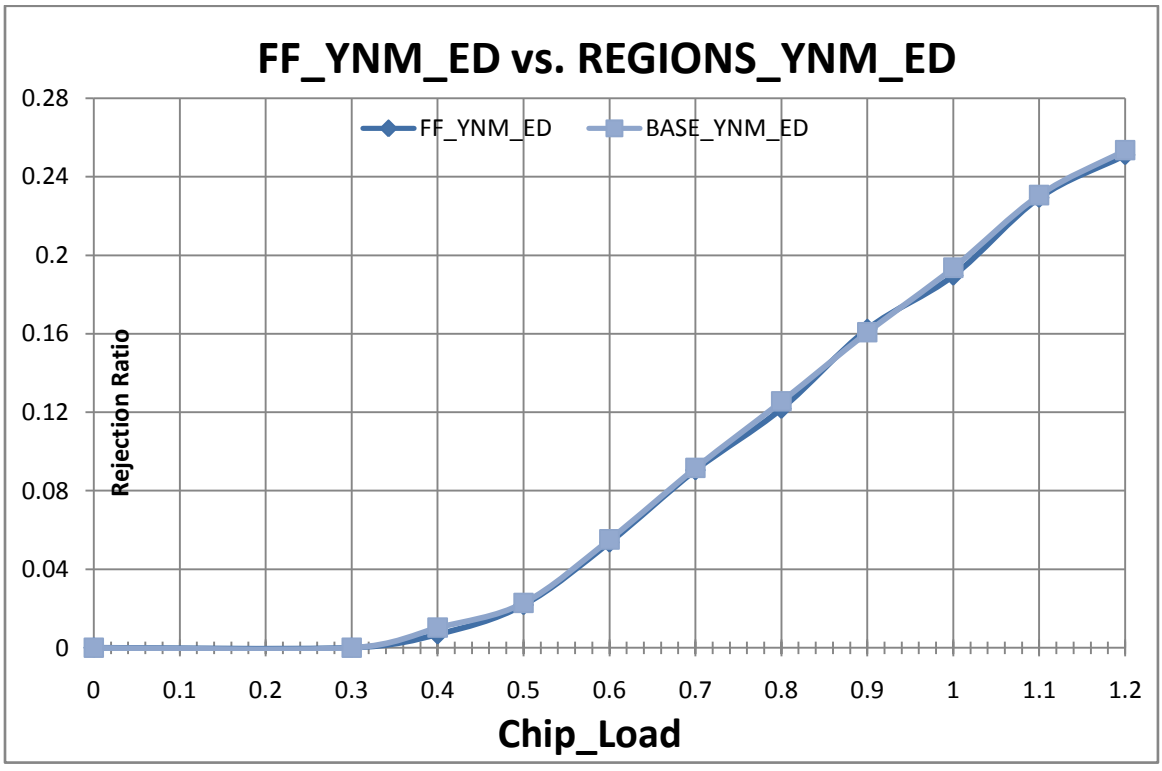


Figure 4-10 REGIONS_YNM_ED vs. FF_YNM_ED

Figure 4-11 shows the performance of the regions method under different criteria for sorting the pending queue. We have tried the following: Latest Arrival (LA), Latest Deadline (LD), Longest Laxity (LL), Earliest Arrival (EA), Earliest Deadline (ED), and Shortest Remaining Laxity (SRL). From Figure 4-11 we can see that earliest deadline (ED) has the best

performance among all of them and that is why all the results in this section were reported based on the ED criterion.

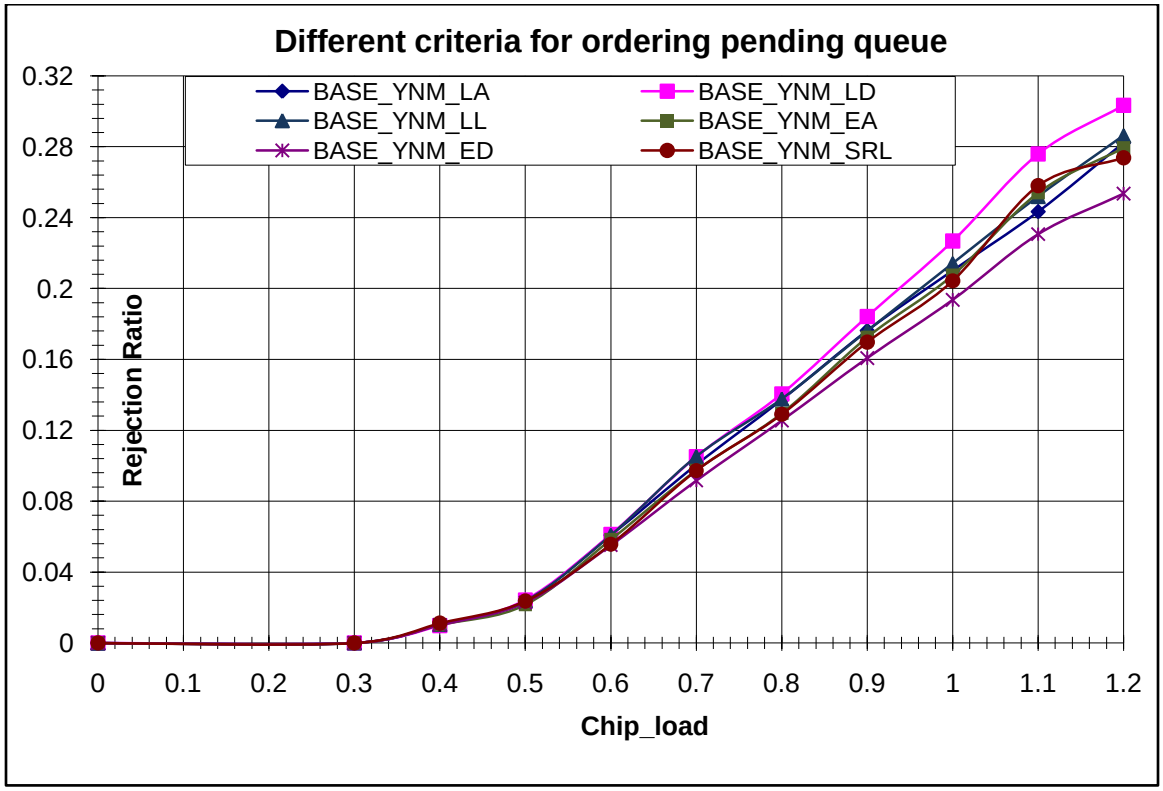


Figure 4-11 Different criteria for sorting the pending queue

CHAPTER 5: FREE SPACE MANAGERS

To schedule a set of tasks in a small amount of time, the scheduling algorithm must manage the free space efficiently. A data structure to represent free space should allow the scheduler to identify free space in which to place a new task and admit efficient updates after placing or removing a task. In this part of our research, we review some existing data structures and analyze their time complexity. We propose a new structure using maximal horizontal and vertical strips to represent the free space. These strips start and stop at task boundaries. Simulation and time analysis showed that this method has better time complexity than many other free space data structures and at the same time has a very reasonable rejection ratio on real-time tasks compared to other methods.

In this chapter we will describe some of the data structures used in the area by other researchers. The bit matrix is one of the well known data structures used in the area, with a bit for each CLB representing whether that CLB is occupied or free. Instead of representing each CLB individually, the free space manager can represent a group of empty CLBs as one entry in the data structure. A group of empty CLBs can be represented as a rectangle called an empty rectangle. There are different types of these rectangle based on the CLBs they represent. Empty rectangles can overlap, sharing some CLBs, or they can represent disjoint sets of CLBs. Empty rectangles were first used by Bazargan *et al.* [BKS00] in the problem of scheduling on RC devices. They used two different types of empty rectangles, one of which is the maximal empty rectangle (MER) and the other is non-overlapping empty rectangles. In this chapter we explain the two types of empty rectangles and compare them from different aspects. We show some of the problems in creating empty rectangles or updating them after inserting or deleting a task. Also, we introduce an algorithm to update the MERs after task insertion and task deletion, and

we compare the time complexity of our update algorithm to the time complexity of other update algorithms presented by researchers who used MERs in their research. At the end of our research about empty rectangles, we will show that creating these rectangles every time we need them from scratch takes less time than keeping them updated after any change on the chip.

In the next chapter we will introduce another data structure that we used to solve the problem of keeping information about the free space on the chip. The idea behind this data structure is to use horizontal and vertical strips to represent the free space on the chip. The maximal horizontal and vertical strips (MHVS) used by a scheduler showed similar performance compared to other data structures but in much less time to generate them and keep them updated every time the space on the chip changes. At the end of that chapter, we will compare these different data structures.

Free space data structures vary in the time needed to search for a placement, time needed to update the data structure after task deletion or insertion, ease of implementation, and the memory needed to store the information of the free space on the chip. These measures are what we will use in our research to compare different data structures.

5.1. Bit Matrix

A bit matrix that is a 2D array of w columns and h rows can directly represent a $w \times h$ 2D reconfigurable area. Each entry in the matrix represents an RCU. If an RCU is occupied by a task (free), then the corresponding entry in the bit matrix is 1 (0). With a bit matrix representation of free space, when the scheduler calls the placer to place a task t with size $c \times r$, the placer searches in the bit matrix for a sub-matrix of size bigger than or equal to $c \times r$ with all zero elements. After placing the task, the placer changes the value of each element in this sub-matrix to be 1. When a task finishes executing, the placer updates the bit matrix by changing all the elements

that represent the area occupied by the task to be zeros. Using a bit matrix is very easy and straightforward but takes a long time to find a placement. Updating the data structure after a task insertion or a task deletion is straightforward but yet takes time.

The time required to search the bit matrix for a placement of size $c \times r$ in the worst case is $O(whcr)$. The update time after each placement or deletion of a task requires $O(cr)$ time. The time complexity of the placer based on bit matrix representation will be $O(whcr)$. The time complexity for using the bit matrix is related to the chip size and the task size. In other data structures that will be analyzed or proposed in this research the data structures depend on the number of executing tasks on the chip, so, we will represent their time and space complexities as functions of n, where n is the number of executing tasks on the chip. Based on simulations with $w = 96$ and $h = 64$ and other parameters as in Section 4.3.1, we have found that the chip might have up to 25 tasks running at the same time. Based on this, the time complexity of using bit matrix representation is greater than $O(n^2)$. Several schedulers discussed earlier [ABBT04, CDHG07, HV04-2, WSP03] use a bit matrix as their free space data structure or employ a variation of the same size to maintain a data structure, at significant time cost. From an implementation point of view, the bit matrix is the easiest data structure to implement. A bit matrix requires $O(wh)$ space to store the information for the free space.

5.2. Empty Rectangles

Empty rectangles representation is a common way to represent the free space on an FPGA. Empty rectangles were applied to represent the free space on the FPGA by Bazargan *et al.* [BKS00]. Instead of dealing with the free space on an FPGA by individual CLBs, the free space can be combined together to form rectangles of free CLBs. So the data structure

representing the free space consists of a set of empty rectangles with the condition that every CLB inside a rectangle is unused.

Using empty rectangles representation, when a task arrives, the scheduler searches the set of empty rectangles for a rectangle that has a width greater than or equal to the task's width and a height greater than or equal to the task's height. So, a task t with width c and height r will be placed in a rectangle m with a width y and a height x only if $y \geq c$ and $x \geq r$. After placing a task, the scheduler updates the free space accordingly by removing rectangle m from the data structure. As the task may not use the whole space of the rectangle, then the scheduler has several options to deal with the remaining space. An easy approach will be splitting the remaining space into rectangles and adding them to the set of rectangles after removing the original rectangle from the set. Or, a more complicated approach would be to try to merge the remaining space to some of the existing rectangles in the set of empty rectangles to form a bigger rectangle. Later in this section we will discuss ways of updating the empty rectangles after placing or deleting a task from an FPGA. The time needed to search the empty rectangles set is less than searching a bit matrix as in empty rectangles we are dealing with group of CLBs (empty rectangle) instead of dealing with individual CLBs.

The work of Bazargan *et al.* [BKS00] is one of the major contributions in using empty rectangles as a way of free space representation. They employed two different types of empty rectangles: non-overlapping rectangles and overlapping rectangles. In the next sections will give details on how to use them and how to update them and also we will introduce some of our contributions in enhancing the performance of the update process for empty rectangles.

5.2.1. Non-Overlapping Empty Rectangles

Bazargan *et al.* [BKS00] proposed the idea of representing free space as a set of non-overlapping rectangles. In their non-overlapping rectangles representation, the free space can be represented by $O(n)$ rectangles, where n is the number of active tasks. A placer uses $O(n \log n)$ space to store the empty rectangles and $O(\log n + k)$ time to check which empty rectangle can accommodate a task, where k is the number of reported candidate rectangles. Figure 5-1 shows an example for non-overlapping rectangles.

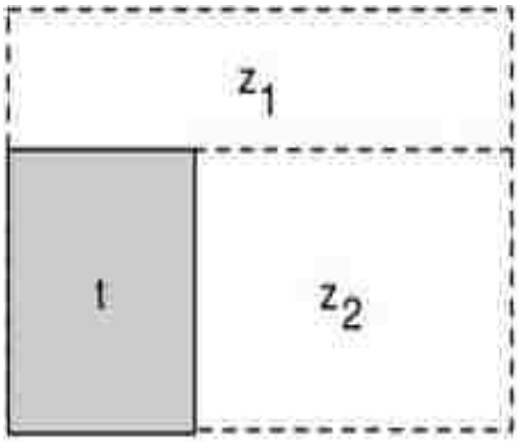


Figure 5-1 Non-overlapping rectangles

Steiger *et al.* [SWPT03] enhanced the non-overlapping rectangle representation of Bazargan *et al.* [BKS00] to permit limited overlaps. Bazargan *et al.*'s placer uses heuristics to decide whether to split a free space rectangle vertically or horizontally upon task placement. The key to Steiger *et al.*'s enhancement is to delay the decision of which way to split the free space rectangle until a new task arrives that can use this space.

5.2.2. Overlapping Maximal Empty Rectangles

Another type of empty rectangle is the Maximal Empty Rectangle (MER). In a chip that has n running tasks, a *maximal empty rectangle* is an empty rectangle that is not contained by

any other empty rectangle and does not overlap any of the *n* running tasks (see Figure 5-2 for examples).

Note that a non-overlapping rectangle does not have to be maximal, and this is one of the main differences between the two representations. In Figure 5-1, for example, z_1 is maximal but z_2 is not maximal.

The placer manipulates the MER data structure when searching for free space, inserting new tasks, and deleting tasks that finished executing. Figure 5-2 shows an example of a set of MERs and how that set changes with a task insertion and a task deletion. Figure 5-2(a) shows a set of MERs when a chip has three executing tasks. Figure 5-2(b) shows the update after placing a new task (T_4), and Figure 5-2(c) shows the update after deleting task T_2.

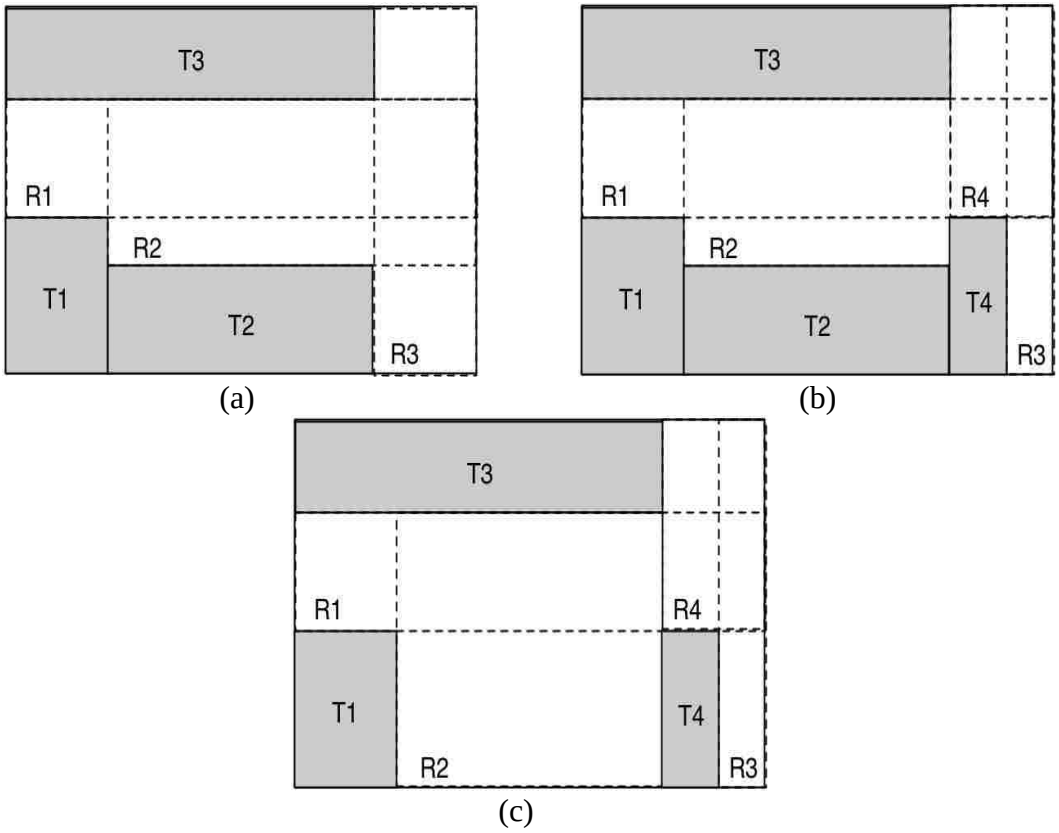


Figure 5-2 (a) Example of a set of MERs, (b) updated after inserting task T4, and (c) updated after deleting task T2.

Several other researchers in the area have studied MERs for different applications like VLSI layout. Overmars and Wood [OW88] presented an algorithm to compute the MERs given a set of n points. Also, Orlowski [O90], Nandy *et al.* [NBR90], and Datta [D92] developed methods to construct all MERs given n tasks (or rectangles or points) that run in $O(n \log n + d)$ time, where d is the number of MERs. In the worst case, the set of MERs includes $O(n^2)$ rectangles, while the expected number of MERs is $O(n \log n)$ for a random distribution of tasks [NLH84]. Nandy *et al.* [NBR90] utilized maximal horizontal strips in their algorithm to create MERs.

Bazargan *et al.* [BKS00] were among the first researchers in the area to use MERs in representing the free space on a reconfigurable device. They called this method as Keeping All Maximal Empty Rectangles (KAMER). They also stated that the time to search for a free rectangle to accommodate a task is $O(n^2)$. Chiu and Chen [CC99] asserted that the time complexity to insert a task and update the KAMER structure is $O(d^2)$, where d is the number of MERs, and that the time complexity to delete a task and update the KAMER structure is $O(d^2)$. (Zhou *et al.* [ZWHP06] identified a similar time complexity.) In the worst case d is $O(n^2)$ so, in the worst case, then, the time complexity to update the MERs is $O(n^4)$.

Time complexity is one way to compare different types of free space data structures. We can also compare them based on if the data structure is recognition-complete. A structure is recognition-complete if it is able to recognize all free rectangles on the reconfigurable device. This feature may affect not just the time complexity but also will affect the performance of the scheduler as a data structure that is not recognition complete may not represent every free region on the chip, and this can result in rejecting a task even though there is a space for it on the chip. A bit matrix deals with each CLB on the chip, and this means that the bit matrix can detect any

free region on the chip. So, a bit matrix is a recognition-complete data structure. Also, The MER free space data structure is recognition-complete. The time to search or update MERs, however, is relatively high compared to other methods that we will discuss in the next chapter. As far as the performance, the MER proved to have a better performance than non-overlapping rectangles by recognizing more free regions on the chip. Free space stored as non-overlapping rectangles is not recognition-complete as the scheduler may not be able to detect all free regions even though the free space manager covers all free CLBs on the chip. For example, in Figure 5-1 if a task u arrives with a height equal to the sum of the height of z_1 and the height of z_2 and a width smaller than the width of z_2, then u will be rejected even though sufficient free space is available because it does not fit in z_1 or z_2.

One of the main problems associated with MERs is how to update them after inserting or deleting a task. In the next section we will represent an update algorithm and analyze its time complexity and compare this to a very basic thought which is creating MERs from scratch every time we insert or delete a task.

5.2.3. Updating MERs after Insert Action

In this section we will present a new method to update the set of all MERs. The worst case analysis for our method and methods presented before from other researchers, such as Chiu and Chen [CC99], have the same time complexity. The method presented in this section will exploit the MER Task Insertion Theorem and MER Task Deletion Theorem below.

Let R denote the set of all MERs in an FPGA. Let q denote a rectangle selected in which to place a new task. Assume that at least one rectangle in R completely contains q.

MER Task Insertion Theorem. The only MERs that may change when task t is inserted in rectangle q are the MERs that overlap with q.

Proof: Let R denote the set of MERs in the FPGA before placing t. Each MER in R that overlaps the placement of t obviously must change because the space occupied by t is no longer empty. Consider now MER $z \in R$ such that z does not overlap the placement of t. Since z was maximal before placing t and since placing t does not add any free space to the FPGA, then z cannot grow in size (because tasks and boundaries that determined its limits have not changed) and z cannot shrink (because it does not overlap with t). □

Let g denote a rectangle representing the space freed after a task t has finished and we want to incorporate g into R. Rectangle g is disjoint from R, that is, for all $h \in R$, $h \cap g = \varnothing$.

MER Task Deletion Theorem: The only MERs that may change when a task t is deleted, freeing a rectangle g, are the MERs that share part of a border with g.

Proof: Let R denote the set of MERs in an FPGA before deleting t. Let g denote the rectangle freed by the deletion of t. Rectangle g is disjoint from R. If no MERs touch g, this means that g is a MER. If some MERs in R touch g, then those MERs may be able to expand to incorporate some or all of the free space of g. Consider now MER $q \in R$ such that q does not touch g. Rectangle q was maximal before the deletion of t with q's limits determined by surrounding tasks (other than t) and FPGA boundaries. These conditions did not change with the deletion of t, so q cannot grow in size, and, of course, q cannot shrink because the deletion of t did not remove any free space in q. □

Update methods based on the above theorems require maintaining structures that indicate for each MER the set of MERs that overlap with it and the set of tasks that share a border. The MER Task Insertion Theorem states that the only MERs that can change after a task insertion are those that overlap q. Therefore, the algorithm maintains a graph describing the overlapping of MERs to be able to efficiently update, delete, and recognize the affected MERs. As the graph

will represent the relations among MERs and running tasks, we will call it a *relation graph*. The graph has five types of relations between tasks and MERs. In the relation graph we may have the following:

- tasks touching other tasks,
- tasks touching MERs,
- MERs touching tasks,
- MERs touching other MERs, and
- MERs overlapping other MERs.

We represent each of the five relations as an edge in the relation graph. If task *t* touches rectangle *z*, then the graph has an edge from *t* to *z* representing that *t* touches *z*. Also, it has an edge from *z* to *t* representing that *z* touches *t*. We store the graph as adjacency lists. For example, for each task *t* we have a list representing all tasks touching task *t* and a list representing all MERs touching *t*. Also, for each MER *z*, we have a list representing all tasks touching *z*, a list representing all MERs touching *z*, and a list representing all MERs overlapping with *z*. The following notation denotes the adjacency lists for the five different relations.

- *Adj_touch_task* (*r*) : adjacency list representing all tasks touching MER *r*,
- *Adj_touch_MER* (*r*) : adjacency list representing all MERs touching MER *r*,
- *Adj_overlap_MER* (*r*) : adjacency list representing all MERs overlapping MER *r*,
- *Adj_touch_task* (*t*) : adjacency list representing all tasks touching task *t*, and
- *Adj_touch_MER* (*t*) : adjacency list representing all MERs touching task *t*.

If task *t* touches rectangle *r*, then *t* is be in *Adj-touch-task*(*r*) and *r* is in *Adj-touch-MER*(*t*). Also, we have a pointer that shows where *t* is in *Adj-touch-task*(*r*) and a pointer that

shows where *r* is located in *Adj-touch-MER*(*t*). By using the pointers we will be able to delete *t* from *Adj-touch-task*(*r*) in constant time. The same style of pointers works for all other relations.

Algorithm 5-1 updates the set of MERs by using the relation graph, and the algorithm also maintains (updates) the relation graph. The following paragraphs explain Algorithm 5.1, cutting the algorithm into phases and explaining each phase separately. At the end of each phase of the code we will discuss the time complexity for each phase and state the total time complexity at the end of discussing all phases.

The inputs to the algorithm are the new placed task *t*, rectangle *r* in which *t* was placed, the set of MERs, and the relation graph. The output of the algorithm is an updated MERs set and an updated relation graph.

Algorithm 5-1 updating MERs after inserting task *t*.

Inputs: task *t* parameters, MER *r*, set of MERs, relation graph

Output: updated set of MERs, updated relation graph

% Add a new node in the graph for *t* and create an empty adjacency list for it %

1: ***Adj_touch_MER* (*t*) ← Ø**

2: ***Adj_touch_task* (*t*) ← Ø**

The first phase (Steps 1 - 2) adds a new node in the relation graph to represent *t* and creates adjacency lists. This phase takes constant time.

% updating the relation between *t* and the current executing tasks %

3: For each task *u* in *Adj_touch_task* (*r*)

4: If *t* touches *u*

5: ***Adj_touch_task* (*u*) ← *Adj_touch_task* (*u*) ∪ *t***

6: ***Adj_touch_task*** **(*t*)** $\leftarrow$ ***Adj_touch_task*** **(*t*)** $\cup$ ***u***

7: END

8: END

The next phase (Steps 3 - 8) updates the relations between *t* and the current executing tasks. In Step 4, the algorithm checks if two tasks are touching. The touching check for tasks or rectangles executes in constant time by comparing the coordinates of the four borders of the first rectangle to the coordinates of the four borders of the second rectangle. As the number of executing tasks is *n*, so in the worst case *Adj_touch_task* (*r*) have a maximum of *n* tasks. This phase (Steps 3 -8) in the worst case takes $O(n)$ time.

% updating the relation between *t* and the existing MERs %

9: For each MER *g* $\in$ **{*Adj_overlap_MER*(*r*)** $\cup$ ***Adj_touch_MER*(*r*)}**

10: If *t* touches *g*

11: ***Adj_touch_MER*** **(*t*)** $\leftarrow$ ***Adj_touch_MER*** **(*t*)** $\cup$ ***g***

12: ***Adj_touch_task*** **(*g*)** $\leftarrow$ ***Adj_touch_task*** **(*g*)** $\cup$ ***t***

13: END

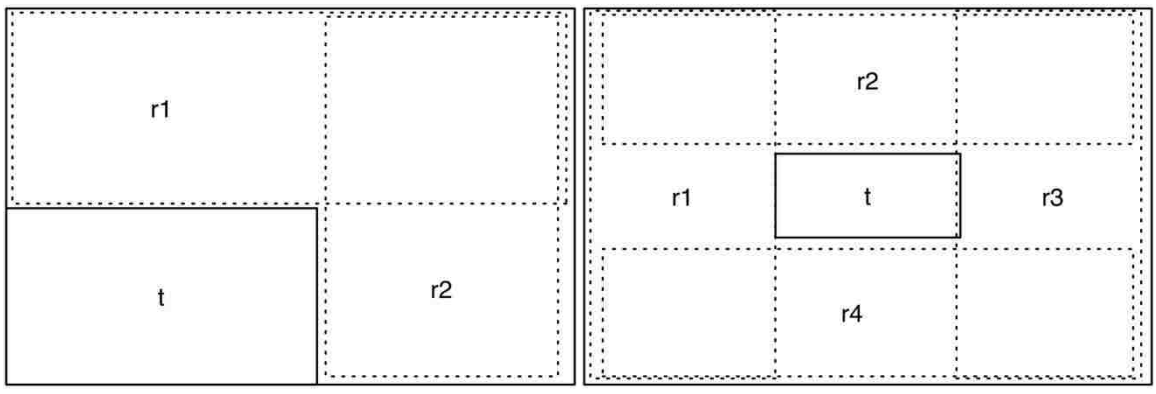
14: END

This phase (Steps 9 - 14) updates the relations between *t* and the existing MERs that touch *t*. (A later phase, Steps 54-81, updates MERs that overlap *t*'s location.) As the number of MERs is *d*, so in the worst case it will check a maximum of *d* MERs. This phase (Steps 9 -14) in the worst case will take $O(d)$.

When the placer puts *t* in *r*, this destroys some MERs and may create others. The newly created rectangles are in two groups.

- The first group consists of new rectangles created from the remaining space (unused by *t*) in *r* in the case where *t* is smaller than *r*.
- The second group is because of the fact that *t* may overlap with MERs other than *r*. Task *t* (or part of *t*) may occupy some of the space in MER *g* that overlapped *r*. The result is that the algorithm deletes *g* and creates new MERs from the free space in *g* not occupied by *t*.

The next phase has different stages. The first stage creates records for the newly created MERs in *r* (Steps 15 – 23). A general placement of *t* in MER (Figure 5-3(a)) can generate up to four new MERs, but as we are using bottom-left placement, at most two new MERs can arise (Figure 5-3(b)). Step 15 defines the size of the free space in *r* unused by *t*. We can define this space by comparing the width and the height of *t* with the width and height of *r*. Create two MERs *r*1 and *r*2 in this space in *r*. Creating them includes inserting nodes to represent them in the graph. This stage of the current phase runs in constant time. This phase defines a new array called *New-MERs* that will store all newly created MERs in the next few phases. We will need this array in the last phase of the algorithm.



(a) (b)

Figure 5-3. (a) General case of newly created MERs (b) newly created MERs when bottom left corner placement is used

% creating records for newly created MERs in *r* %

15: define the coordinates of the remaining space not used by t

16: Create *r*1 and *r*2

17: *Adj_touch_task* (*r*1) ← Ø

18: *Adj_touch_MER* (*r*1) ← Ø

19: *Adj_overlap_MER* (*r*1) ← Ø

20: *Adj_touch_task* (*r*2) ← Ø

21: *Adj_touch_MER* (*r*2) ← Ø

22: Adj_overlap_*MER* (*r*2) ← Ø

23: *New-MERs* = {*r*1, *r*2}

The next stage in the current phase is to update the adjacency lists for *r*1 and *r*2. As explained before each MER has three types of relations. All of them need to be updated for the newly created MERs *r*1 and *r*2. In this stage, Steps 24 - 33 update the relation of newly created rectangles with currently executing tasks. Steps 34 – 43 update the touching relation between newly created MERs and existing MERs. Steps 44 – 53 update the overlapping relation between newly created MERs and existing MERs.

% updating relation between newly created MERs and existing executing tasks %

24: For each task *u* ∈ *Adj_touch_task* (*r*)

25: If *u* touches *r*1

26: *Adj_touch_task* (*r*1) ← *Adj_touch_task* (*r*1) ∪ *u*

27: *Adj_touch_task* (*u*) ← *Adj_touch_task* (*u*) ∪ *r*1

28: END

29: If *u* touches *r*2

30: ***Adj_touch_task*** **(*r*2)** ← ***Adj_touch_task*** **(*r*2)** ∪ ***u***

31: ***Adj_touch_task*** **(*u*)** ← ***Adj_touch_task*** **(*u*)** ∪ ***r*2**

32: **END**

33: **END**

Steps 24 – 33 update the relations among *r*1, *r*2, and executing tasks. This part of the current phase takes $O(n)$ time as we have n currently executing tasks.

% updating touching relation between newly created MERs and existing MERs %

34: **For each MER** ***g*** ∈ ***Adj_touch_MER*** **(*r*)**

35: **If** ***g*** **touches** ***r*1**

36: ***Adj_touch_MER*** **(*r*1)** ← ***Adj_touch_MER*** **(*r*1)** ∪ ***g***

37: ***Adj_touch_MER*** **(*g*)** ← ***Adj_touch_MER*** **(*g*)** ∪ ***r*1**

38: **END**

39: **If** ***g*** **touches** ***r*2**

40: ***Adj_touch_MER*** **(*r*2)** ← ***Adj_touch_MER*** **(*r*2)** ∪ ***g***

41: ***Adj_touch_MER*** **(*g*)** ← ***Adj_touch_MER*** **(*g*)** ∪ ***r*2**

42: **END**

43: **END**

Steps 34 - 43 check if *r*1 and/or *r*2 touch any of the current MERs and update the adjacency lists accordingly. This stage of the current phase takes $O(d)$ time as we have d MERs to check in the worst case. The last stage of the current phase is to check if *r*1 and/or *r*2 overlap any of the current MERs and update the adjacency lists accordingly.

% updating the overlapping relation between newly created MERs and existing MERs %

44: For each MER *g* ∈ *Adj_overlap_MER* (*r*)

45: If *g* overlaps *r*1

46: *Adj_overlap_MER* (*r*1) ← *Adj_overlap_MER* (*r*1) ∪ *g*

47: *Adj_overlap_MER* (*g*) ← *Adj_overlap_MER* (*g*) ∪ *r*1

48: END

49: If *g* overlaps *r*2

50: *Adj_overlap_MER* (*r*2) ← *Adj_overlap_MER* (*r*2) ∪ *g*

51: *Adj_overlap_MER* (*g*) ← *Adj_overlap_MER* (*g*) ∪ *r*2

52: END

53: END

Steps 44 – 53 check if *r*1 and/or *r*2 overlap any of the current MERs and update the adjacency lists accordingly. This part of the current phase takes $O(d)$ time as we have d MERs to check in the worst case. The total time complexity for the current phase (Steps 24 – 53) in the worst case is $O(n + d)$.

As *t* may overlap with some of the MERs overlapping with *r*, this will lead to creating new MERs in any rectangles that cover some of the space used by *t*. Task *t* may only overlap with MERs overlapping with *r*. The next phase (Steps 54 – 81) tests the coordinates of *t* against the coordinates of each rectangle *g* that overlaps with *r* and defines the part of *g* that overlaps with *t*. Once this is defined, the update algorithm creates new rectangles in *g*. The number of new rectangles to be created in *g* is constant. The algorithm handles the new rectangles the same way as *r*1 and *r*2 in the previous phase. As we may have $O(d)$ MERs overlap with *r* in the worst case, so for each new rectangle we need to check the new rectangle versus $O(d)$ MERs. This phase will take $O(d^2)$ time.

% creating new rectangles in MERs overlapping *r* and affected by *t* %

54: For each MER *g* ∈ *Adj_overlap_MER* (*r*)

55: define the part of *t* that falls into *g*

% this step may result in a maximum of 4 new rectangles based on Figure 5-3(a). Array *new_rectangles(g)* will hold new rectangles %

56: *new_rectangles* (*g*) = {*r*1, *r*2, *r*3, *r*4}

57: *New-MERs = New-MERs* ∪ *new_rectangles* (*g*)

58: add records for rectangles in new_rectangles

% adding records is similar to Steps 15 – 22 %

59: for each MER *f* ∈ *new_rectangles* (*g*)

60: for each task *u* ∈ *Adj_touch_task* (*g*)

61: If *u* touches *f*

62: *Adj_touch_task* (*f*) ← *Adj_touch_task* (*f*) ∪ *u*

63: *Adj_touch_MER* (*u*) ← *Adj_touch_task* (*u*) ∪ *f*

64: END

65: END

66: for each MER *m* ∈ *Adj_touch_MER* (*g*)

67: if *f* touches *m*

68: *Adj_touch_MER* (*m*) ←*Adj_touch_MER* (*m*) ∪ *f*

69: *Adj_touch_MER* (*f*) ← *Adj_touch_MER* (*f*) ∪ *m*

70: END

71: END

72: for each MER *m* ∈ *Adj_overlap_MER* (*g*)

73: **if *f* overlaps *m***

74: ***Adj_overlap_MER* (*m*) ← *Adj_overlap_MER* (*m*) ∪ *f***

75: ***Adj_overlap_MER* (*f*) ← *Adj_overlap_MER* (*f*) ∪ *m***

76: **END**

77: **END**

78: **END**

%% at this point we need to delete MER *g*. The algorithm will delete g from the list of every neighbor in the lists of *g.* Using the pointers in the lists of *g,* we can find *g* in constant time in any of the neighbors, lists. Details of *delete_MER* will be provided after this code.%%

79: ***delete_MER* (*g)***

80: END

81: *delete_MER (r)*

The *delete_MER* (*z*) routine deletes MER *z* from the set of MERs, deletes *z*'s adjacency lists, and deletes *z* from the adjacency list for any task or MER that had a relation with *z*. The time complexity for this subroutine is $O(n + d)$.

Delete_MER(z)

1: for each task *u* ∈ *Adj_touch_task* (*z*)

2: ***Adj_touch_MER*(*u*) = *Adj_touch_MER*(*u*) – *z***

3: END

4: for each MER *m* ∈ *Adj_touch_MER* (*z*)

5: ***Adj_touch_MER*(*m*) = *Adj_touch_MER*(*m*) – *z***

6: END

7: for each MER *m* ∈ *Adj_overlap_MER* (*z*)

8: *Adj_overlap_MER*(*m*) = *Adj_overlap_MER*(*m*) – *z*

9: END

10: delete *z* record from MER list and delete all of *z*'s adjacency lists

The last phase is to delete redundant rectangles. Redundant rectangles can be one of the new created MERs or any of the affected rectangles in the previous phases. The algorithm stored all newly created MERs in previous phases in *New-MERs*. A rectangle is redundant if it is identical to another rectangle or contained in a bigger rectangle.

82: for each MER *g* ∈ NEW_MERs ∪ *Adj_overlap_MER*(*r*)

83: for each MER *f* ∈ New_MERs ∪ *Adj_overlap_MER*(*r*) – *g*

84: if g and *f* are identical or if *g* is contained in *f*

85: delete_MER (*g*)

86: Exit loop

87: end

88: END

The time complexity for the last phase is $O(d^2)$. As explained in previous phases we can conclude the total worst-case time complexity for the whole algorithm is $O(n + d^2)$. As explained before in this chapter, in the worst case d is $O(n^2)$, so we can conclude that the time complexity for updating MERs after inserting a task can take up to $O(n^4)$ time. Of course, this is the worst-case analysis, but as we only need to check the affected MERs and tasks, we expect the algorithm in practical situations to not have to check all d MERs and n tasks every step. So, the practical running should be less than $O(n^4)$.

5.2.4. Updating MERs after Delete Action

Algorithm 5-2 shows the details of the update process after deleting a task *t*. The first phase is to create a new rectangle, *r*1, in the area used by *t.* At this step *r*1 does not have to be a MER but by the end of the update process all rectangles in the data structure will be MERs. The inputs to Algorithm 5-2 are the parameters for *t*, the set of MERs, and the relation graph. The outputs of Algorithm 5-2 are the updated set of MERs and an updated relation graph.

Algorithm 5-2 updating MERs after deleting task *t*.

Inputs: task *t* parameters, MERs, relation graph

Output: updated MERs, updated relation graph

%% Add a new node in the graph for *r*1 and create an empty adjacency list for it

%% *r*1 has the same coordinates and size as task *t*

1: *Adj_touch_task* (*r*1) ← Ø

2: *Adj_touch_MER* (*r*1) ← Ø

3: *Adj_overlap_MER* (*r*1) ← Ø

As described before, adding records and creating the adjacency lists take constant time, so this phase will take $O(1)$ time. Updating the free space after deleting a task follows the MER Task Deletion Theorem, so the only rectangles affected by deleting task *t* are MERs that share part of a border with *t.* For the simplicity of explaining the procedure we will call them *affected MERs*. As new free space is available by deleting *t*, this means that some of the affected MERs may no longer be maximal as they can extend over this space. The algorithm will check if each affected MER can be extended.

To explain how a rectangle can extend over the space occupied by another rectangle, consider two rectangles *f* and *g* that share part of a border. Different cases exist in which *f* can

extend over part or all of the space of *g*. Without loss of generality, assume that *g* is on the left side of *f*. Figure 5-4 depicts several cases; other cases exist (for example, *g* may be below *f*) and they are analogous to these. We have three main categories: *f* and *g* share no borders at all; *f* and *g* share one border; and *f* and *g* share two borders. Sharing one border means their top or bottom borders has the same *y* dimension value. Sharing two borders means the top borders have the same *y* dimension value and so do their bottom borders.

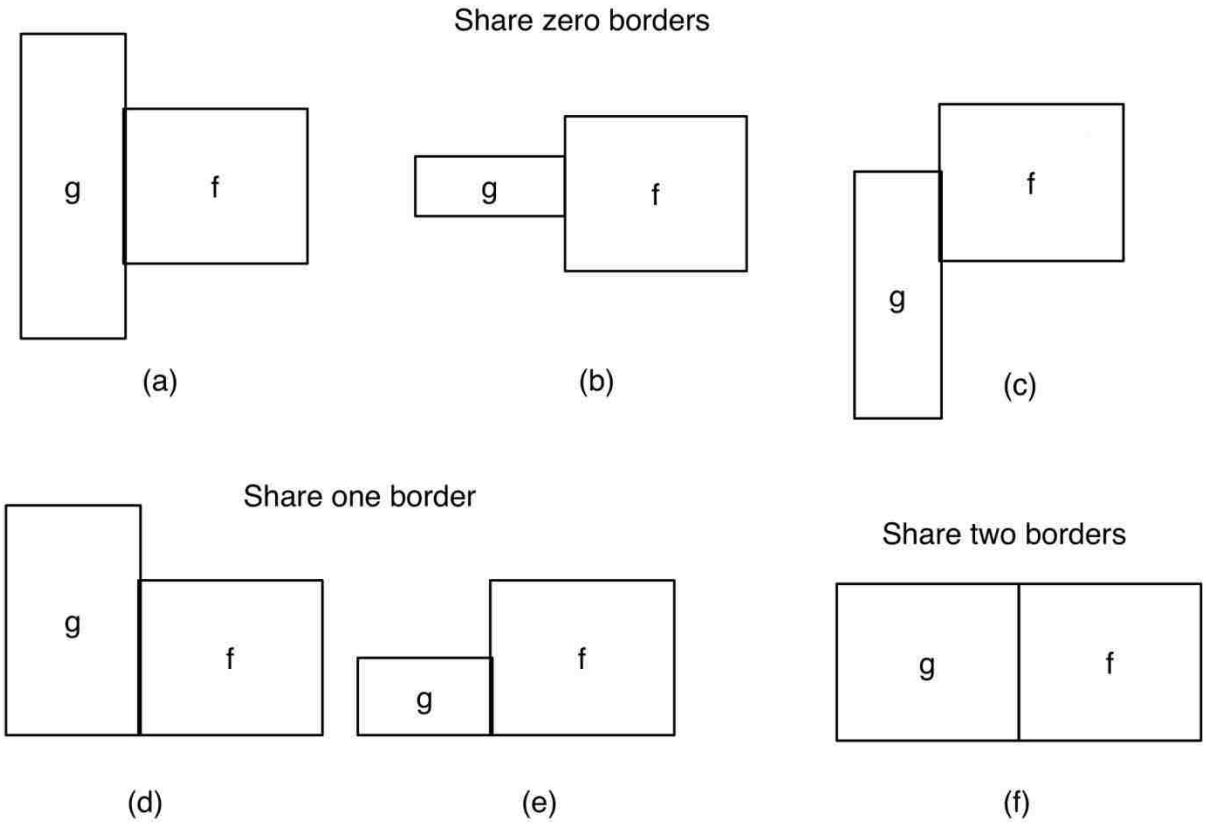


Figure 5-4 Different cases for possible extension of rectangles.

In the first category, *f* and *g* are sharing no borders. In Figure 5-4(a), *g*'s top is above *f*'s top and *g*'s bottom is below *f*'s bottom. In this case, *f* can extend into space occupied by *g*. The width of *f* will change but the height will stay the same. Rectangle *g* will stay the same after the extension. In Figure 5-49(b), *g*'s top is below *f's* top and *g*'s bottom is above *f*'s bottom, so *f* cannot extend into *g*. (Note, however, that *g* can extend over the space occupied by *f*. The

algorithm will perform this extension when it goes over the loop in which it handles *g* instead of *f*). Figure 5-4(c) shows a case in which *f* and *g* are sharing no borders and neither can extend in the space occupied by the other. In this case, the algorithm will create a new rectangle that will cover part of the space covered by *f* and *g*.

In the second category, *f* and *g* are sharing one border. By analogy with Figure 5-4(a), in Figure 5-4(d), *f* will extend into the space occupied by *g*. Again, the width of *f* will change and *g* will stay the same. In Figure 5-4(e), *f* cannot extend into *g*.

The last category is shown in Figure 5-4(f) in which *f* and *g* are sharing two borders. In this case, the algorithm will extend *f* into *g* and delete *g*.

The following code deals with the above cases in which *f* extends or a new rectangle results. (For brevity, we omit the analogous cases.) So, in this phase Algorithm 5-2 checks each pair of rectangles in the affected MERs list for a possible expansion. In the following, array *affected_MERs* contains the set of affected MERs by deleting *t*. When deleting a rectangle or creating a new rectangle, the *affected_MERs* array dynamically changes, and the algorithm will account for this change. The algorithm will check each affected MER against all others because if one of them expanded into the space occupied by *r*1, then some other MERs may be able to expand more.

```
4: affected_MERs = { r1 ∪ Adj_touch_MER(t)}
5: for each MER f ∈ affected_MERs
6:         for each MER g ∈ affected_MERs – f
7:                 if g and f share no borders
8:                         If top (g) > top (f) && bottom(g) < bottom(f) %Figure5-4(a)
9:                                 width (f) = width (g) + width (f)
```

10: END

11: if top (*g*) < top (*f*) && bottom (*g*) < bottom (*f*)

%% Figure 5-4(c) %%

12: Create *R2* in the space in *f* and *g* in which a new rectangle can be created

13: *Adj_touch_task* (*R2*) ← Ø

14: *Adj_touch_MER* (*R2*) ← Ø

15: *Adj_overlap_MER* (*R2*) ← Ø

16: affected_MERs = affected_MERs ∪ {*R2*}

17: END

18: END

19: if *g* and *f* share the bottom border

20: If top (*g*) > top (*f*) %% Figure 5-4(d)

21: width (*f*) = width (*g*) + width (*f*)

22: END

23: END

24: If *g* and *f* share top and bottom borders %% Figure 5-4(f)

25: width (*f*) = width (*f*) + width (*g*)

26: delete (*g*)

27: *affected_MERs = affected_MERs - g*

28: END

29: END

30: END

As explained before, assume we have d MERs. In the worst case, we may have d MERs in the *affected_MERs* array. So this phase (steps 4 – 48) in the worst case will have a time complexity of $O(d^2)$.

In the next phase, the algorithm updates the relations of all tasks touching t and the affected MERs. This phase can be done as in Algorithm 5-1. In the worst case, we have n tasks to compare against d MERs and update the relations among tasks and MERs. This stage of the current phase takes $O(nd)$ time. Also we need to check and update the relations among all affected MERs as some of them changed in size. This stage of the current phase may take up to $O(d^2)$ time. So, the total time complexity for the current phase is $O(nd + d^2)$. As explained before, in the worst case, d can be $O(n^2)$. So, the total worst-case time complexity for Algorithm 5-2 is $O(n^4)$.

The proposed update algorithm has the same time complexity as other update algorithms presented by other researchers in the worst-case analysis. The proposed algorithm, however, builds on theorems that limit the number of MERs and tasks to be checked during the process to some of the MERs and not all of them. Therefore, in practical cases, we expect less than d affected MERs and so a faster update process than previous methods.

5.2.5. Creating MERs from Scratch

Surprisingly, however, creating the set of MERs from scratch is more efficient than updating an existing set of MERs — one can create the set of MERs in $O(n \log n + d)$ time, which is $O(n^2)$ in the worst case [NBR90]. Consequently, the time complexity of task insertion and deletion by a placer that uses MERs as a free space data structure is $O(n^2)$.

CHAPTER 6: HORIZONTAL AND VERTICAL STRIPS

In this chapter we will present the use of maximal horizontal and vertical strips as a data structure to represent free space on an FPGA. The idea behind the maximal horizontal and vertical strips (MHVS set) is to represent the 2D reconfigurable area as rectangular strips that are relatively few in number, but that give good coverage of the available free space.

6.1. Introduction

The strips are of two types: *maximal horizontal strips* (MHS) and *maximal vertical strips* (MVS). The top of each horizontal strip aligns with a top or bottom of a task and the bottom of the strip aligns with the next task top or bottom, and each strip has maximal width. One can construct vertical strips analogously (see Figure 6-1). In this chapter we will use different notations. "MHS" stands for a maximal horizontal strip. "MHS set" stands for the set of all MHSs. The same applies for "MVS" and "MVS set". "MHVS set" stands for the union of MHS set and MVS set.

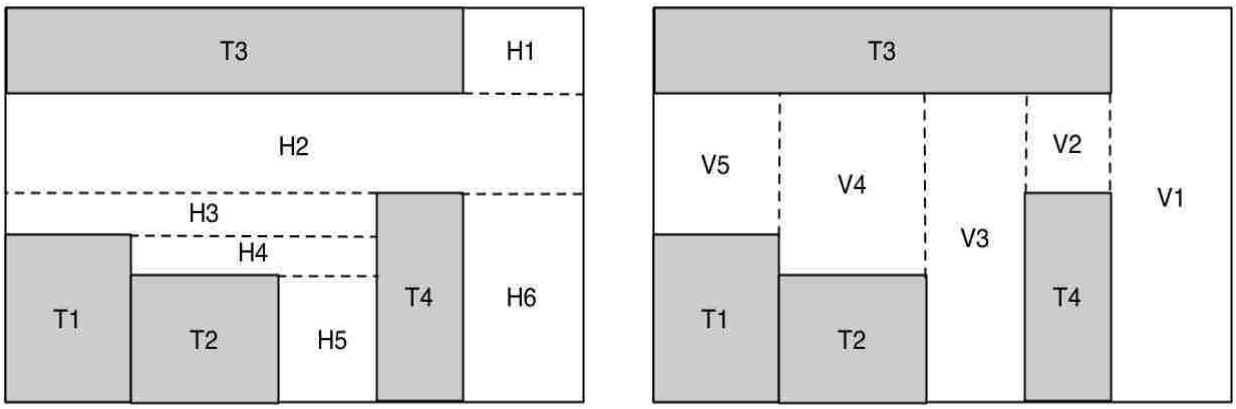


Figure 6-1 Example for (a) maximal horizontal strips and (b) maximal vertical strips, where shaded rectangles are tasks.

In this chapter we present an algorithm for creating MHS set and MVS set in $O(n \log n)$ time given a set of n tasks. A second algorithm will start with an initial MHS set and MVS set and update them after each task insertion or deletion. The update algorithm takes only $O(n)$ time.

Section 6.2 explains in detail the steps required for creating MHVS set given *n* tasks. Section 6.3 explains in detail the steps needed after each task deletion and task insertion. Section 6.4 gives a detailed comparison between the method of MHVS and other data structures like bit matrix and MERs.

6.2. Creating MHVS Set from Scratch

In this section we will show how MHVS set can be created from scratch given a set of *n* executing tasks. The algorithm for creating MHS set and MVS set depends on an idea similar to the algorithm for creating MERs by Nandy *et al.* [NBR90]. The algorithm for creating MHS set scans the chip from its top border with a falling curtain, tracking intervals of free space. Each time the algorithm reaches a top or bottom of a task it closes intervals, creates maximal horizontal strips, and creates new intervals.

Algorithm 6-1, *create_MHS*, specifies how to generate the set of all maximal horizontal strips, given a set of active tasks. The algorithm starts with an initial interval of width equal to the chip width. As the stopping and starting points for MHSs are the tops and the bottoms of tasks, the algorithm begins by sorting all tops and bottoms of the executing tasks in descending order. There are some special cases in ordering the tops and bottoms. For example, if task *A* is touching task *B* from above (Figure 6-2), then the bottom of task *A* is sorted before the top of task *B*.

Also in Figure 6-2, tasks *C*, *D*, and *E* have their bottoms at y_1. We assume without loss of generality that the algorithm will sort them from left to right so the bottom of C will be before the bottom of *D* which will be before the bottom of *E*. Also note that tasks *F* and *G* have their tops at y_1 too, so, we assume again that the algorithm will sort the bottoms of *C*, *D*, and *E* in the above order first and then the top of *F* and the top of *G*.

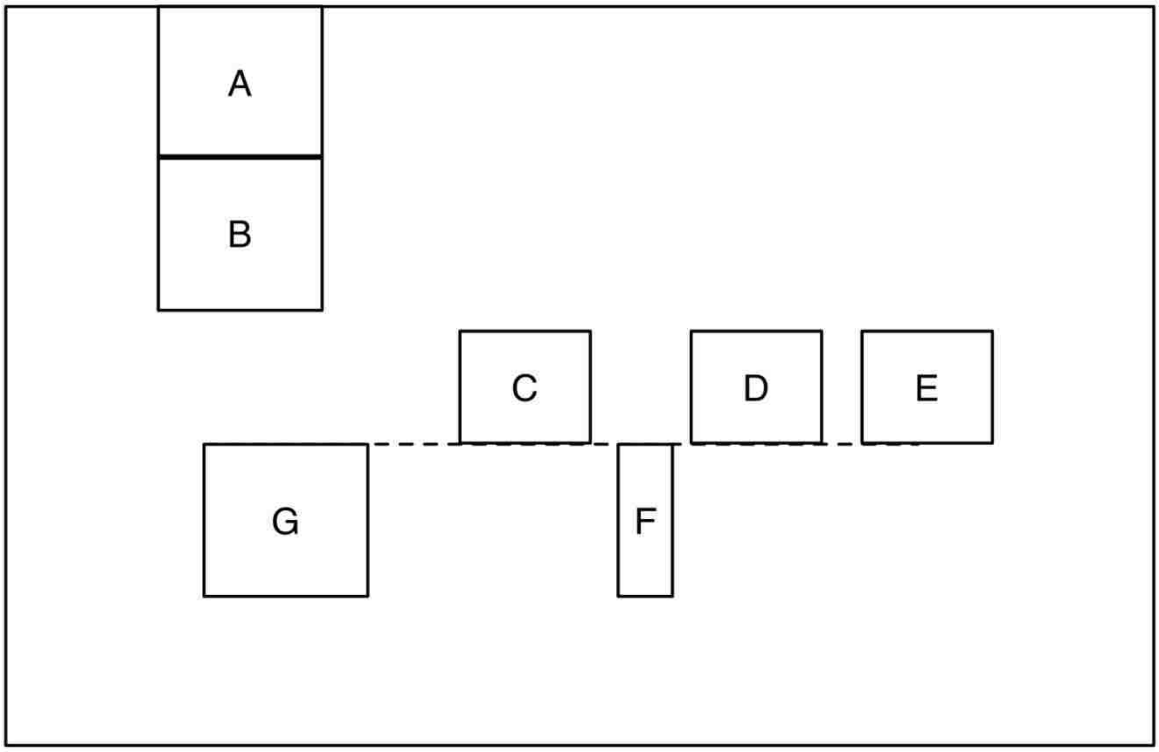


Figure 6-2 Special cases in sorting tops and bottoms

Each time the algorithm reaches the top of a task, one interval overlaps with the task. (Overlapping between a task and an interval does not mean they overlap in space, rather that their *x* ranges overlap.) The algorithm deletes this interval and, if it has non-zero height, creates an MHS with the same width. For example, in Figure 6-1(a) when the algorithm reaches the top of task T_4, it will close the interval on top of T_4 and create the strip H_2 with the same height and width as the interval closed. After creating the strip, the algorithm will delete the interval and open two new intervals starting at the top of the task, located on the right and the left of the task (if the previous interval was wider than the task and if no other tasks or borders are to the immediate left and right). Note in Figure 6-1(a) when the interval on top of T_1 closes, one interval on the right of T_1 is created.

Each time the algorithm reaches the bottom of a task, it searches for the two intervals on the left and the right of the task. (In some cases, the algorithm can get null for one or both of these

intervals when the task touches another task from the side or the border of the chip.) It creates a strip for each interval found and deletes the intervals. It creates a new interval starting just under the task bottom *y* value covering the span of the two intervals and the task. (Note: This is why only one interval overlaps the top of a task, even if one task *a* sits on top of another task *b*. Processing the bottom of task *a* creates an interval spanning the task and free space to the left and right and does so before processing the top of *b*.) In Figure 6-1(a) when the algorithm reaches the bottom of T_3, one interval is found on the right of T_3 and no interval on the left side of T_3. The algorithm will close the right interval, create H_1, and start a new interval under T_3 spanning the width of both H_1 and T_3.

When several tasks have their bottoms at the same *y* value, for example, tasks *C*, *D*, and *E* in Figure 6-2, the algorithm will start with the leftmost task *C*, close the intervals on the right and the left, and create a new interval that will be on the left of the next task *D*. When the algorithm reaches the bottom of task *D*, the interval on the left of *D* will have a height of zero as it was created at the same *y* value as the bottom of task *C* which is the same as the *y* value for the bottom of *D*. In this case, the algorithm will close the interval on the left of task *D* without creating an MHS. It also creates a new interval under *D* spanning the interval created under *C* and the width of task *D*. The algorithm continues until finishing the bottom of task *E*, and the final result will be an interval that spans the width of the three tasks and all the gaps between and besides them. Again when the algorithm reaches the top of the next task *E*, there will be only one interval overlapping the task.

Algorithm 6-1: create_MHS_set

Input: set *E* of executing tasks

Output: set *H* of maximal horizontal strips of free space

1: initialize interval set *Int* to include initial interval = [chip height, 1, chip width]

/* The format of an interval is (top y, left x, right x), including, for the purpose of MHS generation, the y value at which it was created. */

2: $T\_B \leftarrow$ sort task tops and bottoms from E

/* Let each element of *T_B* hold (*y, left_x, right_x, T_B_flag*), where *T_B_flag* indicates whether it is a top or a bottom. */

3: $H \leftarrow \varnothing$

4: for i = 1 to 2 * $|E|$

5: *current_T_B* ← *T_B*(*i*)

6: if *current_T_B*(*T_B_flag*) = top

7: *current_int* ← *int_search_top*(*Int*, *current_T_B*)

/* find the only interval that overlaps with task */

8: *newstrip* ← *makeMHS*(*current_int*, *current_T_B*(*y*) − 1)

/* create an MHS from the overlapping interval */

9: $H \leftarrow H \cup \{newstrip\}$

10: {*newint*1, *newint*2} ← *create_two_intervals*(*current_int*, *current_T_B*)

/* generate intervals left and right of the task top */

11: $Int \leftarrow Int \cup \{newint1, newint2\} - \{current\_int\}$

12: else

13: {*current_int*1, *current_int*2} ← *int_search_bottom*(*Int*, *current_T_B*)

/* find two intervals that touch task left and right */

14: *newstrip*1 ← *makeMHS*(*current_int*1, *current_T_B*(*y*))

15: *newstrip*2 ← *makeMHS*(*current_int*2, *current_T_B*(*y*))

16: $H \leftarrow H \cup \{newstrip1, newstrip2\}$

17: $newint \leftarrow create\_one\_interval(current\_int1, current\_T\_B, current\_int2)$

/* generate an interval from the left interval, task bottom, and right interval */

18: $Int \leftarrow Int \cup \{newint\} - \{current\_int1, current\_int2\}$

19: end

20: if the bottom of the last task is not at the bottom of the chip

21: $H \leftarrow H \cup$ {strip with width of chip and height from bottom of chip to bottom of last task}

The above algorithm calls several subroutines. Subroutine *int_search_top* finds the interval that overlaps with the top of a task; *int_search_bottom* finds the two intervals on the right and left side of a task when reaching the bottom of that task. When reaching a top (bottom) of a task, the *makeMHS* subroutine creates a strip from the current interval overlapping with the task (two intervals on the right and the left of a task). Subroutine *create_two_intervals* creates the two intervals when reaching a task top; *create_one_interval* joins three intervals to create one interval when reaching a task bottom.

As *create_MHS_set* generates the set of MHSs by scanning the tops and bottoms of the executing tasks, there are $O(n)$ MHSs. To perform the searches in *int_search_top* and *int_search_bottom* efficiently, we store the set of active intervals in an interval tree [CLRS01]. Searches take time $O(\log n)$ in an interval tree with up to n intervals; interval insertions and deletions in this data structure also take $O(\log n)$ time. Consequently, *create_MHS_set* runs in $O(n \log n)$ time. Thus, $O(n \log n)$ bounds the time for a placer using the MHS set (or MVS set or MHVS set) to insert or delete a task on an FPGA.

6.3. Updating MHVS set after Inserting/Deleting a Task

We next describe how to update the strips every time the scheduler places a task or deletes a task. This update process will take $O(n)$ time to update the strips but needs to maintain additional information to perform efficient updates. Inputs to the algorithm include task t to be placed or deleted, list H of MHSs, list V of MVSs, and an array of neighbor pointer lists (pointing to adjacent tasks and strips for each task and strip), where each list is doubly linked and sorted. The algorithm outputs updated H, V, and neighbor pointer lists. Let $T(p)$ denote the doubly-linked, sorted list of top neighbors of p, where p is a strip or a task. Similarly, let $B(p)$, $L(p)$, and $R(p)$ denote lists for bottom, left, and right neighbors, respectively.

We will describe the MHVS set update algorithm for task insertion in some detail (specifically, insertion in an MHS) then the update algorithm for task deletion more briefly. Let s denote the strip into which task t will be placed. We assume that the placer locates t in the bottom-left position of s, but a generalization of the update algorithm will work for any placement in a strip. The algorithm has two parts. The first part handles the strip in which the task is placed. The second part handles the strips of the other type that the task overlaps. For example, if the scheduler placed task t in an MHS z, then the first part will handle the update needed for strip z as it is the only affected MHS. The four borders of all MHSs other than z will not change because the tasks and chip boundaries that determined these borders do not change with insertion of t. Since z is an MHS, no task top or bottom can be on the right or left border of z between its top and bottom. The second part of the update algorithm will handle all the affected MVSs that task t cuts (overlaps). Also, when placing t in an MHS, MVSs that do not overlap t do not need to be updated for the same reasons as MHSs that do not overlap t.

Algorithm 6-2, *task_insert_update_MHVS*, explains the steps needed to update the MHSs and MVSs after placing a task within an MHS. Also, the algorithm assumes that MHS list *H* is sorted from top to bottom and MVS list *V* is sorted from left to right. (It will keep them sorted after the update process.) Initially, before placing any task and as the chip is empty, the algorithm starts with one MHS as big as the chip size and one MVS as big as the chip size.

Algorithm 6-2 task_insert_update_MHVS

Input: MHS list *H*, MVS list *V*, task *t* parameters, MHS *z* in which *t* is placed, array of neighbor pointers for strips and tasks

Output: updated *H*, *V*, pointers array

Part 1 of the algorithm, after bottom-left placement of *t* in *z*, will create zero, one, or two MHSs (based on the height and width of *t* compared to *z*) in the remaining area of *z* and will update neighbor pointers accordingly. The general case is if the width and the height of *t* are less than the width and the height, respectively, of *z*. The algorithm will create two strips as shown in Figure 6-3. If *t* has the same size as *z*, then no new strips need to be created. If *t*'s width equals *z*'s width, or if the *t*'s height equals *z*'s height, but not both, then the algorithm will create only one new strip. As the algorithm will create a maximum of two strips, then it can scan the list and place them in sorted order without the need to resort the whole list. The algorithm determines neighbors of *t*, z_1, and z_2 from neighbors of *z*.

In the following code we will explain the general case in which two new strips are created. The pseudo-code below uses two subroutines *copy_pointer* and *check_update*. The subroutine *copy_pointer* (*C*, *D*) creates list *C* as a copy of list *D*. Subroutine *check_update* (C_1, D_1, C_2, D_2) creates lists C_1 and D_1 and copies to C_1 (D_1) the neighbors in C_2 (D_2) that touch the strip or task corresponding to C_1 (D_1). For example, *check_update* (*L*(*t*), *B*(*t*), *L*(*z*), *B*(*z*)) copies to *L*(*t*) the

left neighbors of *z* that are also left neighbors of *t* and copies to *B*(*t*) the bottom neighbors of *z* that are also bottom neighbors of *t*.

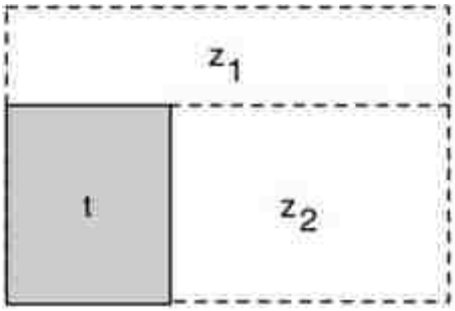


Figure 6-3 New strips z_1 and z_2 after placing task t in MHS z

1: create two strips z_1 , z_2 as shown in Figure 6-3 and insert them in *H* in sorted order

/* Create lists of neighbor pointers for *t* */

2: *copy_pointer* (*R*(*t*), ‹z_2›)

3: *copy_pointer* (*T*(*t*), ‹z_1›)

4: *check_update* (*L*(*t*), *B*(*t*), *L*(*z*), *B*(*z*))

/* Create lists of neighbor pointers for z_1 */

5: *copy_pointer* (*T*(z_1), *T*(*z*))

6: *copy_pointer* (*B*(z_1), ‹*t*, z_2›)

7: *check_update* (*R*(z_1), *L*(z_1), *R*(*z*), *L*(*z*))

/* Create lists of neighbor pointers for z_2 */

8: *copy_pointer* (*T*(z_2), ‹z_1›)

9: *copy_pointer* (*L*(z_2), ‹*t*›)

10: *check_update* (*R*(z_2), *B*(z_2), *R*(*z*), *B*(*z*))

Part 1 of the algorithm runs in $O(n)$ time based on the fact that, for a given set of n executing tasks, the array of neighbor pointers can have a total of up to $O(n)$ pointers for all tasks and strips.

The second part of the algorithm will test task t against each MVS and update each MVS that t overlaps. As t is placed in an MHS z and as a bottom-left corner placement is used, eight cases describe the possible overlap of t with an MVS. Task t may totally overlap MVS v. Task t may span the height of v but end without reaching the right border of v. Task t may cut across v. This cut might happen at the top or the bottom or the middle of v. Task t may cut v from the left border and end without reaching the right border of v; this also can happen at the top, the bottom, or in the middle. Figure 6-4 shows some of these cases. For example, task t cuts across v_1 from the bottom. Also, t totally overlaps v_2 and cuts across the middle of v_4. Task t cuts the left border of v_5 in the middle and ends inside v_5.

The newly created strips will replace the original strips to keep the strips list in order, that is, v_4' replaces v_4. The only case that needs special handling is the case when t cuts the left border of a strip (as where t cuts v_5 in Figure 6-4). The last part of the task will split the strip into at most three strips. Also, these three new strips can be placed in order by scanning across the n strips. Neighbor pointers derive from neighbor pointers of the original overlapped MVSs.

(a) (b)

Figure 6-4 Placing a task in an MHS strip and updating MVSs accordingly: (a) before insertion and (b) after insertion

The pseudo-code below discusses the MVS update process for the case shown in Figure 6-5, in which *t* cuts the left border of *v* in the middle, but ends before the right border of *v*. As Figure 6-5 shows, three new MVSs created. The framework for this pseudo-code is that the placer tests each MVS for overlap with task *t* and determines the appropriate case.

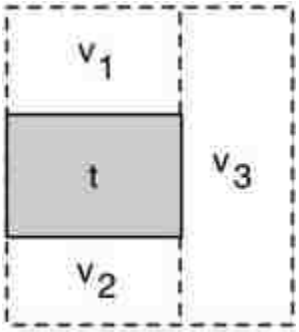


Figure 6-5 Updated MVSs for the case of a task overlapping an MVS described in the pseudo-code

1: define the part of the task that falls into MVS *v*

2: create three new MVSs v_1, v_2, v_3 as shown in Figure 6-5

/* Create lists of neighbor pointers for v_1 */

3: *copy_pointer* ($R(v_1)$, ‹v_3›)

4: *copy_pointer* ($B(v_1)$, ‹t›)

5: *check_update* ($T(v_1)$, $L(v_1)$, $T(v)$, $L(v)$)

/* Create lists of neighbor pointers for v_2 */

6: *copy_pointer* ($T(v_2)$, ‹t›)

7: *copy_pointer* ($R(v_2)$, ‹v_3›)

8: *check_update* ($L(v_2)$, $B(v_2)$, $L(v)$, $B(v)$)

/* Create lists of neighbor pointers for v_3 */

9: *copy_pointer* ($L(v_3)$, ‹v_1, t, v_2›)

10: *copy_pointer* ($R(v_3)$, $R(v)$)

11: *check_update* ($T(v_3)$, $B(v_3)$, $T(v)$, $B(v)$)

/* Create lists of neighbor pointers for t. Lists of MVS neighbors for t are distinct from lists of MHS neighbors for t, though we use the same notation here for simplicity. */

12: if t extends to the left of v then

13: append v_1 to $T(t)$

14: append v_2 to $B(t)$

15: else

16: *copy_pointer* ($T(t)$, ‹v_1›)

17: *copy_pointer* ($B(t)$, ‹v_2›)

18: *check_update* ($L(t)$, $L(v)$)

/* same as earlier *check_update* but for pair of lists*/

19: *copy_pointer* ($R(t)$, ‹v_3›)

The algorithm needs to perform an additional check for new MVSs. If the left or right neighbor of a new MVS is also a new MVS, then the algorithm checks if it should merge them. (For example, in Figure 6-4, v_3' results from merging new MVSs that were originally part of v_3, v_4, and v_5.)

The time complexity of Part 2 of the update algorithm, updating the MVSs, is $O(n)$. It checks each of $O(n)$ MVSs for overlap with task *t*. Because the total number of neighbor pointers is $O(n)$, the total time to create new neighbor lists across all overlapped MVSs is $O(n)$, where each case follows along the lines of the pseudo-code above.

We now sketch the process for updating the MHS set after a task deletion. The process for updating MVS set is, of course, similar. The concept is to execute a limited-width falling curtain (as in *create_MHS_set*) to create strips after removing a task. Let *t* denote the task to be deleted. The algorithm creates one interval that covers *t*. This interval starts at or above the top of *t*. The algorithm sweeps down across the space of *t*, creating MHSs and neighbor pointers, and then stops at or below the bottom of *t*. A key point for fast execution time is that it has only one interval at a time.

Creating the initial interval depends on the top neighbors of *t*. If *t* has one task as top neighbor or multiple tasks and strips as top neighbors (Figure 6-6(a)), then create the interval to start at the top of *t*. (If *t* has at least one task *g* as a top neighbor, then no strip above *t* can expand into the space that was occupied by *t* because of the bottom border of *g*.) If the top left neighbor of *t* is a task, then the left side of the interval is the left side of *t*. Otherwise, if the top left neighbor of *t* is a strip *s*, then the left side of the interval is the left side of *s*, and the algorithm also eliminates *s* because *s* can expand into the space occupied by *t*. Set the right side of the interval analogously.

Figure 6-6 The two different cases for the delete algorithm.

On the other hand, as in Figure 6-6(b), if *t* has one strip *s* as top neighbor, and the bottom boundary of *s* derives only from the top of task *t*, then *s* may grow down into the space of *t*, so set the top, left, and right sides of the initial interval to the corresponding sides of *s* (and eliminate *s*). If the bottom of strip *s* derives from *t* and some other task other than *t*, then the initial interval will start at the top of *t* (as in the Figure 6-6(a) case).

The update algorithm proceeds as in *create_MHS_set* to sweep downward with this interval, creating MHSs and neighbor pointers for these MHSs as determined by tops and bottoms of tasks to the left and right of *t* (finding these using neighbor pointers). Finishing the update algorithm follows cases at the bottom of *t* that follow the pattern of creating the initial interval at the top of *t*.

The time complexity of this update of MHS set and MVS set after a task deletion is $O(n)$. Because neighbor pointers are already sorted, this algorithm does not need to sort the tops and bottoms of tasks (as was done in *create_MHS_set*). Also, the algorithm sweeps down only one interval, not *n* intervals as was the case with *create_MHS_set*, so it can deal with each top and bottom that it encounters to the left and right of *t* in constant time, instead of $O(\log n)$ time as in

create_MHS_set. In the worst case, the update algorithm will deal with n tops and bottoms to the left and right of t, so the update algorithm will have a time complexity of $O(n)$.

None of MHS set, MVS set, or MHVS set as free space data structures is recognition complete. This will lead sometimes to rejecting a task even though sufficient free space exists for it on the chip but this space was not recognized by the structure. Simulations in the next section show that, while the rejection ratio for MHS set alone or MVS set alone is high, the rejection ratio of placers using MHVS set, the union of MHS set and MVS set, is nearly as good as bit arrays and MERs but with much lower time complexity.

6.4. Simulation Results

In this section we present the results we obtained from simulations, comparing results of first-fit scheduling using different data structures for free space. We performed simulations with a set of parameters randomly generated and uniformly distributed over certain ranges. The simulations used the same parameters used in Section 4.3.1. We also examine the effects of varying the task area range and varying the laxity range.

This section presents the simulation results for the scheduler with a first-fit placer using each of the following free space representations: MER, MHVS set, and MHS set (only horizontal strips without vertical strips). (Simulations included the bit matrix representation, but figures omit these results as they are almost identical to MER results.)

Figure 6-7 reports rejection ratios for the free space representations. The MHVS set rejection ratio is nearly as good as that of MER across the range of chip loads, at much less cost and despite the fact that MHVS set can hold up to $O(n)$ free rectangles while MER can hold up to $O(n^2)$.

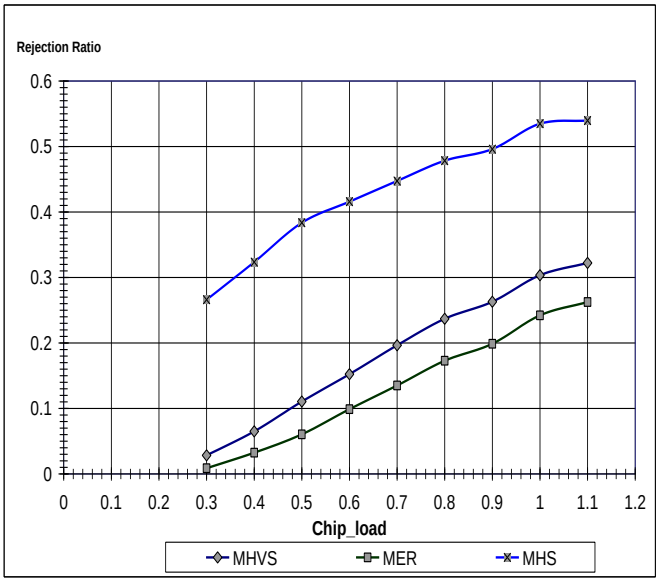


Figure 6-7 Rejection ratio for different free space structures.

Note also the significantly better rejection ratio that MHVS set (both MHS set and MVS set) achieves over MHS set alone, even though MHVS set has roughly only twice the number of free rectangles that MHS set does. This supports our intuition that MHS set tends to have wide free rectangles that disadvantage tall tasks and vice versa for MVS set, but that their combination can accommodate more tasks than either alone.

Figure 6-8 shows how different data structures utilize the chip free space. In general, for this work and prior work, most data structures are unable to utilize more than 70% of the free space. The main reason behind this is the fragmentation that happens because of inserting and deleting tasks. Simulations in Figure 6-8 were computed by averaging the chip utilization over running time and taking the average. As the chip is usually less busy in the beginning and the end of the running time, we ignored the first 200 time units and the last 200 time units, so the results indicate steady state conditions. As for rejection ratio, chip utilization for MHVS set is competitive with that of MER and much better than that of MHS set.

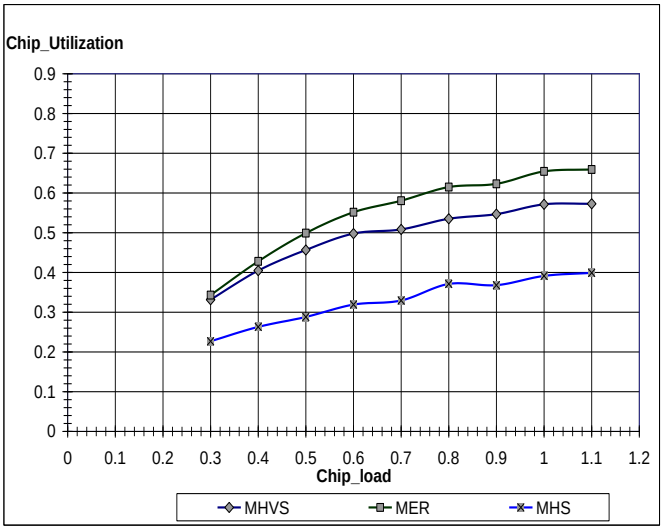


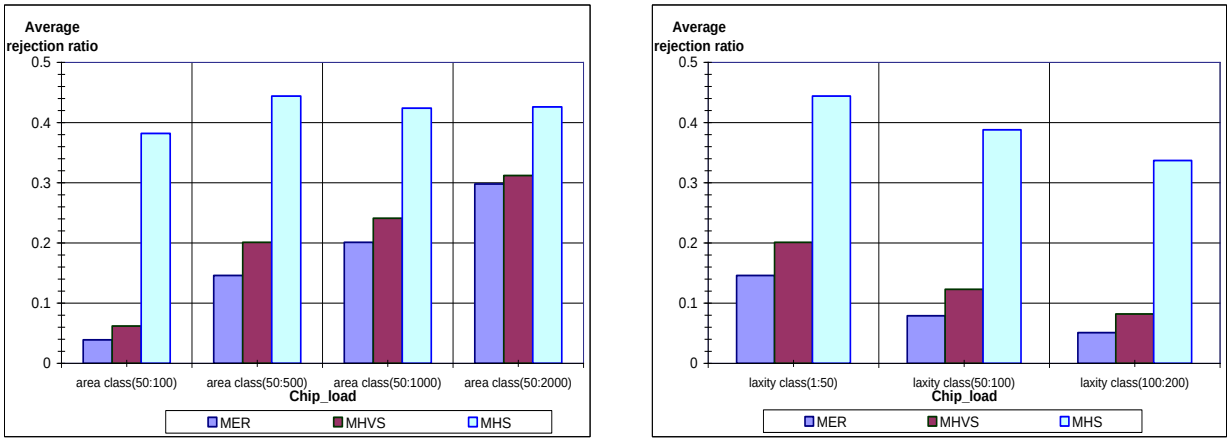
Figure 6-8 Chip utilization for different free space structures.

Table 6-1 shows the average rejection ratio across all simulations for different orderings of the free space data structures. MHVS set includes, first, MHS set then, second, MVS set generated by *create_MHS_set* and *create_MVS_set*. Entry *MHVS-sortedx* is MHVS set sorted based on the *x* value of the left edge of each strip. The idea behind this order is that the placer will start searching from the left side of the chip, so tasks may be more tightly packed. Entry *MHVS-smallsize* is MHVS set sorted in increasing order based on strip area. With this order, the placer searches the strips in a best-fit order. Entries *MER*, *MER-sortedx*, *MER-smallsize* are analogous. The bit matrix results are for a bottom-left placement of tasks. MHVS set and MER results in Figure 6-7 correspond to the "MHVS set" and "MER" in Table 6-1. From Table 6-1, we observe that sorting the free rectangles has minimal impact on performance.

Figure 6-9(a) displays the average rejection ratio for different data structures with different task area ranges. We have simulated different data structures dealing with four different task area classes [50:*A*], where *A* is 100, 500, 1000, 2000. Task area range [50:500] corresponds to Figure 6-7. Note that the rejection ratio of MHVS is even closer to that of MER in the other area ranges.

Table 6-1 Average rejection ratio of different orderings of free space.

MHVS	MHVS sortedx	MHVS smallest size	MHS Only
0.201	0.192	0.190	0.444
MER	**MER sortedx**	**MER smallest size**	**bit matrix**
0.146	0.142	0.140	0.142



(a) **(b)**

Figure 6-9 Average rejection ratio for different (a) task area and (b) laxity ranges.

Figure 6-9(b) shows the average rejection ratio for different laxity ranges. We tested the data structure for sets of tasks with laxity range varying between [1, 50], [50, 100], and [100, 200]. Range [1, 50] corresponds to Figure 6-7.

CHAPTER 7: HETEROGENEOUS RC DEVICES

In the previous chapters, we studied the problem of scheduling real-time tasks on reconfigurable devices. One assumption made in these chapters was that the reconfigurable device has a homogeneous structure. This means that only CLBs are in the reconfigurable area and all memory blocks, multipliers, embedded processors, and other components are outside the reconfigurable area. A task can be placed anywhere on the chip as long as a sufficient space of free CLBs exists. To be able to study the problem of scheduling on reconfigurable devices, we, like most other researchers, assumed a homogeneous FPGA space. Most reconfigurable devices offered in the market today, however, are not homogeneous but heterogeneous. This means that memory blocks, multiplexers, embedded processors, and other components may exist between CLBs. A common structure is to have several homogeneous CLB blocks separated by other components. Observe that a placer cannot necessarily place a task anywhere on the chip with sufficient free CLBs because if the task requires a memory block to perform its function, then the task has to be placed next to a memory block.

The main goal of this chapter is to examine whether the ideas behind schedulers designed for homogeneous devices will work for heterogeneous devices. To conduct this study, we will adapt our schedulers presented in previous chapters to a heterogeneous FPGA. To observe the impact of the heterogeneous structure, we will compare the performance of these schedulers on a heterogeneous FPGA against the performance on a homogeneous FPGA as a baseline. This baseline will help to reveal where performance shortcomings are due to heterogeneity and where they are due to the scheduler and basic scheduling problem. These results will guide our modifications of the schedulers to better fit the heterogeneous case. We conclude that the approach we took and other researchers have taken of studying homogeneous FPGAs is a valid

one, as the scheduling ideas discovered there do carry over to heterogeneous FPGAs. The need exists for more study of scheduling tasks on heterogeneous FPGAs.

Section 7.1 describes the differences between homogeneous and heterogeneous structures. In Section 7.2, we define the scheduling problem on a heterogeneous structure and then we list the parameters used to test the performance of schedulers designed in previous chapters when applied to a heterogeneous structure. We report experimental results in Section 7.3. Section 7.4 presents a scheduler designed to give better results on a heterogeneous structure. In Section 7.5, we present some modifications allowing the designed scheduler in Section 7.4 to further improve its performance. Section 7.6 shows more comparisons for different area sizes.

7.1. Heterogeneous FPGA Structure

The Xilinx Virtex-II [X07] is one of the current reconfigurable devices in the market. Figure 7-1 shows the arrangement of CLBs in a Virtex-II chip. The actual FPGA structure is not homogeneous as assumed before; instead, the CLB area has some *SelectRAM* blocks in the middle. Several problems face researchers when designing schedulers for a heterogeneous structure. The first problem is that task widths cannot exceed that of the widest homogeneous block. As shown in Figure 7-1, the maximum supported task width is *b*. Also, a second problem is that a task may require a memory block on one side, so it cannot be placed in any free spot on the chip but must be placed next to a memory block. In a homogeneous structure, these considerations were absent.

As shown in Figure 7-1, several columns of memory blocks exist between areas of CLBs. A task may require memory blocks in addition to CLBs. In different structures we may have embedded processors or other components instead of memory blocks, but the idea of scheduling tasks on a heterogeneous structure will be the same. For the generality of the problem, we call

these columns as *special columns*. In the next section we define the problem of scheduling tasks and explain how we test the schedulers designed in previous chapters on a heterogeneous structure.

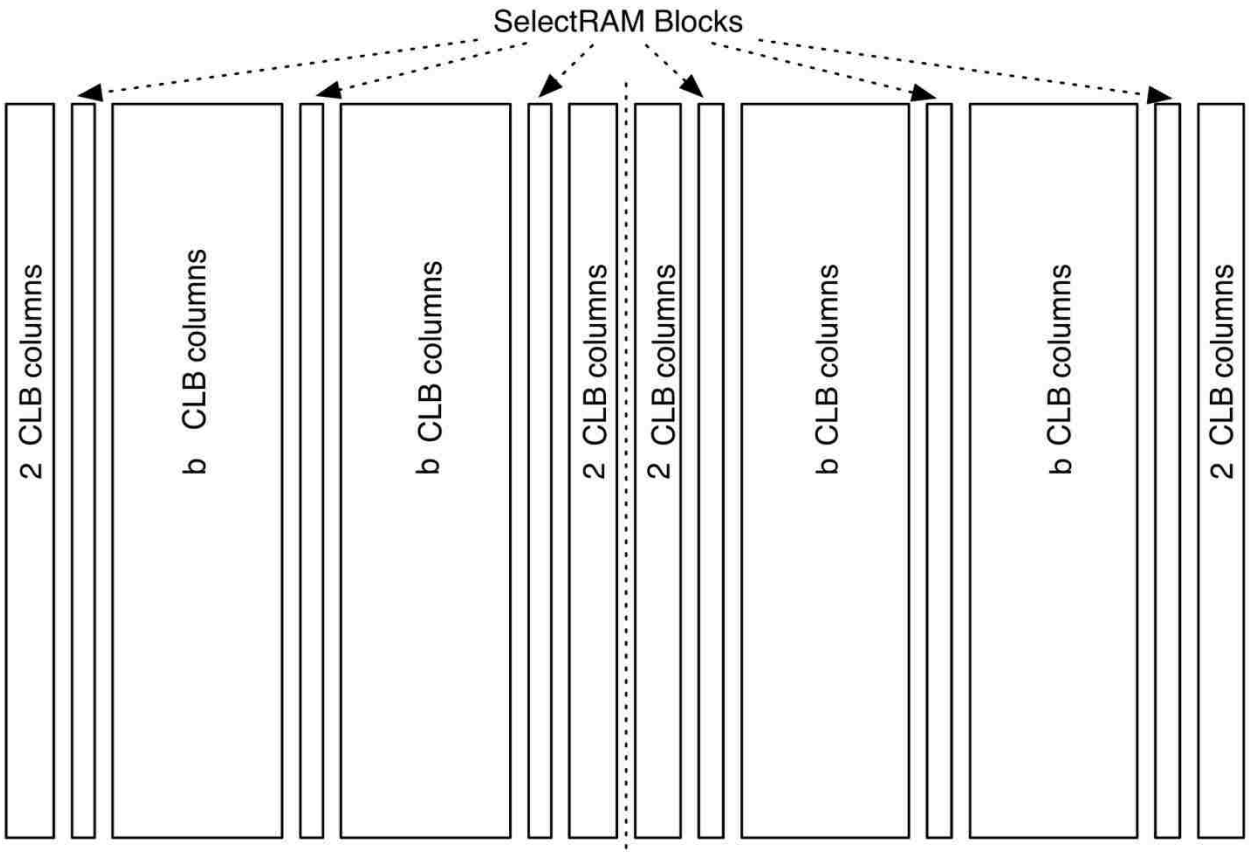


Figure 7-1 Chip structure in Xilinx XC2V series [X07]

7.2. Problem Definition and Experimental Parameters

In this section we start first by explaining the problem of scheduling real-time tasks on a heterogeneous FPGA. There are some differences with scheduling on a homogeneous structure. For a heterogeneous structure, the scheduler has to satisfy not only the CLB requirement of the task but also the special column requirement. Also, the scheduler must keep information not only about free CLBs but also about free special columns and whether they are next to free CLBs. We

assume that all tasks are rectangular. Incoming tasks can be in the following three types based on the need for special columns.

- C-type task: task that does not require any special blocks, so it needs only CLBs.
- CS-type task: task that needs a special column in the rightmost column. The remaining columns of the task will be CLBs.
- SC-type task: task that needs a special column in the leftmost column. The remaining columns of the task will be CLBs.

Tasks are randomly generated from all three types. As we explain at the end of this section, the percentage of special tasks to the C-type tasks defines the special load on the reconfigurable FPGA. As discussed in previous chapters, we use chip load to measure how much demand a task set places on the FPGA in each simulation. The chip load definition is analogous for both heterogeneous and homogeneous structures. The chip load for homogeneous FPGA defines the demand for resources; resources in this case are CLBs. In the heterogeneous case, the chip load again defines the demand for resources, and it does not distinguish between CLB and special cell. A task is defined by the height and the width of the required CLB area. So when we say a 9 × 6 task, this means the task requires a CLB area of 9 × 6. If a 9 × 6 task is a CS-type (SC-type) task, then this means that the task requires a 9 × 6 CLB area and a special column to the right (left) of the CLB area. When calculating chip load, a 9 × 6 CS-type or a 9 × 6 SC-type is considered as a 10 × 6 task.

$$Chip\ load\ = \frac{\sum_{all\ C-type\ tasks}\ w_i \cdot h_i \cdot e_i + \sum_{all\ CS-type\ and\ SC-type\ tasks}\ (w_i + 1) \cdot h_i \cdot e_i}{w \cdot h \cdot t_{max}}$$

where

- h_i, w_i are the height and width of task i,
- e_i is the execution time of task i,

- h, w are the chip height and width,
- t_{max} is the latest time a task is expected to arrive.

For a heterogeneous FPGA, chip load reflects task demands for CLBs but not necessarily demands for special columns. To explain the reason behind this claim, assume we have two sets of tasks with identical chip load, but all tasks in the first set are of C-type and all tasks in the second set are of CS-type or SC-type. The two sets generate the same demand for overall chip resources, but the second set depends on having free memory blocks and not only free CLBs. To measure the load generated by a set of tasks, we use the above chip load definition and we also use *special load*, defined below. Special load measures the special column demand of a set of tasks:

$$Special\ load\ = \frac{\sum_{all\ CS-type\ and\ SC-type\ tasks} h_i \cdot e_i}{h \cdot special\_num \cdot t_{max}}$$

where

- h_i is the height of task i,
- e_i is the execution time of the task i,
- h is the chip height,
- $special\_num$ is the number of special columns, and
- t_{max} is the latest time a task is expected to arrive.

The scheduler will use the MHVS data structure explained in Chapter 6. Despite the fact that we have several CLB areas (eight in Figure 7-2), in the following section, the MHS set (set of all maximal horizontal strips) and the MVS set (set of all maximal vertical strips) cover the free space on the chip the same way described in Chapter 6 assuming that the chip is one homogeneous chip of width equals to the width of all CLB blocks plus the number of special columns. (In later sections, though, when we present modifications to the scheduler, we use a

separate data structure for each block, so we will have eight data structures, each representing one of the eight blocks (see Figure 7-2).) The placer, however, does not ignore the existence of special columns. When considering a strip as a candidate for placement, it checks the location(s) of special column(s) in the strip to determine whether the strip can satisfy the CLB and special column demands of the task.

We use a structure similar to the structure of the Xilinix XC2V6000 to test the work presented in this chapter. The chip size is 96 rows by 88 columns distributed on different blocks as shown in Figure 7-2. (This is the same as the bottom figure in Figure 7-1 with b = 20.) Also, six memory blocks exist as shown in Figure 7-2.

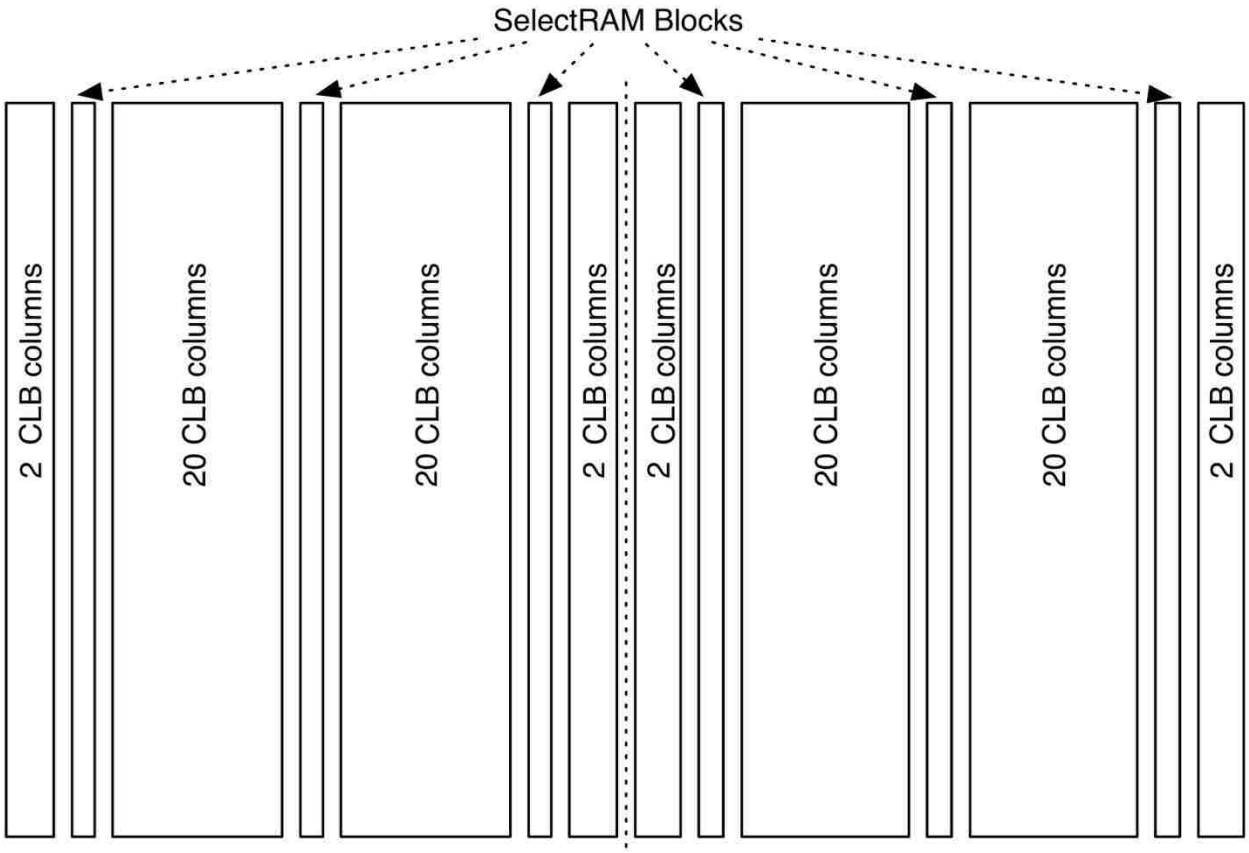


Figure 7-2 Chip structure in Xilinx XC2V6000 [X07]

As we have two different load measures, we will test the scheduler performance when changing one of them and keeping the other one in a small fixed range. To test the performance over different chip loads and different special loads, we test the scheduler based on the parameters given in Table 7-1. The following parameters are fixed and will not change for different chip load or different special load.

- The simulated device is similar to the Xilinx XC2V6000 FPGA, consisting of $96 \times 88 = 6144$ reconfigurable units and six special columns.
- The aspect ratio used in generating the tasks is in the range of [0.2, 5]. Half of the generated tasks will be uniformly distributed in the range of [0.2, 1] and half will be uniformly distributed in the range of [1, 5].
- Task area is in the range of [20, 80] reconfigurable units. Section 7.6 compares all proposed schedulers in this chapter with longer area ranges, specifically [20,200] and [20,400]. Using the aspect ratio assumed above and area range [20, 80] results in task widths up to 20; as the widest block of CLBs has a width of 20, each of these tasks can fit in a block. For larger area sizes, the random task generator can create tasks with width larger than 20 units; we will adjust the width to 20 and adjust the height accordingly to have the same area generated. (This changes the distribution of aspect ratios to permit the desired distribution of task areas.) Of course, if a CS-type or SC-type task has a width larger than 20 units, then this means the task width will be adjusted to 20 and an extra column will be added after that, so the total width will be 21.
- The arrival time is uniformly distributed over the period [1, t_{max}], where t_{max} is the latest time a task is expected to arrive. In our simulations, $t_{max} \in \{300, 500, 950\}$.
- The execution time of each task is in the range of [5, 100] time units.

- Laxity is in the range [1, 50] time units.

Table 7-1 Task parameters to generate different special load at different chip load for area range [20, 80]

	Chip load 0.375 - 0.425		
RUN #	Percentage of special tasks	# of tasks	Special load
1-30	0.15	420	0.0 – 0.1
31-60	0.3	420	0.1 – 0.2
61-90	0.45	420	0.2 – 0.3
91-120	0.55	420	0.3 – 0.4
121-150	0.65	420	0.4 – 0.5
151-180	0.7	420	0.5 – 0.6
181-210	0.8	420	0.6 – 0.7

(a)

	chip load 0.475 - 0.525		
RUN #	Percentage of special tasks	# of tasks	Special load
1-30	0.1	520	0.0 – 0.1
31-60	0.2	520	0.1 – 0.2
61-90	0.35	520	0.2 – 0.3
91-120	0.5	520	0.3 – 0.4
121-150	0.6	520	0.4 – 0.5
151-180	0.65	520	0.5 – 0.6
181-210	0.7	520	0.6 – 0.7

(b)

	chip load 0.675 - 0.725		
RUN #	Percentage of special tasks	# of tasks	Special load
1-30	0.1	730	0.0 – 0.1
31-60	0.2	730	0.1 – 0.2
61-90	0.3	730	0.2 – 0.3
91-120	0.4	730	0.3 – 0.4
121-150	0.5	730	0.4 – 0.5
151-180	0.55	730	0.5 – 0.6
181-210	0.6	730	0.6 – 0.7

(c)

	chip load 0.875 - 0.925		
RUN #	Percentage of special tasks	# of tasks	Special load
1-30	0.05	1000	0.0 – 0.1
31-60	0.1	1000	0.1 – 0.2
61-90	0.2	1000	0.2 – 0.3
91-120	0.3	1000	0.3 – 0.4
121-150	0.35	1000	0.4 – 0.5
151-180	0.45	1000	0.5 – 0.6
181-210	0.5	1000	0.6 – 0.7

(d)

We test the performance based on 210 different runs for each chip-load, where each run has a certain number of tasks based on the chip load required. Each run starts with a fresh empty chip. The performance is averaged over the 210 runs. As shown in Table 7-1, at chip load 0.675 - 0.725, for example, the number of tasks for each of the 210 runs is 730. The total number of tasks in this case using the common parameters mentioned before controls the chip load. To change the special load between different values while keeping the chip load fixed (in a small

range such as between 0.675 and 0.725 and considering this as 0.7), we change the percentage of special tasks to the total number of tasks every 30 runs to represent a new special load. Half of the special tasks SC-type and half CS-type. For example, at chip load 0.675 – 0.725, each of the first 30 runs has 730 tasks with 657 C-type tasks and 73 special tasks. In the example, out of the 73 tasks, 37 tasks are CS-type and 36 tasks are SC-type. The next thirty runs have 584 C-type tasks and 146 special tasks from which 73 tasks are SC-type and 73 tasks are CS-type. The task generator adds an extra column to an SC-type or CS-type task. As we increase the percentage of special tasks, more of these columns are added causing a small difference in chip load, but the chip load stays within the stated range.

7.3. Performance of FF Scheduler on Heterogeneous Structure

In this section, we will test the performance of the FF_YNM scheduler explained in Section 4.1 on a heterogeneous structure to study the effect of the heterogeneity of reconfigurable devices. We call the FF algorithm when applied to a homogeneous structure as the homogeneous first-fit scheduler (HM-FF), and we call the FF algorithm when applied to the heterogeneous structure as the heterogeneous first-fit scheduler (HT-FF).

In HT-FF, the MHVS strips can cover CLBs and special columns. So when a task arrives, the scheduler searches for a strip that has enough area. If the task is a C-type task, the scheduler tests if the strip covers an area of CLBs that is large enough to accommodate the task. In Chapter 6, we placed a task in the bottom-left corner of a free strip, but here the placer tries first to place the task in the bottom-left corner of the strip, but if not enough CLBs exist at the bottom-left corner of the strip, then the placer places the task in the bottom-left corner of a block of CLBs covered by the strip. Also, in HT-FF, when a CS-type or SC-type task arrives, the placer searches

for a strip covering enough CLB area next to special cells. Then the placer places the task according to the special cell locations, starting the search from the bottom-left corner of the strip.

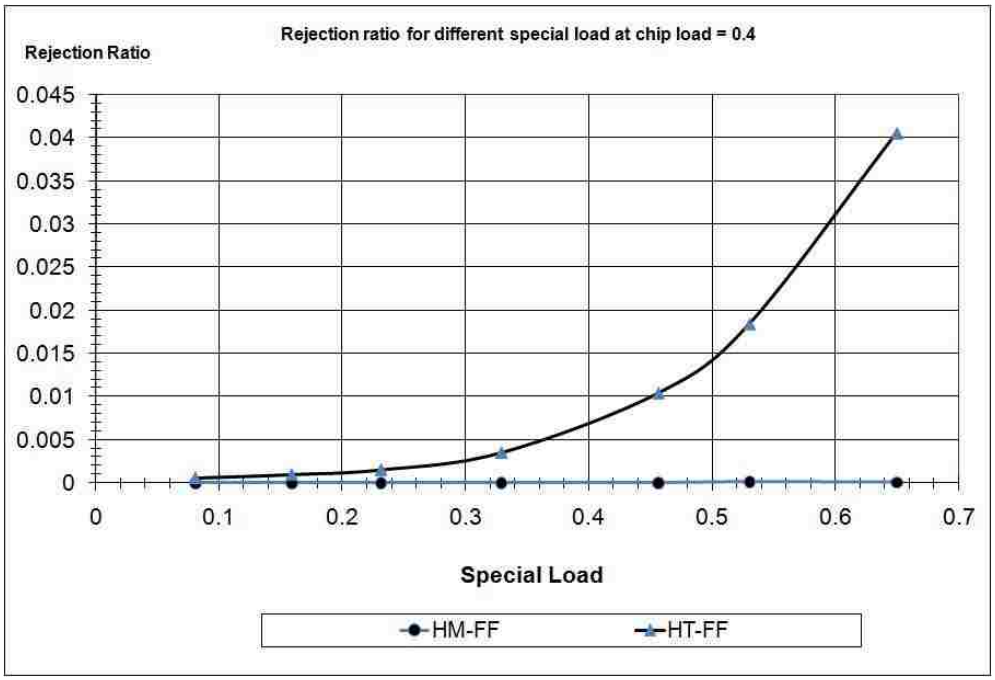
In HM-FF, all special column cells are treated as regular CLBs. HM-FF handles the simulated chip as a homogeneous rectangle of size 96 × 94 CLBs instead of considering it as eight blocks of CLBs (88 columns) and six columns of memory blocks as in the heterogeneous structure. The scheduler considers the task area provided, regardless of its type, as CLBs. All special columns required by CS-type and SC-type tasks considered as extra CLB area. For example, if we have a CS-type or an SC-type task of size 9 × 6, this means the task requires a 9 × 6 CLB area and a special column of height 6, so, for HM-FF we will consider this task as a regular C-type task of size 10 × 6.

Ignoring the special columns in the chip and treating them as regular CLBs and ignoring the request of special tasks for special columns, give us a base line that helps us study the effect of heterogeneity on the scheduler's performance. Also, it guides us to know if the performance shortcomings are due to the scheduler and the inherent scheduling problems or due to the heterogeneity.

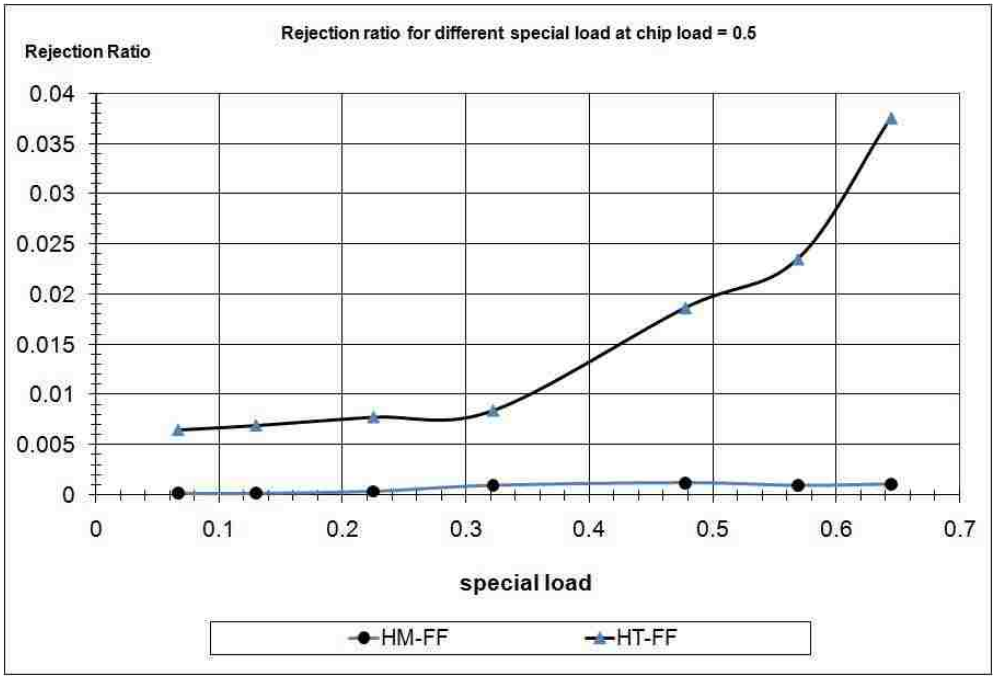
In this section we compare the performance of the same scheduling strategy (FF-YNM) when applied on both homogeneous and heterogeneous structures. We test the performance for different chip loads using parameters shown in Table 7-1. We show that in the heterogeneous structure case both chip load and special load have an effect on the performance of schedulers.

In Figure 7-3(a), the chip load is fixed in a small range (0.375 - 0.425). By increasing the special load we can see that the performance of the scheduler gets worse. By increasing the special load, more tasks request special cells, and the bottleneck at higher special load is the

availability of special cells. At very low special load, we can see that HT-FF was able to perform nearly as well as HM-FF.



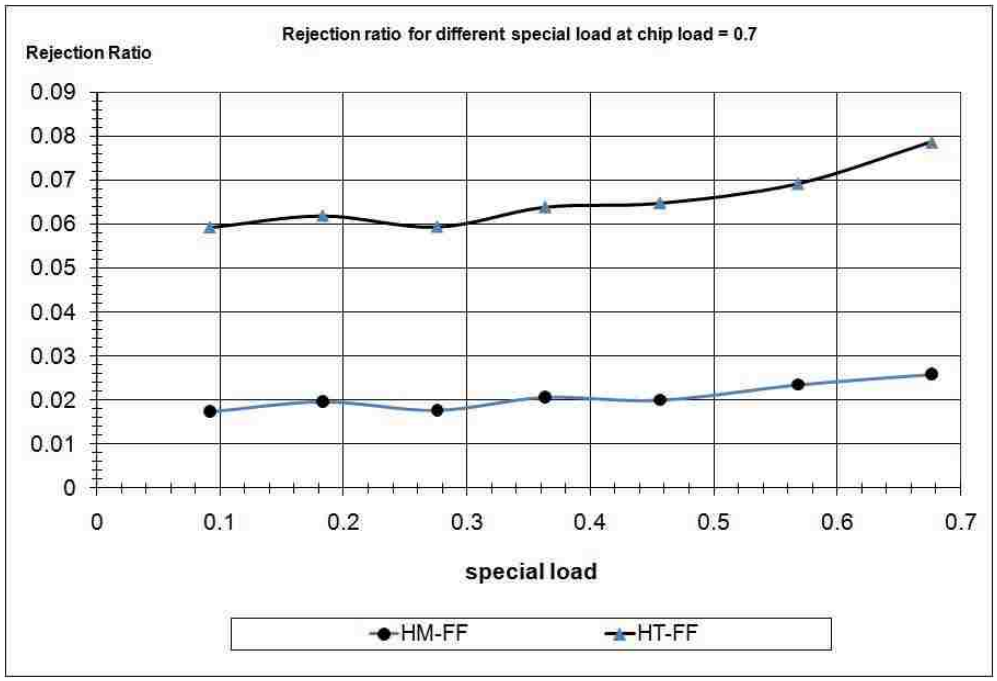
(a) Chip load = 0.4



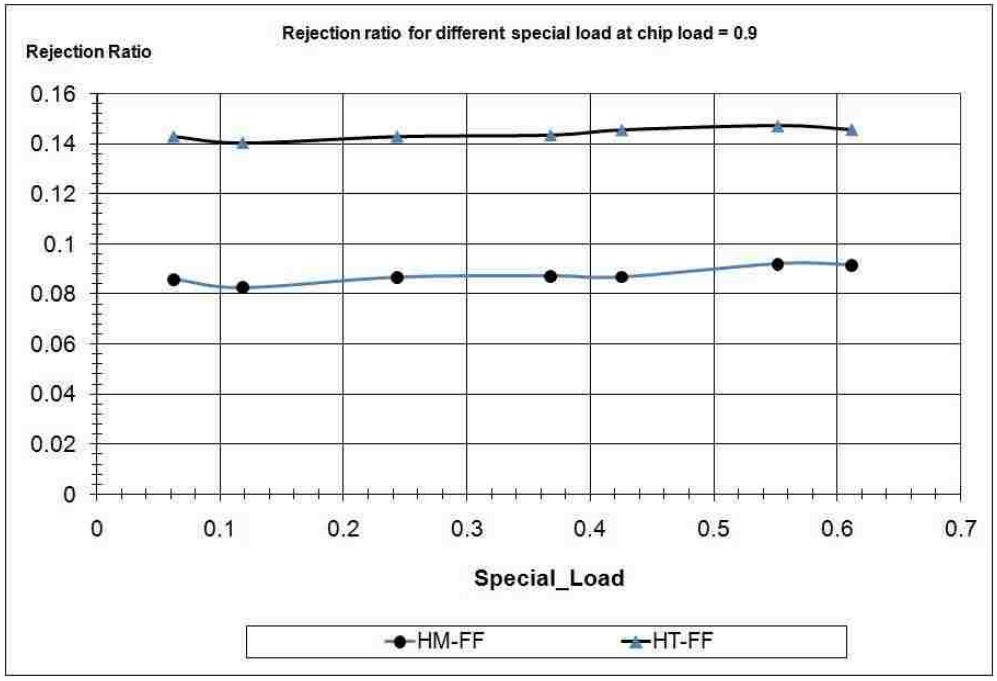
(b) Chip load = 0.5

Figure 7-3 Performance for different chip loads on heterogeneous structure

(Figure 7-3 con'd)



(c) Chip load = 0.7



(d) Chip load = 0.9

By increasing the chip load to 0.5 (Figure 7-3(b)), we can see the same effect on the performance of the HT-FF scheduler when increasing the special load. The scheduler rejects more tasks at higher special load than at lower special load. Also, HT-FF is affected with the chip load increase, at very small special load, HT-FF rejected a small number of tasks even though in the case of applying it to homogeneous structure (HM-FF) the result was to reject a smaller number of tasks and this is due to the effect of chip load on a heterogeneous structure. Note in the same figure, there is a small difference in the performance of HM-FF at very low special load compared to the performance at high special load even though HM-FF treats special columns like CLB columns. The reason is that there is a small increase in the value of the chip load because of the increase in the special load, since HM-FF treats special cells as regular CLBs.

By increasing the chip load to 0.9 (Figure 7-3(d)), we can conclude that at this point the chip load effect became dominant over the special load and that is why the performance of HT-FF is almost flat, as it is for HM-FF. At chip load 0.7 (Figure 7-3(c)), the effect is mixed between the chip load and the special load, though chip load dominates.

Figure 7-4 shows the performance of HT-FF at different chip loads with special load 0.1. From Figure 7-4 we can study the effect of chip load on the performance of schedulers when applied to a heterogeneous structure. To be able to study that effect we fix the special load to 0.1. At this small special load we can assume that the performance will only be affected by the chip load. To understand the reason behind this performance, let us consider an example. Given a task with width 10 and an FPGA with two free areas each of width 5 but in two different blocks separated by a special column, even though the FPGA has a total free CLB area with width 10, it

cannot be used for this task. On the other hand, in a homogeneous structure, this area will be considered as one area so the scheduler would be able to place this task.

Now as shown in the above results, the performance of HT-FF is affected by the special load and the chip load. In the following sections we present some modifications on HT-FF to reduce the effect of special load and get better performance.

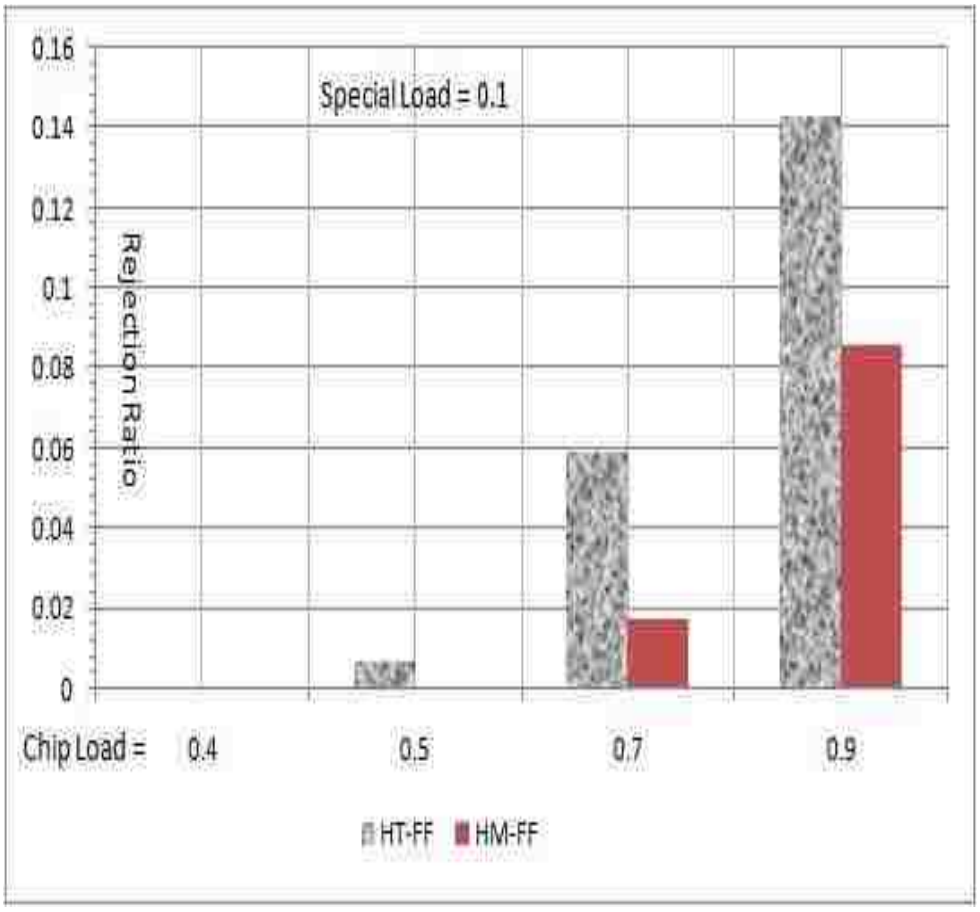


Figure 7-4 Performance for different chip loads at fixed special load 0.1

7.4. Heterogeneous Block Preference Scheduler (HT-P)

In the previous section we showed the effect on the performance of a first-fit scheduler when applied to heterogeneous tasks on a heterogeneous FPGA. In this section, we modify the scheduler specifically for a heterogeneous structure. The proposed scheduler performs better for heterogeneous tasks and FPGA over certain ranges of chip load and special load. We describe

the proposed scheduler in the context of the Xilinx XC2V6000, but the idea is general and applies with simple modifications to other heterogeneous structures. The main idea for this scheduler is similar to the idea presented in Chapter 4 for the region-based scheduler which partitions the chip according to executed task width. Here, we partition the chip based on the task type instead of task width.

In the structure shown in Figure 7-2, the XC2V6000 has eight CLB blocks with six special columns in between. In this structure, a special column separates each two successive CLB blocks. We decided based on this structure to partition the chip into areas to be used for only C-type tasks and other parts of the chip to be used for only CS-type and SC-type tasks. Because this algorithm will have preference on where to place tasks based on task type, we will call it HT-P, standing for heterogeneous scheduler with preferences. Figure 7-5 shows the same chip in Figure 7-2 with the partition marked on each block.

We decided to assign blocks for CS-type and SC-type (S-type) tasks between blocks assigned for C-type tasks, as shown in Figure 7-5. This setup allows all special tasks placed in a block assigned to special tasks to have free access to special cells on both sides. So as shown in Figure 7-5, the general idea is to assign half the blocks for special tasks (we call these as *special blocks*) and the other half to C-type tasks (we will call these as *C-type blocks*). The setup is based on the following observations and conditions.

- Each special block is in the middle of two C-type blocks or in the middle of a C-type block and a chip border. Also, each C-type block is in the middle of two special blocks or a special block and a chip border.

- Each C-type block is in the middle of two special columns or a special column and a chip border, and the C-type tasks that the scheduler places in this C-type block do not use these special columns.
- Each special block has one or two special columns at its borders. If it has only one special column, then the other side of the block must be a chip border. Note: in Figure 7-3, the chip is divided in the middle, so we treat this like a chip border.
- If a special block has a special column to the right but not to the left, then this block is used for only CS-type tasks. Likewise, if the special block has a special column to the left but not to the right, then this block is used for only SC-type tasks.

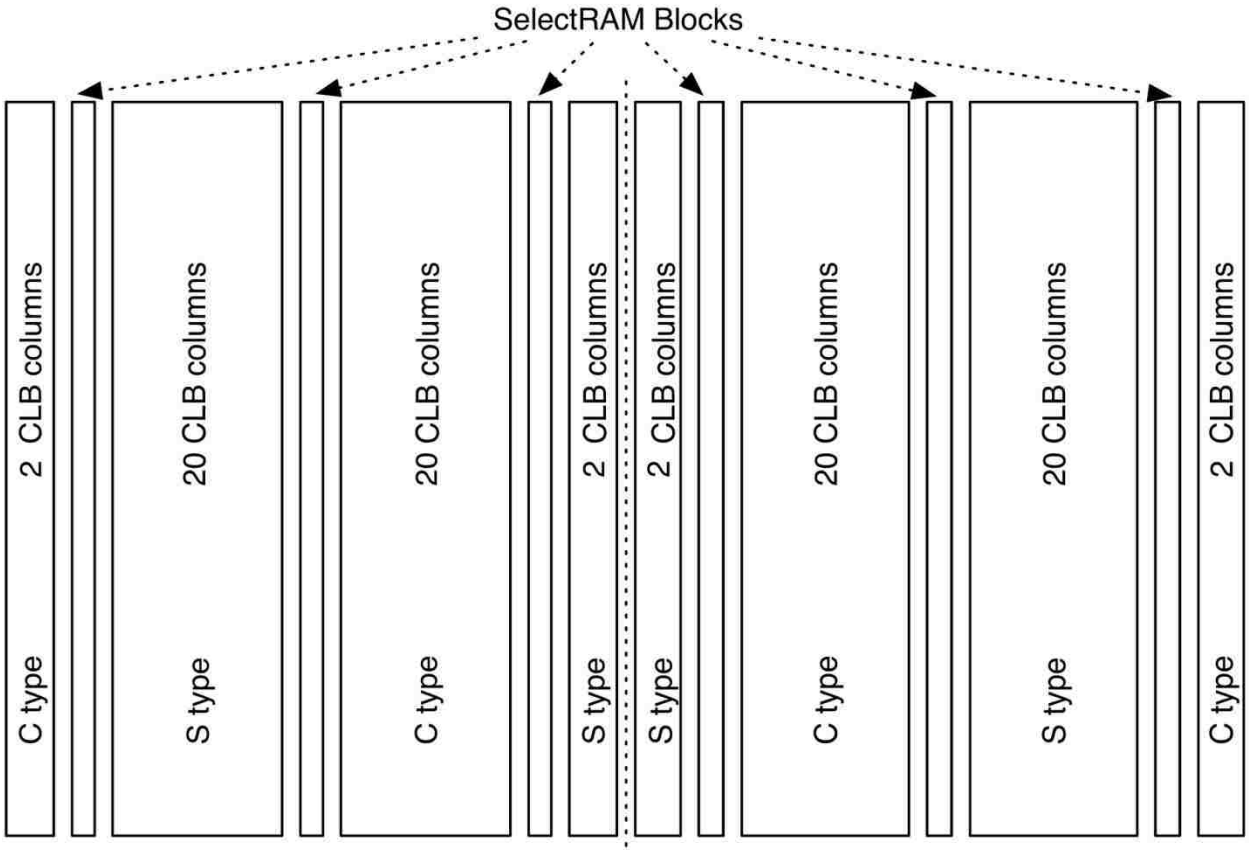


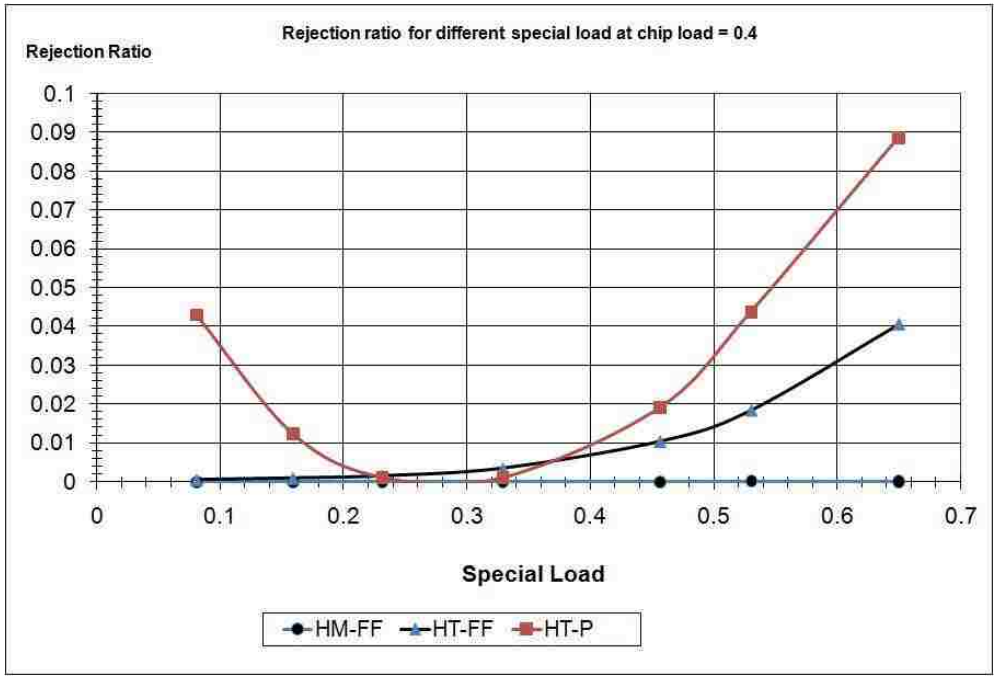
Figure 7-5 Assignment of CLB blocks to task types in HT-P.

When the scheduler receives a task, the first step is to check the task type, and then, based on the type, the scheduler searches the corresponding blocks for a placement. The scheduler deals with each block separately, so each block has its own MHVS free space data structure. In Figure 7-5, we have eight different blocks, so the scheduler has eight MHVS sets. As discussed before, we assign four blocks for C-type tasks and four blocks for special tasks. The algorithm tries first to place the received task in a strip in the first block assigned to the task type, and if no placement was found, then the algorithm searches other blocks assigned for this task type one by one till the last block or till a placement is found or till exhausting the blocks. At the end, if no placement was found in the assigned blocks, then the scheduler handles tasks in the same way as discussed in Chapter 4 by moving them to the pending queue and trying to place them later when free space is available.

We tested the scheduler using the same data sets used in the previous section. Figure 7-6 compares the performance of HT-P versus HT-FF and HM-FF. We have two different resources, CLBs and special cells, and we have two measures for these resources, chip load and special load. Both of them affect the performance of the scheduler. From the results shown in Figure 7-6, HT-P is performing better than HT-FF only in a small window in which the demand for special cells is contributing about equally to the chip load as is the demand for CLBs (that is, special load is about equal to chip load). As we have half of the chip assigned for CLBs and the second half of the chip assigned for special cells, when the demands for one of the resources gets more than the demands for the other one, which means that the demands are out of balance, then one of them is controlling the rejection ratio. So if the special load is low compared to the chip load, then the bad performance is due to high CLB demand, and if the chip load is low compared to the special load, then the bad performance is due to the high special column demand.

At higher chip loads, the performance of the scheduler becomes flat and the reason is that the chip load is dominant and the demand for CLBs is the main factor in the performance and that explains the behavior of the algorithm in Figure 7-6(d) at chip load 0.9. At higher special load, we can see that the demand for different type of resources is balanced again and the performance is almost the same between HT-FF and HT-P.

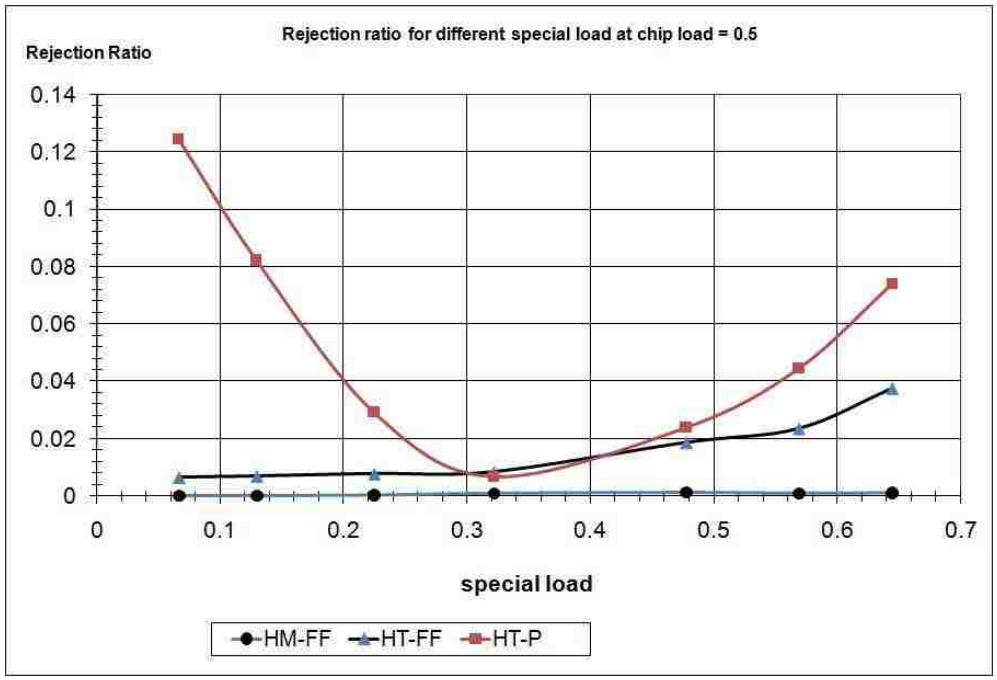
We can conclude from these results that by assigning certain blocks to special tasks and certain blocks for C-type tasks, the HT-PP scheduler was able to perform as well as HT-FF if the demands for different types of resources are balanced. Still, HT-P was not able to perform better than HT-FF except in these small intervals. In the next section we modify the scheduling techniques of HT-P to achieve better performance than HT-FF.



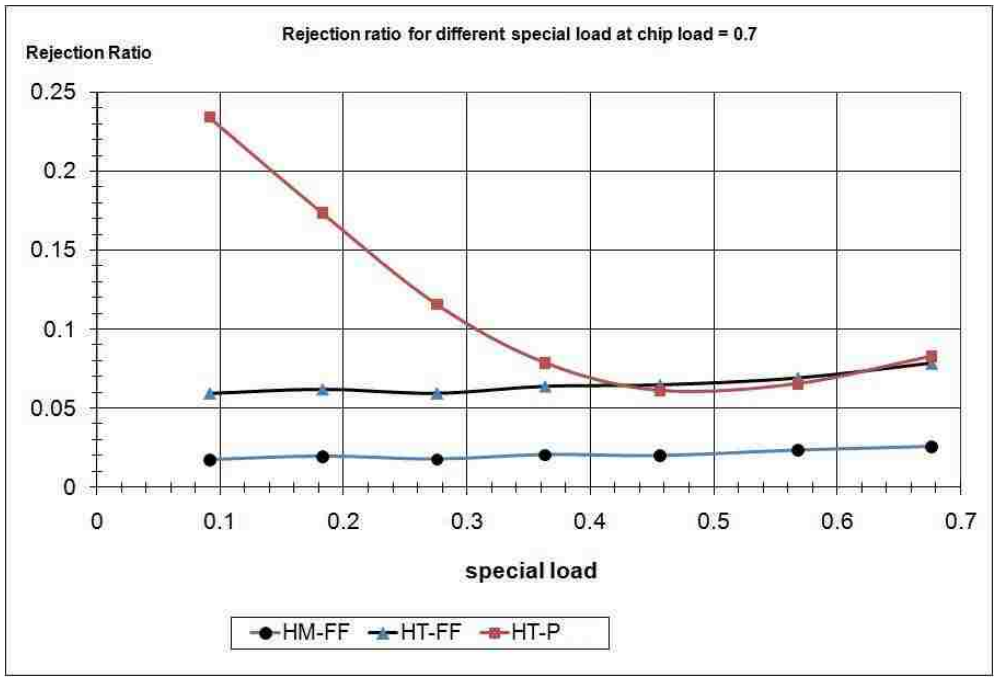
(a) Chip load = 0.4

Figure 7-6 Results of HT-P compared to HT-FF and HM-FF

(Figure 7-6 con'd)

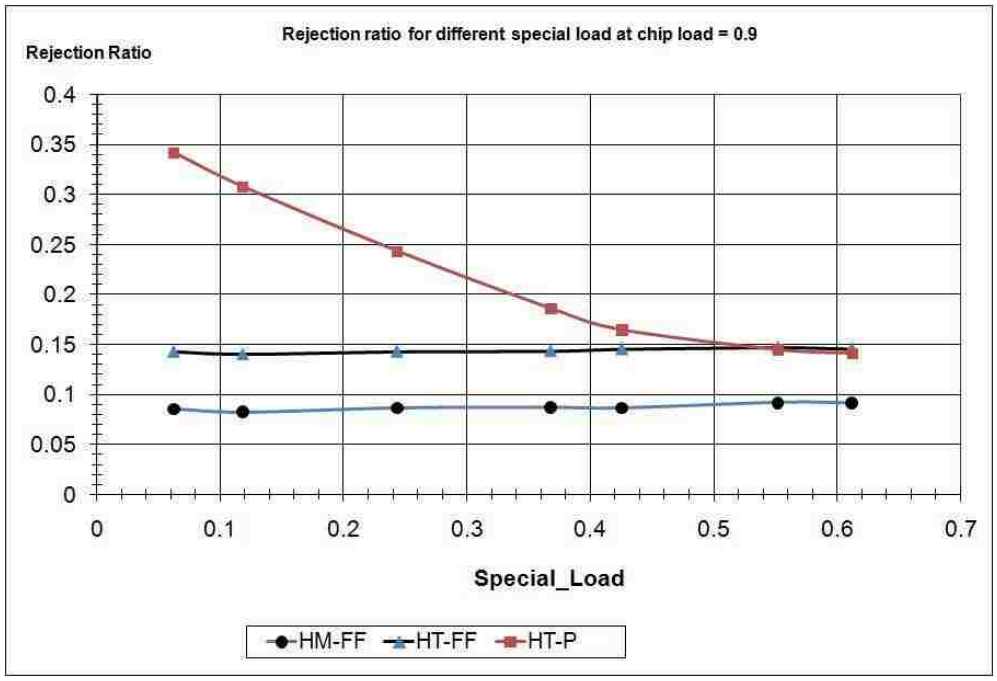


(b) Chip load = 0.5



(c) Chip load = 0.7

(Figure 7-6 con'd)



(d) Chip load = 0.9

7.5. Heterogeneous Block Preference Scheduler with Sharing (HT-PS)

From results shown in the previous section, we can see that HT-P has a problem when the load due to special tasks is not balanced with the load due to C-type tasks. If we have more load due to C-type tasks than load due to special tasks, then one way to solve this problem is to allow C-type tasks to be placed in blocks reserved for special tasks, And if we have more load due to special tasks than load due to C-type tasks, then one way to solve this problem is to allow special tasks to be placed in blocks reserved for C-type tasks. The concept of sharing resources was successful, as we show in this section, enhancing the performance of HT-P.

The *heterogeneous block preference scheduler with sharing* (HT-PS) scheduler works the same way the HT-P scheduler works with only one difference. As before, when a task is received, the scheduler tries to place the task in one of the blocks assigned for the task type. If it finds no placement, however, then the scheduler tries to place the task in blocks of the other type,

sharing one type of block with the other type of task. If no placement was found in any block, then the scheduler moves the task to the pending queue. So HT-PS retains block assignments to types, but these are preferences and allow restricted sharing of the same block by different type tasks.

Placing a C-type task in a special block is easy. The scheduler searches the special block and ignores the special columns associated with this block. Placing a special task in a block reserved for C-type, however, is more complicated, requiring the following.

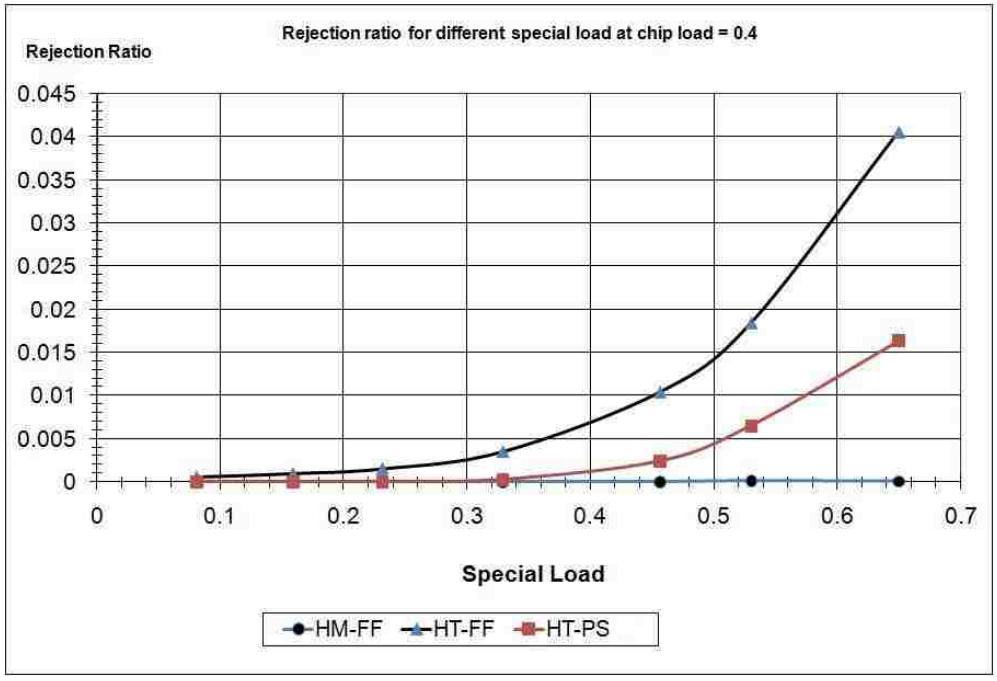
- The block has a strip that can accommodate the task.
- The strip has to be next to a special column.
- The special cells needed by the task must be available in the data structure for the adjacent block that owns the special column.

To test these conditions, the scheduler checks the MHVS data structure for the C-type block, then, if it finds appropriate space, it checks the MHVS data structure for the adjacent special block to find if the corresponding special cells are free. The update to the special block free space can proceed as if for a task of width one and the original task's height.

Figure 7-7 shows the results for HT-PS compared to HT-FF and HM-FF. From Figure 7-7, we can see that HT-PS is performing better than HT-FF across a range of chip loads and special loads. The results show that HT-PS was able to overcome the problems of HT-P. From results in Figure 7-7 we can conclude that the performance of HT-PS is better than HT-FF for all different chip loads and over all ranges of special load. We can also conclude that HT-PS has the same trend as HT-FF in being affected by both the chip load and the special load, regardless of whether the demand for resources is balanced or out of balance. We can see that at low chip load

the performance gets bad at higher special load and at high chip load the performance becomes almost flat because chip load is dominant.

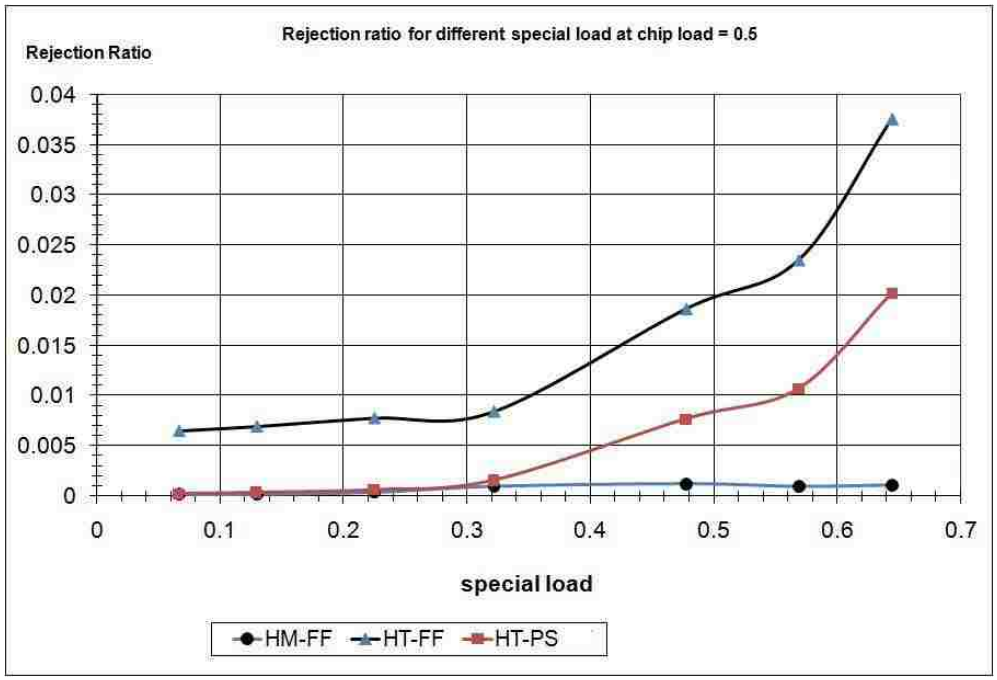
The results in this section support the assumption that we and other researchers used which is that studying a homogeneous structure is a valid step to understanding scheduling on heterogeneous devices, as scheduler ideas designed for a homogeneous structure can result in good performance with some small and direct modifications. Our modifications in this chapter were not in the scheduling or the data structures used but were on how to deal with the chip. We think that these results will open the door for researchers in the future to target the heterogeneous structure and modify their proposed schedulers to be able to apply them on a heterogeneous structure. In the next section we will show even better performance when we test HT-PS on higher area ranges.



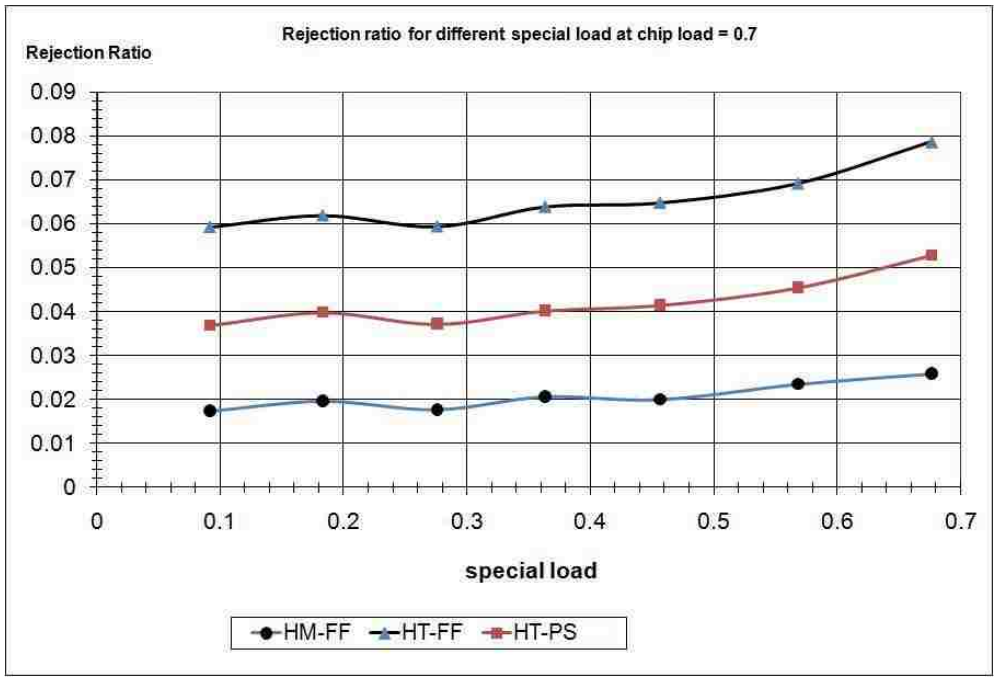
(a) Chip load = 0.4

Figure 7-7 Results for HT-PS compared to HT-FF and HM-FF

(Figure 7-7 con'd)

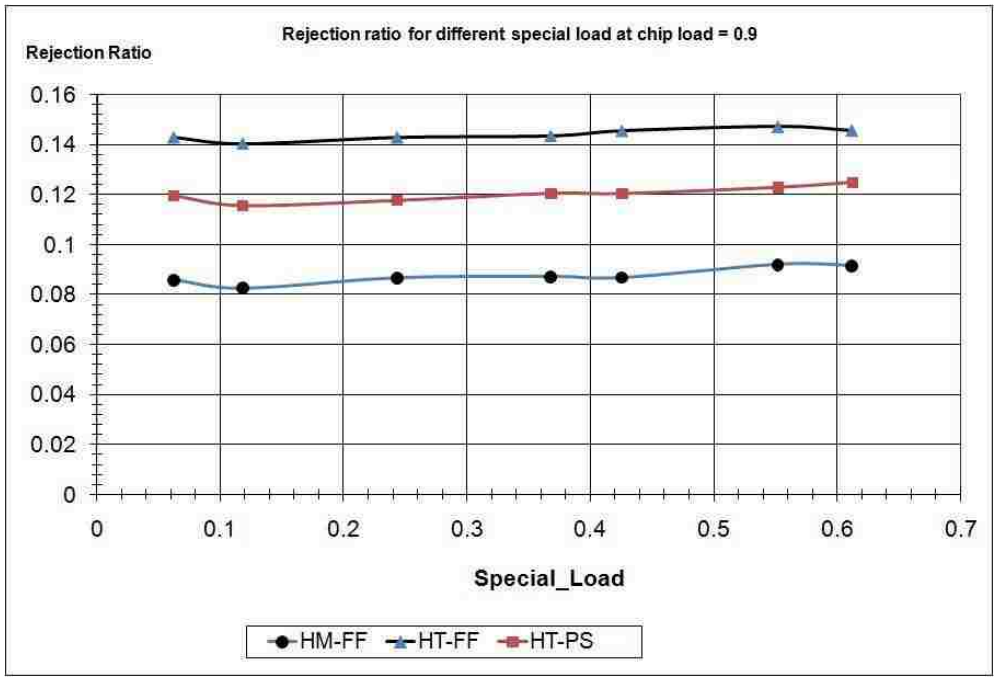


(b) Chip load = 0.5



(c) Chip load = 0.7

(Figure 7-7 con'd)



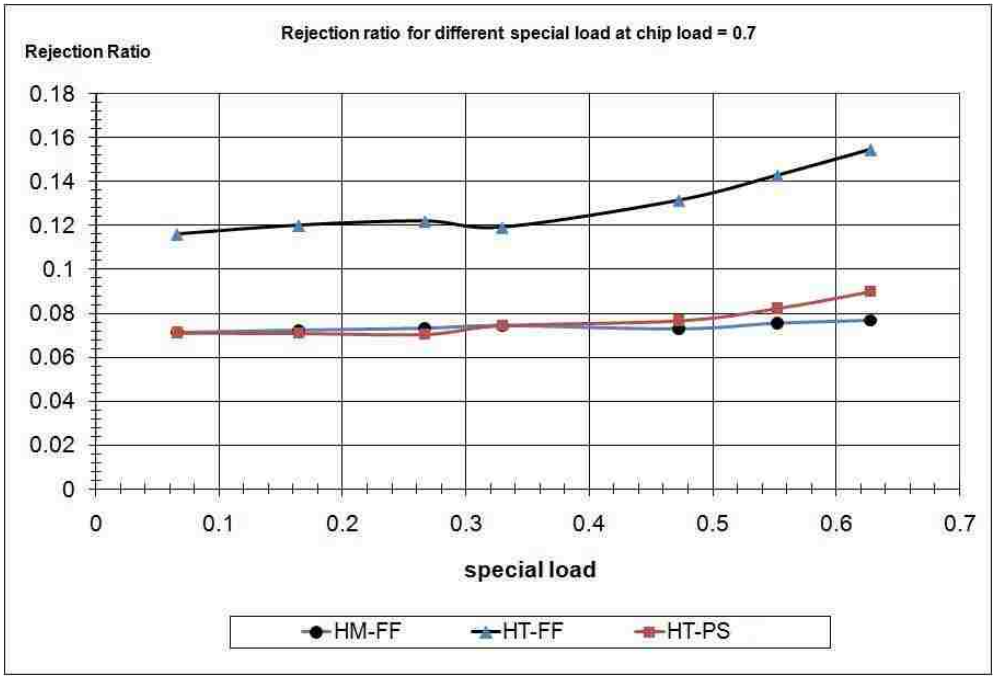
(d) Chip load = 0.9

7.6. Simulation Results for Higher Area Ranges

In this section we test the HT-FF and HT-PS schedulers using higher data ranges, specifically, [20, 200] and [20, 400]. Using the aspect ratio of [0.2, 5] and these ranges generates some tasks with width more than 20 units. The simulated chip XC2V6000 has eight blocks but the biggest blocks are 20 units wide. To constrain simulated tasks, if the randomly generated task width is bigger than 20, then we reduce the width of the task to 20 units and adjust the height of the task to have the same area. Consequently, all tasks arriving at the scheduler have task widths up to 20 units and so can fit in some block.

Figure 7-8 compares the results for area range [20, 200]. The HT-PS scheduler has a very good performance and was able to overcome the obstacles of heterogeneous structure and tasks, performing as well as HM-FF, which ignores all special columns and task requests for special columns. The reason behind the better performance of HT-PS in this area range and worse

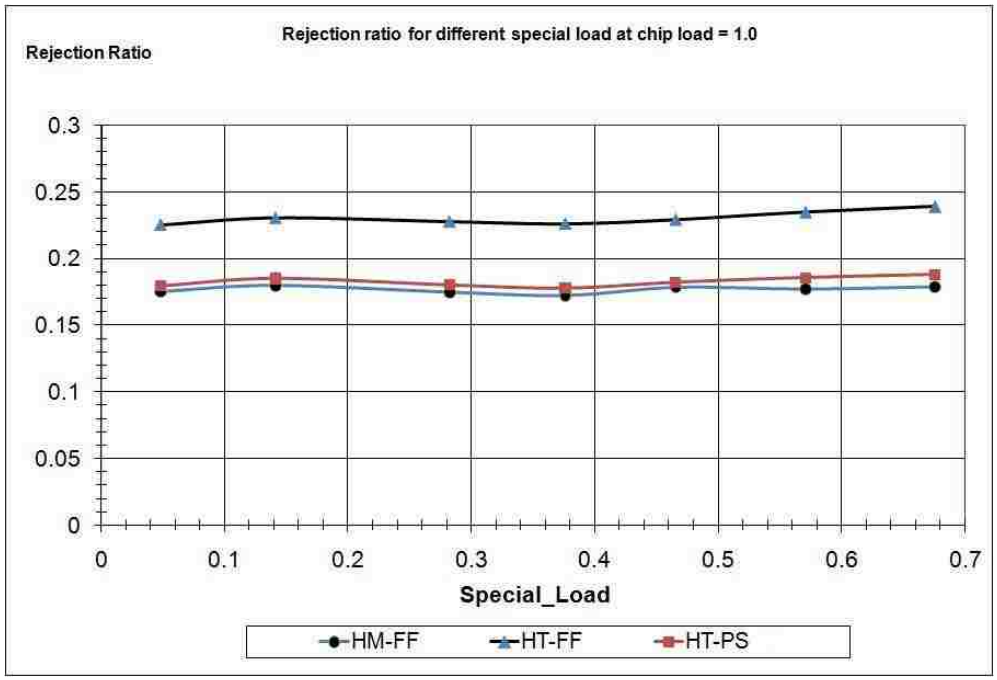
performance in the area range [20, 80] is that using higher area ranges results in randomly generated tasks with widths more than 20 units that the task generator will change to width 20. The HT-PS scheduler stacks many of these tasks on top of each other, fitting exactly the width of the blocks. This results in less fragmented free space left after placing tasks. For example, using area range of [20, 200] and aspect ratio [0.2, 5] and by splitting tasks equally in this aspect ratio as explained before in Section 7-2, the scheduler may receive up to 20% of the total number tasks in each run with width equal to 20 units. Using the same parameters but area range of [20, 400], the scheduler may receive up to 33% of the total number of tasks in each run with width equal to 20 units. Figure 7-9 shows the results obtained from testing the schedulers on area range [20, 400]. The HT-PS scheduler was able to generate better results even than HM-FF, for the reasons explained above.



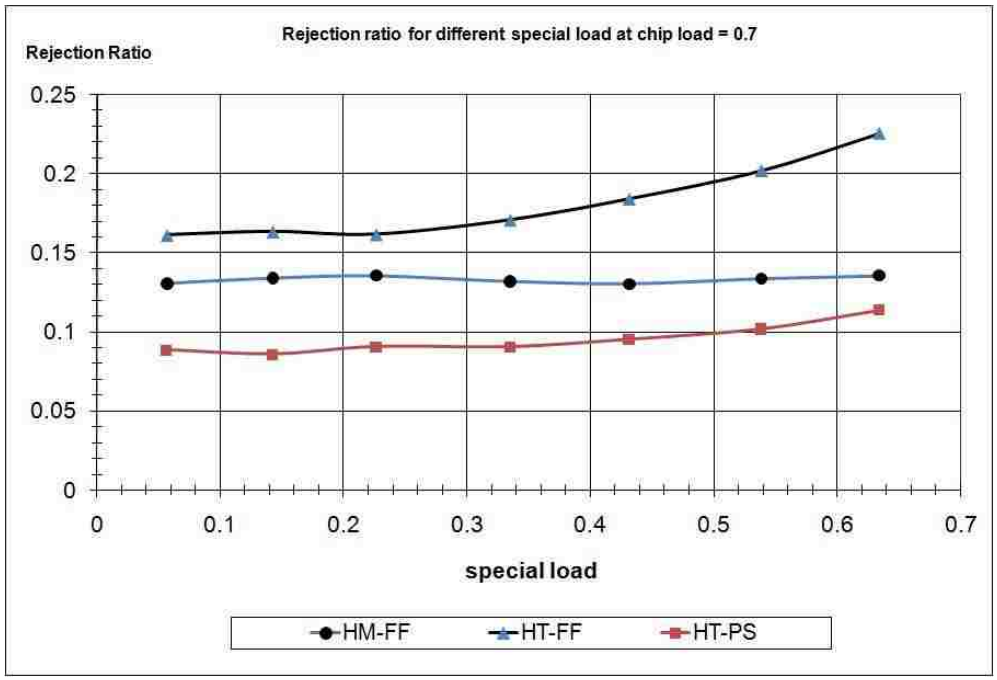
(a) Chip load = 0.7

Figure 7-8 Results for area range [20, 200]

(Figure 7-8 con'd)



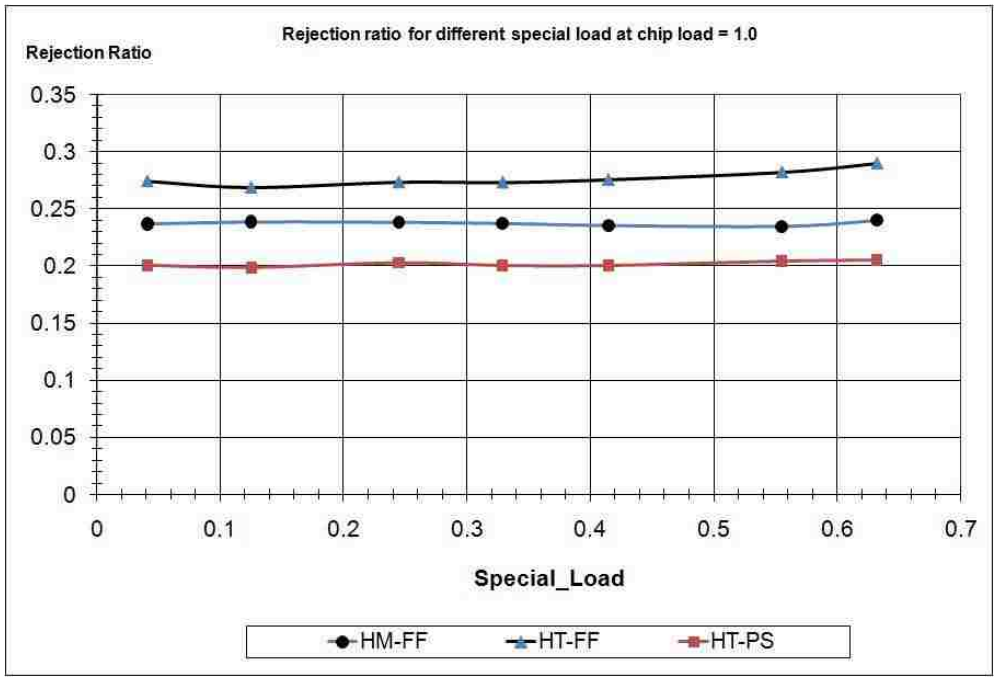
(b) Chip load = 1.0



(a) Chip load = 0.7

Figure 7-9 Results for area range [20, 400]

(Figure 7-9 con'd)



(b) Chip load = 1.0

CHAPTER 8: CONCLUSION

In this research, we studied several problems in the area of scheduling real-time tasks on reconfigurable devices: real-time task scheduling; free space management; and scheduling real-time tasks on heterogeneous FPGAs. In the area of scheduling we proposed an algorithm based on dividing the chip into different regions and assigning tasks to regions based on their width. We proved that this algorithm will have a better rejection ration than FF_YN and the same rejection ratio as FF-YNM with a faster running time.

With respect to the regions idea proposed in Chapter 4, one of the open problems is how to find the optimal or the best values for the bases value. There are several open problems of interest in the area of scheduling. The problem of scheduling tasks with precedence constraints or priority between them has received little attention. Scheduling a dependent set of tasks requires rejecting a task if it is dependent on a task that was rejected earlier. Also, consider scheduling applications, where each application comprises a set of dependent tasks. Because the scheduler rejects the whole application if one task was rejected, one suggestion to solve this problem is to give the application that has fewest tasks left a higher priority than other applications.

Another open problem in the scheduling area, derives from the fact that most work assumed schedulers run on a host computer. The time analysis in the area was based on reporting the time complexity of the proposed algorithm. It is a good way to compare different algorithms but it does not give a sense of real execution time and costs on RC devices. Testing the proposed algorithm on an embedded processor inside an FPGA is important to guide the categories of applications in which RC devices can be used. Also, a related problem is measuring the power

requirements for algorithms. The power consumption will affect the ability of using RC devices in handheld units like cell phones.

In the area of free space management, we were able to propose a free space data structure composed of $O(n)$ strips and the search time for a scheduler that uses MHVS set is $O(n)$. Also, we proved that the data structure can be updated after an insert or delete action in $O(n)$ time. The simulations showed a good comparison compared to other data structures like MERs even though MHVS is not recognition complete. One of the open problems is to study how to enhance the results of the MHVS to be equal to the performance of MERs with losing the feature of fast time complexity.

Very few researchers studied the problem of scheduling on a heterogeneous structure. We have a promising algorithm, and there are several open problems to study on the effect of heterogeneity on the performance of schedulers designed for homogeneous structures and developing schedulers specifically for heterogeneous FPGAs.

REFERENCES

[ABBT04] A. Ahmadinia, C. Bobda, M. Bednara, and J.Teich (2004), "A New Approach for On-Line Placement on Reconfigurable Devices," *Proc. 2004 Reconfigurable Architectures Workshop (RAW'04 – at IPDPS'04)*.

[ABF04-1] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. Van der Veen (2004), "Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices," Paper cs.DS/0406035, http://arXiv.org/abs/cs/0406035.

[ABF04-2] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. Van der Veen (2004), "Optimal Routing-Conscious Dynamic Placement for Reconfigurable Devices," Proc. 2004 Int'l. Conf. Field-Programmable Logic and Applications,

[ABF07] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. van der Veen (2007), "Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices," IEEE Trans. Comput., vol. 56, no. 5, pp. 673-680.

[ABT03] A. Ahmadinia, C. Bobda, and J. Teich (2003). "Temporal Task Clustering for Online Placement on Reconfigurable Hardware," Proc. IEEE Int'l. Conf. Field Programmable Technology, pp 359-362.

[AFK08] E. Anderson, M. French, D. Kang (2008), "System on a Programmable Chip Adaptation Through Active Partial Reconfiguration," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms.

[AT03] A. Ahmadinia and J. Teich (2003), "Speeding up Online Placement for XILINX FPGAs by Reducing Configuration Overhead," Proc. IFIP Int'l. Conf. VLSI-SOC, pp. 118-122.

[B96] G. Brebner (1996), "A Virtual Hardware Operating System for the Xilinx XC6200," Proc. 6th Int'l. Workshop Field-Programmable Logic and Applic. (Lect. Notes Comput. Sci. #1142), pp. 327-336.

[BHB07] L. Braun, M. Hubner, J. Becker, T. Perschke, V. Schatz, S. Bach (2007), "Circuit Switched Run-Time Adaptive Network-on-Chip for Image Processing Applications," International Conference on Field Programmable Logic and Applications, FPL Page(s):688 – 691.

[BHH07] J. Becker, M. Hübner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka (2007), Dynamic and Partial FPGA Exploitation, Proceedings of the IEEE 95, PP. 438-452.

[BJS04] K. Baskaran, W. Jigang, and T. Srikanthan (2004), "Hardware Partitioning Algorithm for Reconfigurable Operating System in Embedded Systems," Proc. 6th Real-Time Linux Workshop, pp. 117-123.

{available: www.linuxdevices.com/files/rtlws-2004/ KrishnamoorthyBaskaran-HWPart.pdf}

[BKS00] K. Bazargan, R. Kastner, and M. Sarrafzadeh (2000), "Fast Template Placement for Reconfigurable Computing Systems," IEEE Design & Test of Computers, vol. 17, no. 1, pp. 68-83.

[C83] B. Chazelle (1983), "The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation," IEEE Trans. Comput., vol. C-32, no. 8, pp. 697-707.

[CC99] G.-M. Chiu and S.-K Chen (1999), "An Efficient Submesh Allocation Scheme for Two-Dimensional Meshes with Little Overhead", IEEE Trans. Par. Distr. Sys. 10, 471-486.

[CDHG07] J. Cui, Q. Deng, X. He, and Z. Gu (2007), "An Efficient Algorithm for Online Management of 2D Area of Partially Reconfigurable FPGAs," Proc. Conf. Design Automation & Test in Europe.

[CGG08] C. Conger, A. Gordon-Ross, and A. George (2008), "FPGA Design Framework for Partial Run-Time Reconfiguration," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms, pp. 122-128.

[CGLD07] J. Cui, Z. Gu, W. Liu, and Q. Deng (2007), "An Efficient Algorithm for Online Soft Real-Time Task Placement on Reconfigurable Hardware Devices," Proc. 10th IEEE Int'l. Symp. Object and Component-Oriented Real-Time Distributed Computing (ISORC '07), pp. 321-328.

[CH05] Y.-H. Chen and P.-A. Hsiung (2005), "Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC," Proc. Int'l. Conf. Embedded and Ubiquitous Computing (Lect. Notes Comput. Sci. #3824), pp. 489-498.

[CLC02] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck (2002), "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing," IEEE Trans. VLSI Sys., vol. 10, no. 3, pp. 209-220.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein (2001), Introduction to Algorithms, 2nd ed., McGraw-Hill, Boston, MA.

[CW98] J. Csirik and G. Woeginger (1998), "On-Line Packing and Covering Problems," online algorithms, Lecture Notes in Computer Science 1442, Berlin, Springer, PP., 147-177.

[D92] A. Datta (1992), "Efficient Algorithms for the Largest Rectangle Problem", Information Sciences 64 pp. 121-141.

[DE97] O. Diessel and H. ElGindy (1997), "Run-Time Compaction of FPGA Designs," Proc. 7th Int'l Workshop on Field-Programmable Logic and Applications (FPL '97) (Lect. Notes Comp. Sci. #1304), pp. 131-140.

[DE01] O. Diessel and H. ElGindy (2001), "On Dynamic Task Scheduling for FPGA-based Systems," Int'l. J. Foundations of Computer Science, vol. 12, no. 5, pp. 645-669.

[DEMSS00] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt (2000), "Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs," IEE Proceedings—Computers and Digital Techniques, vol. 147, no. 3, pp. 181-188.

[DP05-1] K. Danne and M. Platzner (2005), "A Heuristic Approach to Schedule Periodic Real-Time Tasks on Reconfigurable Hardware," Proc. Int'l. Conf. Field Programmable Logic and Application, pp. 568-573.

[DP05-2] K. Danne and M. Platzner (2005), "Periodic Real-Time Scheduling for FPGA Computers," Proc. 3rd Int'l. Workshop on Intelligent Solutions in Embedded Systems, pp. 117-127.

[DMP06] K. Danne, R. Mühlenbernd, and M. Platzner (2006), "Executing Hardware Tasks on Dynamically Reconfigurable Devices under Real-time Conditions," Proc. 16th Int'l. Conf. Field Programmable Logic and Applications (FPL), pp. 1-6.

[DP06-1] K. Danne and M. Platzner (2006), "Partitioned Scheduling of Periodic Real-Time Tasks onto Reconfigurable Hardware," Proc. 2006 Reconfigurable Architectures Workshop (RAW'06 at IPDPS'06).

[DP06-2] K. Danne and M. Platzner (2006), "An EDF Schedulability Test for Periodic Tasks on Reconfigurable Hardware Devices," Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems (LCTES).

[DW99] O. Diessel and G. Wigley (1999), "Opportunities for Operating Systems Research in Reconfigurable Computing," Technical Report ACRC-99-018, School of Comput. and Info. Sci., Univ. of South Australia.

[EES07-01] A. A. ElFarag, H. M. El-Boghdadi, and S. I. Shaheen (2007), "Miss Ratio Improvement For Real-Time Applications Using Fragmentation-Aware Placement," Proc. 2007 Reconfigurable Architectures Workshop (RAW'07 at IPDPS'07).

[EES07-02] A. A. El Farag, H. M. El-Boghdadi, and S. I. Shaheen (2007), "Improving Utilization of Reconfigurable Resources Using Two Dimensional Compaction," Proc. Conf. Design Automation and Test in Europe (DATE'07), pp. 135-140.

[FKS08] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. C. van der Veen, J. Angermeier, D. Koch, and J. Teich (2008), "No-Break Dynamic Defragmentation of Reconfigurable Devices," Proc. Int'l. Conf. Field Programmable Logic and Applications, pp. 113-118.

[FKT01] S. Fekete, W. Kohler, and J. Teich (2001), "Optimal FPGA Module Placement with Temporal Precedence Constraints," Proc. Design Automation and Test in Europe Conf., pp. 658-665.

[FVA08] S. P. Fekete, J. C. van der Veen, A. Ahmadinia, D. Gohringer, M. Majer, and J. Teich (2008), "Offline and Online Aspects of Defragmenting the Module Layout of a Partially Reconfigurable Device," IEEE Trans. Very Large Scale Integration Systems, vol. 16, no. 9, pp. 1210-1219.

[GASF02] M. G. Gericota, G. R. Alves, M. L. Sila, and J. M. Ferreira (2002), "On-line Defragmentation for Run-Time Partially Reconfigurable FPGAs," Proc. 12th Int'l. Workshop Field-Programmable Logic and Applic. (Lect. Notes Comput. Sci. # 2438), pp. 303-311.

[GJ79] M. R. Garey and D. S. Johnson (1978), "Computers and Intractability: A Guide to the Theory of NP-Completeness", H. Freeman, San Francisco.

[HPLD08] J. Huang, M. Parris, J. Lee, and R. F. DeMara (2008), "Scalable FPGA Architecture for DCT Computation Using Dynamic Partial Reconfiguration," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms, pp. 269-272.

[HSB06] M. Hübner, C. Schuck, and J. Becker (2006), "Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs," Proc. 2006 Reconfigurable Architectures Workshop (RAW'06 at IPDPS'06).

[HV04-1] M. Handa and R. Vemuri (2004), "Area Fragmentation in Reconfigurable Operating Systems," Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA04), pp. 77-83.

[HV04-2] M. Handa and R. Vemuri (2004), "An Efficient Algorithm for Finding Empty Space for Online FPGA Placement," Proc. 41st Design Automation Conf. (DAC'04), pp. 960-965.

[HV04-3] M. Handa and R. Vemuri (2004), "Hardware Assisted Two Dimensional Ultra Fast Placement," Proc. 2004 Reconfigurable Architectures Workshop (RAW'04 — at IPDPS'04).

[HV04-4] M. Handa and R. Vemuri (2004), "An Integrated Online Scheduling and Placement Methodology," Proc. 2004 Int'l. Conf. Field-Programmable Logic and Applications (Lect. Notes Comput. Sci. #3203), pp. 444-453.

[KKP06] M. Koester, H. Kalte, and M. Porrmann (2006), "Relocation and Defragmentation for Heterogeneous Reconfigurable Systems," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA'06), pp. 70-76.

[KPR04] H. Kalte, M. Porrmann, and U. Rückert (2004), "System-on-Programmable-Chip Approach Enabling Online Fine-Grained 1D-Placement," Proc. 2004 Reconfigurable Architectures Workshop (RAW'04 — at IPDPS'04).

[LBM06] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford (2006), "Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs," Proc. 2006 Field Programmable Logic Conf.

[NBR90] S. C. Nandy, B. B. Bhattacharya, and S. Ray (1990), "Efficient Algorithms for Identifying All Maximal Isothetic Empty Rectangles in VLSI Layout Design," Proc. 10th Conf. Foundations of Software Technology and Theoretical Computer Science (Lect. Notes Comput. Sci. #472), 255-269.

[NLH84] A. Naamad, D. T. Lee, and W.-L. Hsu (1984), "On the Maximum Empty Rectangle Problem", Discrete Applied Mathematics 8, 267-277.

[O90] M. Orlowski (1990), "A New Algorithm for the Largest Empty Rectangle Problem", Algorithmica 5, 65-73.

[SBALB05] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, and T. Becker (2005), "Modular Partial Reconfiguration in Virtex FPGAs," Proc. Int'l. Conf. Field Programmable Logic and Applic., pp. 211-216.

[SBB06] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght (2006), "Modular Dynamic Reconfiguration in Virtex FPGAs", IEE Proc. Computers and Digital Techniques 153, 157-164.

[SBHB08] O. Sander, L. Braun, M. Hübner, and J. Becker (2008), "Data Reallocation by Exploiting FPGA Configuration Mechanisms," Proc. Int'l. Workshop on Applied Reconfigurable Computing (Lect. Notes Comput. Sci. #4943), pp. 312-317.

[SKHB08] C. Schuck, M. Kühnle, M. Hübner, and J. Becker (2008), "A Framework for Dynamic 2D Placement on FPGAs," Proc. 2008 Reconfigurable Architectures Workshop (RAW'08 at IPDPS'08).

[SMMT06] J. Septién, H. Mecha, D. Mozos and J. Tabero (2006), "2D Defragmentation Heuristics for Hardware Multitasking on Reconfigurable Devices," Proc. 2006 Reconfigurable Architectures Workshop (RAW'06 at IPDPS'06).

[SMMT08] J. Septien, D. Mozos, H. Mecha, J. Tabero, and M. A. García (2008), "Perimeter Quadrature-Based Metric for Estimating FPGA Fragmentation in 2D HW Multitasking," Proc. 2008 Reconfigurable Architectures Workshop (RAW'08 at IPDPS'08).

[SWP03] C. Steiger, H. Walder, and M. Platzner (2003), "Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices," Proc. Field-Programmable Logic and Applic. (Lect. Notes Comput. Sci. #2778), pp. 575-584.

[SWP04] C. Steiger, H. Walder, and M. Platzner (2004), "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks," IEEE Trans. Comput., vol. 53, no. 11, pp. 1393-1407.

[SWPT03] C. Steiger, H. Walder, M. Platzner, and L. Thiele (2003), "Online Scheduling and Placement of Real-Time Tasks to Partially Reconfigurable Devices," Proc. Real-Time Systems Symp.

[TFS01] J. Teich, S. P. Fekete, and J. Schepers (2001), "Optimization of Dynamic Hardware Reconfigurations," J. Supercomputing, vol. 19, no. 1, pp. 57-75.

[TMS03] J. Tabero, H. Mecha, S. Septién, S. Román, and D. Mozos (2003), "A Vertex-List Approach to 2D HW Multitasking Management in RTR FPGAs," Proc. DCIS 2003, pp. 545-550.

[TSMM04] J. Tabero, J. Septién, H. Mecha, and D. Mozos (2004), "A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management," Proc. 2004 Int'l. Conf. Field-Programmable Logic and Applications (Lect. Notes Comput. Sci. #3203), pp. 241-250.

[TSMM06] J. Tabero, J. Septién, H. Mecha, and D. Mozos (2006), "Task Placement Heuristics Based on 3D-Adjacency and Look-Ahead in Reconfigurable Systems," Proc. Asia and South Pacific Conf. Design Automation, pp. 396-401.

[TNY06] M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima, and K. Watanabe (2006), "A New Approach to Online FPGA Placement," Proc. 40th Conf. Information Sciences and Systems, pp. 145-150.

[WP02] H. Walder and M. Platzner (2002), "Non-Preemptive Multitasking on FPGAs: Task Placement and Footprint Transform," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA'02), pp. 17-23.

[WP03-1] H. Walder and M. Platzner (2003), "Online Scheduling for Block-Partitioned Reconfigurable Devices," Proc. Design Automation and Test in Europe (DATE 2003), pp. 290-295.

[WP03-2] H. Walder and M. Platzner (2003), "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," Proc. Int'l. Conf. Engineering of Reconfigurable Systems and Algorithms.

[WSP03] H. Walder, C. Steiger, and M. Platzner (2003), "Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing," Proc. Reconfigurable Architectures Workshop (RAW — at IPDPS'03).

[VFM05] J. C. Van der Veen, S. P. Fekete, M. Majer, A. Ahmadinia, C. Bobda, F. Hannig, and J. Teich (2005), "Defragmenting the Module Layout of a Partially Reconfigurable Device," Proc. Engineering of Reconfigurable Systems and Algorithms (ERSA 2005); http://arxiv.org/abs/cs/0505005.

[X04-1] Xilinx (2004), "Virtex Series Configuration Architecture User Guide," Application Note 151.

[X04-2] Xilinx (2004), "Virtex-II Platform FPGA: Complete Data Sheet."

[X05] Xilinx (2004), "Virtex-II Platform FPGA User Guide UG002(v2.0)."

[X07] Xilinx (2007), "Virtex-II Platform FPGAs: Complete Data Sheet DS301."

[X08] Xilinx (2008), "Virtex-4 FPGA User Guide." http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[X09] Xilinx (2009) "Virtex-5 FPGA User Guide." http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[ZWHP06] X.-G. Zhou, Y. Wang, X.-Z. Huang, and C.-L. Peng (2006), On-line Scheduling of Real-time Tasks for Reconfigurable Computing System, Proc. IEEE Int'l. Conf. Field Programmable Technology pp. 57-64.

[ZWHP07] X. Zhou, Y. Wang, X. Huang, and C. Peng (2007), "Fast On-Line Task Placement and Scheduling on Reconfigurable Devices," Proc. 17th Int'l. Conf. Field Programmable Logic and Applications (FPL'07), pp. 132-138.

VITA

Mostafa Elbidweihy was born in Cairo, Egypt, in September 1976. Mostafa received his primary and secondary education in various public schools in Egypt. He joined the faculty of Engineering, Cairo University, in 1994. He received his bachelor degree in electrical engineering on May 1999 with honors. He joined the masters program on Sep. 2001 and got his first master's degree in electrical engineering in May 2003. He joined Louisiana State University (LSU) in September 2003 to pursue his graduate studies, where he got a second master's degree in December 2007 and he completed the research presented in this dissertation with a 4.0 GPA on May 2009. He worked under the supervision of Dr. Jerry Trahan. He has been involved in researches in several areas, such as task scheduling on RC devices, free space management on RC devices, and scheduling on heterogeneous RC devices. He has several publications that have been published in prestigious journals in the field of reconfigurable computing and FPGAs. He has also participated in a number of international and national conferences. Mr. Elbidwiehy got married to (Sarah) in January 2004 and he has a two year old boy, Kadry.