2001

Dynamic Scheduling, Allocation, and Compaction Scheme for Real-Time Tasks on FPGAs

Shobharani Tatineni
Louisiana State University and Agricultural and Mechanical College, statin1@lsu.edu

DYNAMIC SCHEDULING, ALLOCATION, AND COMPACTION SCHEME FOR REAL-TIME TASKS ON FPGAS

A Thesis

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
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Shobharani Tatineni
B.E, Andhra University, Waltair, India, 1997
May 2002

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to Dr. Jerry L. Trahan for giving me his valuable guidance and insight during the course of this research work. I also acknowledge and appreciate the help rendered by Drs. Vaidyanathan Ramachandran, and Subhash C. Kak during the course of my graduate study. Thanks are also due to Dr. Susan Welsh of The Coastal Studies Institute for her constant support, encouragement and for also providing me with a graduate research assistantship.

I would also like to thank my parents and my sister for their moral and academic support through all my academic endeavors. Finally, I would also like to thank all the graduate students of The Department of Electrical and Computer Engineering for making my stay in Baton Rouge a pleasant one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Run-time reconfiguration (RTR) is a method of computing on reconfigurable logic, typically FPGAs, changing hardware configurations from phase to phase of a computation at run-time. Recent research has expanded from a focus on a single application at a time to encompass a view of the reconfigurable logic as a resource shared among multiple applications or users.

In real-time system design, task deadlines play an important role. Real-time multi-tasking systems not only need to support sharing of the resources in space, but also need to guarantee execution of the tasks. At the operating system level, sharing logic gates, wires, and I/O pins among multiple tasks needs to be managed. From the high level standpoint, access to the resources needs to be scheduled according to task deadlines.

This thesis describes a task allocator for scheduling, placing, and compacting tasks on a shared FPGA under real-time constraints. Our consideration of task deadlines is novel in the setting of handling multiple simultaneous tasks in RTR. Software simulations have been conducted to evaluate the performance of the proposed scheme. The results indicate significant improvement by decreasing the number of tasks rejected.

CHAPTER 1

INTRODUCTION

System performance is the most important factor for computer system designers. Until recent years, system designers had only two choices of implementing a computing system, which largely varied in terms of flexibility and performance. The first choice was to develop special purpose hardware tailored to meet the needs for a specific task. However, as far as dedicated optimized circuits are concerned, the system performance increases, but even minor changes to the system would require redesign. As a result, flexibility of changing any parameter of the system is lost. On the other hand, we have general-purpose systems that are built to accommodate the problem of flexibility, but system performance is compromised because of the lack of dedicated hardware resources designed for specific problems. The introduction of a new paradigm in hardware design called **Configurable Computing** (CC) offers to solve the problems addressed here, by retaining the general purpose nature, yet changing hardware configurations to offer the performance of dedicated circuits. Configurable computing enables mapping software into hardware with the ability to reconfigure its connections to reflect the software being run. The ability to completely reprogram the computer's hardware implies that this new architecture provides immense scope for emulating different computer architectures.

The key that has opened the door to configurable computing is the design of **Field Programmable Gate Arrays** (FPGAs), which are highly tuned hardware circuits that can be modified at almost any point during use. Recent years have seen faster reconfiguration times for FPGAs as well as partially reconfigurable FPGAs that can reconfigure part of their logic resources while the remainder continues to operate [BR96, H98]. This has permitted reconfiguration to play an active role in computations performed on FPGAs. In fact, an FPGA can reconfigure its logic and interconnections from one application to the next and even within the same application. This form of computation is known as **Run-Time Reconfiguration** (RTR) or **Dynamic Reconfiguration**. Typically, a system that can be changed in basic computational structure, either statically or dynamically at run-time, without adding physical devices is referred to as a configurable computing system. In its current realizations, a configurable computing system may consist of a set of general-purpose microprocessors augmented with a set of FPGAs. The time scale for changes in configurations can vary from every few months, to one application to the next, and even to run-time within a single computation.

RTR can increase system performance by using highly optimized circuits that are loaded and unloaded dynamically during the operation of the system. In this way, system flexibility is maintained and extra utilization of the ideal circuitry is achieved. Developing RTR is difficult, however, because of the need for both software and hardware expertise to determine how best to partition a computation into sections to implement in hardware, how to sequence these circuit sections, and how to tie them together to produce an efficient computation.

Researchers have developed RTR solutions to problems in numerous areas, including image and video processing, cryptography, and networking [HH95-A, JTY99, NMN99, SML98, VM97, WH95, WE99]. This work has confirmed the merits of RTR and established a range of specific applications. Looking forward, one can expect reconfiguration and reconfigurable logic to see much wider usage. To move towards common use of reconfigurable logic as a resource shared by multiple applications, however, we must step past the viewpoint of tailoring the entire FPGA resource to a single application, and develop methods to control the use of the reconfigurable resource in a sort of **"Hardware Operating System"** (HOS) [B96, DW99]. Also, for the general system setting, many challenges that one needs to overcome arise because of real time considerations.

The core requirement of real time systems is that tasks must complete by their deadlines. The tasks may be periodic and known in advance, so one could develop a schedule off-line. Instead, tasks may be irregular and not known in advance, so one must create a schedule on-line as tasks arrive and without knowledge of future tasks. Clearly, this is a difficult problem, as the problems of on-line scheduling of real-time tasks on a multiprocessor and dynamic processor allocation of tasks are NP-hard [DH91, LC89, MD78].

If dealing with general tasks not tied to a specific problem like cryptography, then we need an operating system-like interface for hardware. Such an interface becomes necessary in order to manage reconfiguration of the hardware at run-time and to fairly allocate FPGA resources among multiple processes. The operating system design implementation for a space-shared, time-shared multi-tasking FPGA could be realized to consist of five interdependent tasks: partitioning, placement, routing, scheduling, and swapping [DW99].

This thesis takes up the direction of using reconfigurable logic as a shared resource. We will present a method to schedule real-time tasks on a partially reconfigurable FPGA. A task allocator will receive tasks on-line, each with a given time duration, deadline, and (rectangular) area requirement. As each task arrives, it will allocate space and time on the FPGA to satisfy the time deadline of the task. This method takes a general stance on tasks independent of the specific applications and helps to move RTR toward more widespread utility.

A few scheduling algorithms for FPGAs that are application independent have been proposed in the past [DE97-A, DE97-B, DKW99, JTY99]. Most of these techniques deal with partial reconfiguration and multi-tasking for rearrangement of tasks on FPGAs. Nevertheless, they lack real-time capabilities. Our consideration of task deadlines is novel in the setting of handling multiple simultaneous tasks in RTR.

Our method for scheduling real-time tasks on a partially reconfigurable FPGA attempts a variety of strategies, attempting later strategies only when earlier ones fail. We order the strategies from those that are least disruptive on currently executing tasks to those that are most disruptive. The task allocator employs the following strategies in four phases.

1. Allocate directly — Either place the new task immediately or reserve a portion of the FPGA for a future time interval.
2. Reschedule reserved tasks — Reschedule tasks that have reservations for future execution but have not yet started, along with scheduling the new task.

3. Pre-empt an active task — Suspend operation of a currently executing task, place and run the new task, and reschedule the remaining time of the pre-empted task.
4. Compact tasks — Move the placement of active and reserved tasks to free up enough space for the new task.

The first three phases draw from an algorithm for scheduling tasks on a two-dimensional mesh of processors developed by Yoo and Youn [YY95]. A natural similarity exists between the contexts of mesh and FPGA; both place rectangular tasks with deadlines on a rectangular surface. Differences exist, however, in accounting for time to load FPGA tasks (both initially and in resuming a suspended task), time to store the configuration and state of a suspended task on an FPGA, and time to reload these to resume a suspended task. The processor setting assumes sufficient memory on the processor to hold the memory and state of a suspended task, while this is not available on an FPGA. Our approach accounts for these complicating factors. The fourth phase, compaction, draws from a task compaction algorithm for FPGAs developed by Diessel and ElGindy [DE97-A] that receives task requests on-line, as ours does, but without real-time constraints. These constraints compel design of new methods to handle both active and reserved tasks, and, of course, to account for deadlines relative to task running times and task relocation cost.

Summarizing, this thesis proposes and assesses techniques for rearranging a subset of real-time tasks on FPGAs to maximize utilization of the chip and minimize execution times. We consider moving the tasks from the chip, storing their state information, and reloading them to a new location on the chip. Comprehensive computer simulation reveals that the proposed scheme significantly improves system performance by reducing the number of tasks rejected.

We organize our description into the following chapters. Chapter 2 presents material on configurable computing, run-time reconfiguration, and hardware operating systems, and outlines the potential benefits and challenges of all these systems. Next, it motivates the theme after summarizing the HOS challenges facing the designers of space-shared and time-shared FPGA systems, followed by the problem definition and our proposed solution. Then, it reviews the current research on scheduling algorithms for FPGAs and mesh computers. Chapter 3 discusses the general FPGA architecture and cites various commercially available FPGA architectures. The chapter also presents the important features of FPGAs, such as RTR, and details how a run-time reconfigurable system can be realized with FPGAs. Chapter 4 is central to the rest of the thesis. It presents the hardware and operating system model upon which this thesis is based. It then introduces the notation and terminology that will be used throughout the thesis.

The next two chapters focus on the proposed algorithm and its computer simulation results. In Chapter 5, we present a detailed description of our algorithm with illustrations, followed by a time complexity analysis of the proposed scheme. Chapter 6 begins with a discussion of the simulation platform and reports the simulation results along with detailed interpretation of the results. Chapter 7 outlines a few of the many opportunities for further work in this area and concludes this thesis.

CHAPTER 2

BACKGROUND

2.1. Background

Traditionally, two options exist to implement a computing system, which can vary largely in terms of flexibility and performance. The first option is to use general-purpose microprocessors (such as Pentium® chips found in most personal computers). They are very flexible since a program (software) determines which operations are performed. Also, high level programming languages and sophisticated development environments make it relatively straightforward to develop applications. Microprocessors are relatively slow, however, since they execute most operations sequentially. The second option is the use of custom hardware circuits, or **Application Specific Integrated Circuits** (ASICs). ASICs are tuned to perform a specific task very fast. This is mainly achieved by performing operations in parallel and avoiding the program decoding overhead. ASICs are inflexible, however, since they cannot be programmed for other tasks. Examples of ASICs are custom graphics chips for personal computers that can paint pictures on the screen much faster than a general-purpose microprocessor can.

Recently, a new development in integrated circuits has offered a third option: large, fast FPGAs. As opposed to ASICs, the functionality of an FPGA is not determined in the factory. They are, rather, configured or programmed in the end product. To achieve this flexibility, FPGAs contain arrays of configurable logic blocks, programmable connections between the blocks, and input/output blocks for external communication (see Figure 2.1).

The logic blocks can be configured to perform any basic binary operations on its inputs, such as AND, OR, XOR, etc., and the results of these operations can be stored in registers and wired to the inputs of any other blocks. Arithmetic operators like adders, comparators, and counters are built by combining several logic blocks. Thus, arbitrary digital circuits can be implemented in FPGAs, and they can be reconfigured within milliseconds at any point during use. FPGAs blur the boundary between software and hardware; they approach the performance of ASICs while maintaining the flexibility of programmable processors.

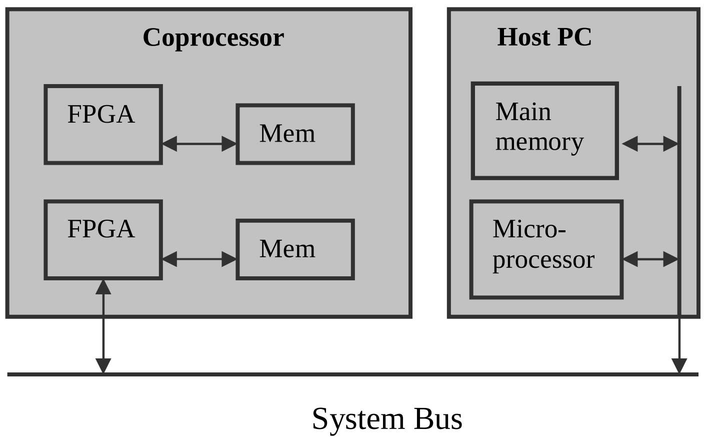

Figure 2.1: Typical architecture with FPGA as a coprocessor

2.1.1. General Configurable Computing

Configurable computing is a relatively new field of research, and it explores the use of configurable hardware, mainly in the form of FPGAs. Many promising application areas exist for this technology. FPGAs allow “field hardware upgrades” for devices with fast changing standards, like cable set-top boxes or mobile phones. One can also imagine novel appliances like a mobile phone that reconfigures itself into an MP3 audio player. Also, a standard microprocessor and FPGAs can be combined to form hybrid computers in which the FPGAs act as flexible coprocessors. Figure 2.1 shows a typical architecture with a coprocessor board containing FPGAs and local memory supporting them. A coprocessor can, for instance, perform hardware data encryption at one point in time, and then be reconfigured for image processing. In general, the coprocessor performs computation intensive, repetitive parts of an application that would be slower on the microprocessor [F95]. This approach is therefore called “hardware acceleration”. Even hybrid chips, which contain both a microprocessor and configurable hardware, are already commercially available.

Configurable computing can provide the following benefits:

Performance: Configurable computing can be highly optimized for a specific data set or specific applications [MH97].

Cost Effectiveness: Configurable computing can reduce system costs through hardware reuse [MH97]. System updates, including hardware configurations, can add new features to the system and, due to the field programmability, many problems with the design can be corrected without any changes to the on-board resources.

System Prototyping: Large systems (either ASICs or boards) can be built on a single or multiple FPGAs during development, making system testing and debugging easier than testing the real system and also cheaper.

Custom I/O: FPGAs provide a flexible set of programmable I/O signals. That gives the designer the opportunity to reuse existing hardware and to add new changes to it easily [MH97]. New FPGAs support different types of I/O levels and standards.

System Density: System density can be defined as the ratio of the utilization of the FPGA resources to the number of applications requesting the said resources. Configurable computing, especially run-time configurable logic, increases the system density and can deliver more functionality from the device many times more than its size by dynamically reconfiguring the system and increases resource utilization through loading and unloading different system modules [D00].

Fault-tolerance: System reliability can be increased through redundancy by duplication or building some testing circuits. Dynamic reconfiguration techniques can destroy old circuits and build new ones after detecting an error.

Many problems must be overcome, however, to make configurable computing a mainstream technology. The following are some of the issues that currently limit configurable computing systems.

FPGA gate capacity: FPGAs with over a million gates have become available recently. These devices are large enough to experiment with the basic strategies for configuration, yet are too small to implement many complex algorithms [BR96].

Configuration speed: Most of the existing FPGAs use serial configuration modes. This might not be an issue for some of the computational models that require slow design swapping, however, for run-time reconfiguration models, this could be a bottleneck [BR96].

Memory structures and interface: Lack of external memory interfaces in some existing FPGA devices forces designers to sacrifice some programmable resources to build an application specific memory interface.

Most importantly, "programming" an FPGA is not as simple as programming a microprocessor: it essentially means designing a digital circuit, that is, a structural description of operators and wires. Yet directly stating what operators need to be configured into the FPGA and how to connect them is a very tedious and error-prone process, and is specific to the particular FPGA device. Therefore, so-called synthesis tools have been developed to simplify FPGA circuit generation. They generate the circuit structures from higher level, device independent descriptions. Several researchers take this even further, aiming to synthesize hardware directly from a software programming language like C. This means that the user describes the desired functionality of a system, not its structure, and the tool automatically determines which hardware operators are needed. This opens configurable computing to software developers without hardware design experience. However, to date, there are not many reusable computing frameworks available that could provide higher levels of abstraction to the hardware and provide a systematic method of determining how well an algorithm could map into hardware.

2.1.2. Configurable Computing Systems

Designers have typically used FPGA systems in the past mainly in ASIC systems' prototyping because of their reconfigurability and the ease and low cost of their development process. They perform multiple operations on different phases of system operations. For example, the system can configure an FPGA to perform system diagnostics at power-up, then reconfigure it to do something else once the system is running. FPGA systems fall into three basic categories [MH97, LS96].

FPGA Emulators: These systems are built with no prior knowledge of the characteristics of the application that is going to use the system. As the name suggests, the purpose of these systems is to create a configurable system that resembles or emulates an FPGA that is much larger than those commercially available. The basic characteristic of an FPGA emulator is that of a large FPGA. In order to achieve this, some of the systems are augmented with embedded memories and hardware for accelerating specific tasks. These systems are primarily composed of FPGA chips connected together through Field Programmable Inter-Connect chips [MH97]. The main advantage of this model is that larger systems can be implemented, but it suffers from high cost and low clock performance due to the lack of system optimization and due to the use of programmable interconnections between FPGAs. FPGA-based emulation units are commercially available from manufacturers such as Quickturn Systems®, Zycad®, and Aptix®. A Quickturn emulation system was used to prototype the Intel Pentium® Chip [F95].

Custom Computing Machines (CCMs): The alternative to FPGA emulators is to design systems based on some prior knowledge of the applications that will be implemented on the platform. These systems are referred to as custom computers because the architectures are customized for a given class of application architectures which form a subset of FPGA applications [MH97]. These systems are built from an array of FPGAs that are connected directly to each other. These FPGAs are located on a single board, usually computer-based boards. These boards have built-in memory modules to store the FPGA configurations and the intermediate results. These machines are generally built for specific applications so the interconnections between FPGAs are optimized to get the best performance while reducing some of the system flexibility [MH97]. Some of the popular CCMs are the Splash® and DecPerle® machines. Some commercially available CCMs are available from many companies such as the Virtual Computer Corporation.

Run-time Reconfigurable Logic: A feature of run-time reconfigurable (RTR) systems is that they offer the best performance with maximum hardware utilization. This is because circuits are dynamically loaded into and unloaded from the hardware during operation of the system. The area used by such systems will be less than static systems due to unloading idle circuits. RTR systems can be built on a single FPGA chip or on multiple-FPGA systems like CCMs. The flow of operation of RTR systems is different from the traditional flow of general computing systems, as the circuit configurations are loaded into the system's memory instead of traditional instructions. These configurations are loaded and unloaded according to application needs.

FPGA-based boards for standard personal computers as well as for some workstations have been produced. For example, DEC's® Paris Research Lab designed and implemented four generations of FPGA-based reconfigurable coprocessors called Programmable Active Memories [F95]. The Prism-I System® from Brown University coupled XC3090 FPGAs with a Motorola® M68010 microprocessor and the Prism–II used XC4010 FPGA devices as co-processors for an AMD® 29050 RISC processor [F95]. These FPGA boards implement a flexible co-processing unit for standard computing architectures. An extensive listing of commercial and non-commercial FPGA based computing machines can be obtained from Guccione [G00].

2.1.3. Run-Time Reconfiguration

Configurable computing at run-time or on the fly is referred to as run-time reconfiguration (RTR) or dynamic reconfiguration or on-the-fly reconfiguration or in-circuit reconfiguration [DRDT]. There are two main implementation approaches for RTR applications: total or partial reconfiguration. In the first method, all FPGA resources are reconfigured or deleted between different configurations. In the partial reconfiguration approach, only the differences between the configurations are modified [HH95-A]. Some of the commercially available dynamically reconfigurable devices are the Atmel® AT40K FPGA family, DynaChip® DL6000 FPGA family, and Xilinx® Virtex FPGA family [DRDT]. Listings of some of the related academic and industrial research projects can be obtained from [DRDT].

RTR offers important benefits for the implementation of reconfigurable systems. They offer the fastest way possible to change an active FPGA circuit since only those parts that need to be reconfigured are interrupted. This results in faster overall system operation. The smaller configuration bit streams also require less external memory for storage. Also, the opportunities for deploying dynamic reconfiguration are increasing as the gate counts of individual FPGAs continue to improve. As larger FPGAs become available, the complexity of the system that can be integrated into a single FPGA increases. Consequently, the probability that the execution of subsets of the logic will be mutually exclusive is also likely to increase. The benefits of faster reconfiguration, and hence faster overall system speed, and reduced component count thus become more apparent with increasing part capacity.

2.2. Motivation

Although RTR applications provide many enhancements over static designs, RTR applications to date are still in the research area. Some of the major factors limiting the evolution of RTR applications are the following.

Hardware Limitations:

- FPGA gate capacity is relatively small compared to ASICs [MHA97]. This increases the complexity of the system because designers often have to use more than one FPGA and they have to partition their designs between the FPGAs. Interfacing problems arise when multiple FPGAs are used which increases the overall system delay.
- Configuration speed is the most important factor that limits the number of RTR applications. New FPGA architectures must be developed to reduce the configuration overhead.

Software Limitations:

- RTR applications need highly skilled developers who can deal with the low-level design, so these applications are developed in the laboratories under the supervision of qualified designers [MH97]. The only solution to this problem is to develop CAD tools that can reduce the design time.
- Benchmarking for applications is one of the important issues that must be addressed to enable designers to discuss the performance of their designs versus the performance of other design techniques [MH97].
- An efficient operating system-like interface is needed for hardware for managing the chip resources.

While significant hardware and software challenges remain, software challenges associated with FPGAs as reconfigurable processors may be more formidable than hardware issues. A major stumbling block for supporting multiple simultaneous applications or tasks and time-sharing among multiple applications on FPGAs is the need for an efficient **"Hardware Operating System"** (HOS).

An HOS is much like the general computing system **Operating System** (OS). As an operating system manages and schedules programs in multitasking environments, so does an HOS allocate hardware modules and page them in and out of the FPGA according to the schedule and flow of a program. These modules can be cached and relocated in the FPGA to get the optimal performance with minimum area. This is one of the advantages of partial reconfiguration.

The relocation problem is a current problem faced by operating system designers. They need to manage memory fragments and paging memory in and out of the physical memory to increase the size of the memory that can be used by running applications beyond the available real memory. In the HOS setting, the key difference is that hardware modules are two-dimensional which makes HOS more complicated and less efficient.

Some of the challenging problems that need to be solved for providing HOS are the following [FP98].

- **Partitioning**

  How should an FPGA be partitioned into functional logic blocks so that multiple independent configurations can be downloaded and executed?

- **Overlaying**

  If parallels can be drawn between different applications that will be implemented on the FPGA, then some of the FPGA resources could be dedicated for executing those common tasks, while the rest of the FPGA can be used for rarely used or mutually exclusive tasks. The question that arises is how can we best determine the configuration of an FPGA so that part of it is configured for computing common functions, while the remainder of the FPGA can be used to execute specific functions?

- **Application or Task Segmentation**

  How can an application or task be decomposed into smaller self-contained computing sub-tasks of variable size so that it could be downloaded and executed on an FPGA?

- **Application or Task Allocation and Scheduling**

  The operating system is responsible for allocating and scheduling a task. A task must therefore be re-locatable. How and where can a task be allocated on the FPGA so that it does not interfere with neighboring tasks and FPGA resources can be efficiently used? When should an application or a task begin execution?

- **Input and Output Multiplexing**

  The operating system assigns I/O resources to the task. How can inputs and outputs be efficiently assigned to the logical function associated to the executing task and how can they be increased when there are not enough physically available?

- **Dynamic Loading**

  Dynamic loading refers to loading the FPGA configuration as required by the executing or running application, either explicitly at run-time upon a system call or when the task is started or reactivated by the operating system. Since tasks need to be re-locatable, there is a need for dynamic loading. Task partitioning, placement, and routing need to adapt according to temporal and spatial constraints. How much pre-processing of the final design can be done? What needs to be done on the fly? How can it done efficiently?

- **Task Relocation**

  In order to provide an efficient and fault-tolerant operating system, it should also be able to support techniques like pre-emption and garbage collection. How can the operating system overheads be minimized? How should operating system activities be implemented? Should they be dynamically reconfigurable tasks? If so, how?

The above mentioned are only a subset of the problems that need to be solved in order to realize the potential of RTR applications in the real world. By providing effective solutions to these problems, we would be able to reap the benefits of FPGAs and in turn RTR.

2.3. Problem

The core requirement of real-time systems is that tasks must complete by their deadlines. Scheduling of random tasks is one of the difficult problems in real-time systems design, as they are unpredictable in terms of arrival time and service requests. If the tasks are periodic and known in advance, one could develop a schedule off-line. Instead, tasks may be irregular and not known in advance, so one must create a schedule on-line as tasks arrive and without knowledge of future tasks. Clearly, this is a difficult problem, as the problems of on-line scheduling of real-time tasks on a multiprocessor and dynamic processor allocation of tasks are NP-hard [DH91, MD78], and so is the problem of deciding whether it is possible to pack a set of rectangular tasks into a larger rectangle without overlap [LC89].

Several researchers have come up with different techniques for scheduling and allocating tasks on FPGAs [DE97-A, DE97-B, DEM00, EMS00, SO98]. Some of those techniques involved rearranging tasks on the FPGAs. All of those schemes, however, were for multi-tasking space-shared FPGAs. None of them looked at real-time restrictions on tasks.

This thesis presents an investigation of a new algorithm that incorporates techniques for attempting to utilize FPGA resources to the maximum extent, accommodating real-time issues of tasks. We will take up the direction of using reconfigurable logic as a shared resource and present a method to schedule real-time tasks on a partially reconfigurable FPGA. A task allocator will receive tasks on-line, each with a given time duration, deadline, and (rectangular) area requirement. As each task arrives, the allocator will allocate space and time on the FPGA to satisfy the time deadline of the task, using scheduling, pre-emption, and compaction as its tools. This method takes a general stance on tasks independent of the specific applications and helps to move RTR toward more widespread utility.

2.4. Related Work

Some researchers have begun looking at related aspects of this problem of controlling an FPGA as a shared computing resource [DW99, FP98]. Diessel et al. [DKW99] designed a multitasking operating system to share a board with eight FPGAs among up to eight users. Each user (each task) receives one entire FPGA, that is, each FPGA handles only one task at a time. Jean et al. [JTY99] designed a resource manager to allocate and load configurations on a system with eight FPGA chips. Each task may request multiple chips, but without partial reconfiguration of a chip. Wirthlin and Hutchings [WH95] time-shared a partially reconfigurable FPGA among different tasks in the DISC system. In this case, each task spanned the width of the FPGA, so placement was simplified. Spillane and Owen [SO98] introduced partitioning and scheduling of a circuit too large for a single FPGA onto a partially reconfigurable FPGA. They discussed scheduling variations and time analyses. Burns et al. [BDH97] incorporated operating system style services into RAGE, their run-time system for managing dynamic reconfiguration of FPGAs. A virtual hardware manager handles incoming tasks. If it determines that the task can be placed, though not in the default position for the task, then it passes the task to the transformation manager to translate and rotate the task into position. It does not allow pre-emption. Shirazi et al. [SLC98] developed a manager for RTR that sequences tasks and determines their placement and can accommodate partial or full reconfiguration. It does not detail placement considerations or deal with real-time constraints, though it can handle tasks with unknown service times. Robinson and Lysaght [RL99] extended the work of Shirazi et al. [SLC98] with a generic model of a configuration controller. Their controller schedules tasks according to priority on a partially reconfigurable FPGA and allows pre-emption of active tasks. They did not consider real-time constraints or task compaction.

On task compaction on FPGAs, Diessel et al. [DEM00] also investigated more flexible compaction schemes (hence, demanding more computation time) and heuristics for scheduling the task movements. Diessel and ElGindy [DE97-B] extended the work of Diessel et al. [DEM00] by proposing another technique for task compaction called "local repacking" which allows the tasks to be rotated. Their study consists of two phases: identifying a feasible rearrangement and scheduling the task movements. They examined local repacking of tasks within a rectangular region of the array and compared to earlier work on ordered compaction [DEM00]. For identifying a re-arrangement, they used a quadtree to store information on free cells and a 2D bin packing algorithm. They dealt with scheduling. ElGindy et al. [WMS00] applied a genetic algorithm to handle task rearrangement and order of task movements in the setting of an application with input streams at constant data rates. On task scheduling, several researchers have studied different scheduling techniques on multiprocessor systems [C99, M97, MPR93, SP96, Y97, YCY99]. Some of the multiprocessor systems are meshes and hypercubes. We will establish later in our thesis that scheduling on FPGAs is similar to scheduling tasks on multiprocessor systems albeit with notable variations. The multiprocessor system models that those researchers used and the FPGA model that we will be using consider the available resources as a rectangular area and assume that each task requests a rectangular region. Miller et al. [MPR93] proposed some new parallel algorithms for solving a wide variety of problems including image processing on reconfigurable meshes. They introduced implementations of some fundamental data movement operations such as read/write, data reduction, and parallel prefix, which lay the foundations for their algorithms. Mohapatra [M97] investigated a real-time scheduling scheme called "deferred earliest deadline first" on hypercubes. The main idea of his scheme is to defer scheduling as long as possible and batch schedule requests ensuring that the tasks meet their deadlines. Sharma and Pradhan [SP96] examined an allocation strategy in mesh computers that tries to find free space for a task by searching along the corners of an allocated space and also along the corners of the mesh system. They attempted to prove that keeping allocated spaces together reduces fragmentation of the system. Chiu [C99] proposed a scheduling algorithm for 2D meshes that considers finding placement for tasks by searching only those spaces that border from the left of the allocated spaces or those that have their left boundaries aligned with that of the mesh. Yoo [Y97] designed a task allocator for 2D meshes that determines the availability of free space through manipulation of allocation status on each row instead of scanning the entire 2D mesh. Yoo et al. [YC96] examined full relocation and partial relocation of tasks in 2D meshes. The full relocation moves all tasks left, then moves all tasks down alternately until enough space is freed up for the incoming task. Partial relocation identifies possible bases for an incoming task. For each possible base, the number of occupied cells that need to be moved if the task were to be placed there is determined. The minimum is selected and only necessary tasks are moved.

CHAPTER 3

ARCHITECTURE

3.1. FPGA Architecture

The Field Programmable Gate Array is a type of programmable logic that can be configured for implementing a wide variety of applications. The key distinguishing property of programmable logic is reconfigurability. Such devices cannot compete with custom ASIC hardware implementations in terms of density or speed, but their reconfigurability allows hardware designs to be created and changed rapidly, thus reducing time-to-market and costs over custom hardware.

Commercially available FPGAs can be categorized into four main types: symmetrical array, row-based, hierarchical PLD, and sea-of-gates. They all differ in the interconnections and the type of programming technology used. Figure 3.1 shows different FPGA architectures, of which symmetrical architecture is assumed in this thesis, and will be discussed later in this chapter.

Currently, four technologies are in use. They are static RAM cells, anti-fuse, EPROM transistors, and EEPROM transistors. Depending upon the application, one FPGA technology may have more desirable features for that application.

Static RAM Technology: In the static RAM FPGA, programmable connections are made using pass transistors, transmission gates, or multiplexers that are controlled by SRAM cells. The advantage of this technology is that it allows fast in-circuit reconfiguration. The major disadvantage is the size of the chip required by the RAM technology.

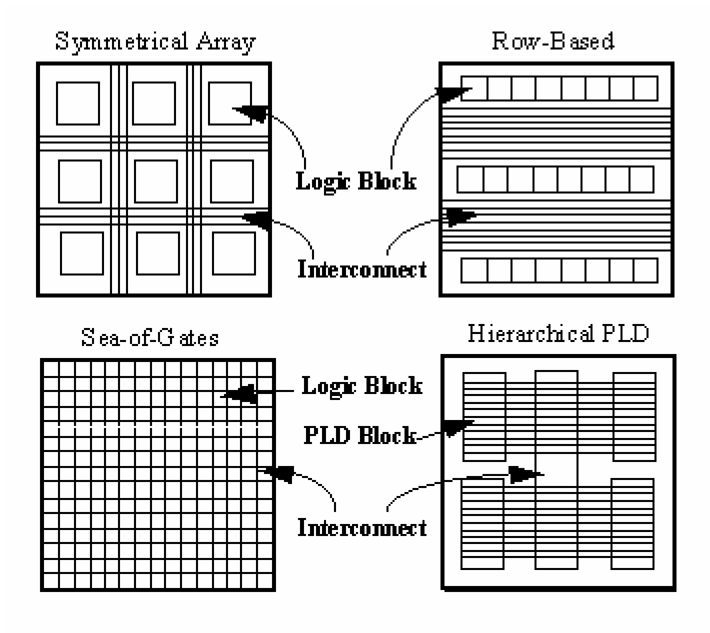

Figure 3.1: Classes of FPGAs [VCC]

Anti-Fuse Technology: An anti-fuse resides in a high-impedance state and can be programmed into a low-impedance or “fused” state. Less expensive than the SRAM technology, this device is a program-once device.

EPROM/EEPROM Technology: This method is the same as used in the EPROM memories. One advantage of this technology is that it can be reprogrammed without external storage of configuration, though the EPROM transistors cannot be re-programmed in-circuit.

Table 3.1 shows some of the characteristics of the above programming technologies and Table 3.2 shows some of the commercially available FPGAs [RGS93].

Table 3.1: Characteristics of FPGA technology [MHA97]

Technology	**Volatile ?**	**Reprogrammable?**	**Chip Area**
Static RAM	Yes	In-circuit	Large
Plice Anti-Fuse	No	No	Fuse (small) Prog. Trans (Large)
ViaLink Anti-Fuse	No	No	Fuse (small) Prog. Trans (Large)
EPROM	No	Out of Circuit	Small
EEPROM	No	Out of Circuit	2x EPROM

Table 3.2: Selected commercial FPGAs of different types.

Company	**Architecture**	**Logic Block Type**	**Programming Technology**
Actel	Row-based	Multiplexer-Based	Anti-fuse
Altera	Hierarchial-PLD	PLD Block	EPROM
QuickLogic	Symmetrical Array	Multiplexer-Based	Anti-fuse
Xilinx	Symmetrical Array	Multiplexer-Based	SRAM

An FPGA consists of three main types of configurable elements: configurable logic blocks (CLBs), input/output blocks (IOBs), and interconnect resources [F95]. As we mentioned earlier, several FPGA architectures are available today, varying in the type of programming technology used, size, structure, and number of logic and I/O blocks, and the amount and connectivity of the routing resources. Figure 3.2 shows the general structure of an FPGA. In this thesis, we assume SRAM-based FPGAs as they allow in-circuit reconfiguration. With SRAM-based FPGA technology, customized configuration is established by programming internal static memory cells that determine logic functions and internal connections implemented in an FPGA.

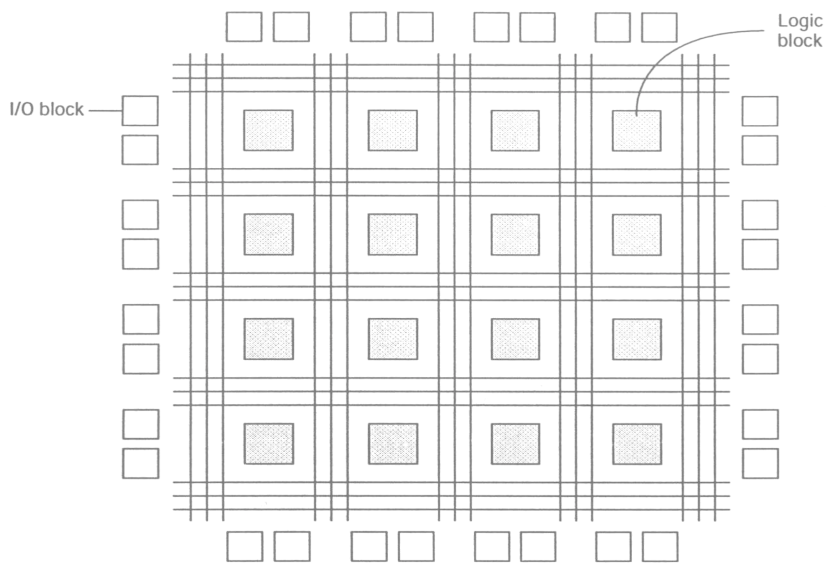

Figure 3.2: Generic FPGA structure.

The CLBs provide the functional elements for constructing a user's logic [MHA97]. CLBs typically contain resources for implementing combinational logic functions and data storage. They implement most of the logic in an FPGA. The flexibility and symmetry of the CLB architecture facilitates the placement and routing of a given application. As an example, a Xilinx XC4000 CLB (Figure 3.3) contains a pair of flip-flops and two independent 4-input function generators. These function generators have a good deal of flexibility as most combinational logic functions need less than four inputs.

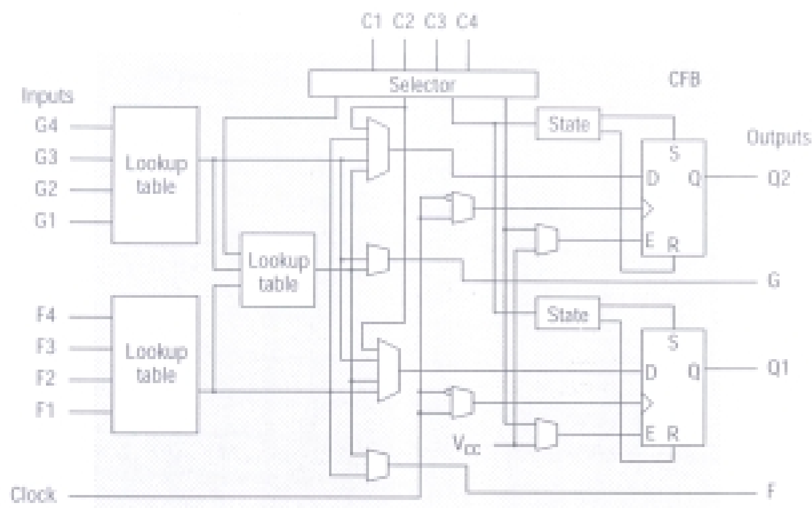

Figure 3.3: Structure of Xilinx XC4000 CLB [BR96]

The IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs into the appropriate networks. As an example, Figure 3.4 shows the routing architecture for an XC4000. In general, all interconnections are composed of metal segments with programmable switching points to implement the desired routing. An abundance of different routing resources is provided to achieve efficient automated routing. There are four main types of interconnect, three are distinguished by the relative length of their segments: single-length lines, double-length lines, and long-lines [BR96]. The fourth type of interconnects are programmable switches for connecting CLB inputs and outputs to the wire segments or to connect one wire segment to the other.

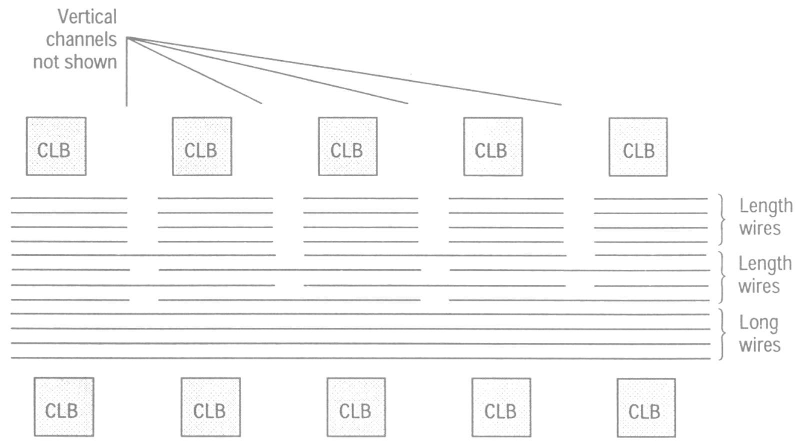

Figure 3.4: Xilinx XC4000 Wire Segments [BR96]

As discussed earlier in this chapter, a variety of FPGA architectures are commercially available. For example, the Xilinx 3000 family uses a symmetrical architecture. We overview the symmetrical architecture here. A symmetrical FPGA consists of an array of logic blocks. Figure 3.5 shows the symmetrical architecture in detail. The wiring channel is made up of wiring segments in a variety of fixed lengths. Logic blocks and I/O block pins connect to wire segments in wiring channels, and wiring channels intersect at switch boxes.

3.2. Reconfigurable System

A reconfigurable system is one that exhibits the property of reconfiguration. For example, reconfigurable devices such as FPGAs can be reconfigured to change logic functions while resident in the system. These are devices whose internal architecture as well as interconnections can be reconfigured to match the needs of a given application. Greater and more effective performance can be achieved by time sharing the FPGA logic array to implement a number of different circuits in sequence. Hence,

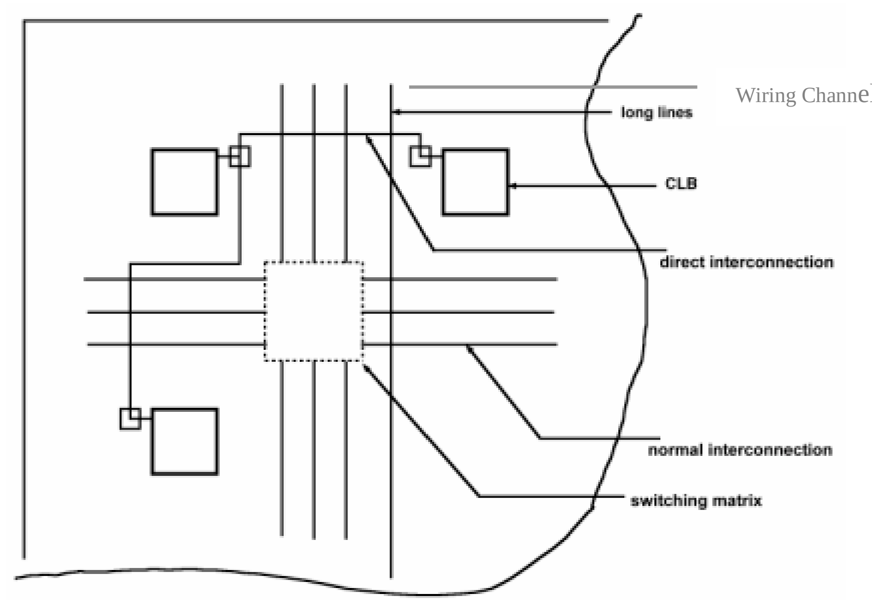

Figure 3.5: Symmetrical FPGA architecture

an FPGA's function is not fixed but rather changes at run time to implement different functions as needed. Reconfigurable systems enable us to implement designs using fewer FPGAs.

3.3. Dynamic Reconfiguration

Dynamic reconfiguration, also called run-time reconfiguration, in-circuit reconfiguration, or on-the-fly reconfiguration, is a type of reconfiguration that allows modifications of a system configuration during its normal operation. There is no need to reset the remaining circuitry or to remove reconfigurable elements for programming. In principle, any RAM or FLASH memory-based FPGAs can be dynamically reconfigured, that is, its configuration can be changed while the rest of the circuit is fully operational.

3.4. Partial Reconfiguration

Partial reconfiguration is the selective updating of a subarray of an FPGA's programmable logic and routing resources while the remainder of the device's programmable resources continue to function without interruption. The speed of dynamic reconfiguration is directly proportional to the number of configuration memory locations that need to be changed in order to implement a dynamic design. Some FPGAs possess partial reconfiguration ability. Thus, such an FPGA does not have to be halted in order to have its function partially reconfigured. The main advantage of partial reconfiguration is that it offers the fastest way to change an active FPGA circuit, since only those parts that need to be reconfigured are interrupted.

Typically, a reconfigurable system is implemented in the form of a single or multiple FPGAs used as co-processor(s), built on the same chip as the processor, or an integration of reconfigurable logic and memory. Some of the commercially available partially reconfigurable devices are AT6000® family, XC6000® family, AT40K® family, DynaChip® DL6000, Xilinx® Virtex family [AT6000, DRDT, XC6000]. Some of these devices are currently withdrawn from production.

The Xilinx 6200 family comprises high performance FPGAs from Xilinx that support partial reconfiguration. These devices are designed to operate as a co-processor attached to either a microprocessor or a microcontroller [XILINX]. Table 3.3 gives the product description of XC6000 FPGAs such as gate counts, etc.

Table 3.3: The XC6200 FPGAs

Device	**XC6209**	**XC6216**	**XC6236**	**XC6264**
Typical Gate Count Range	9000-13000	16000-24000	36000-55000	64000-100000
Number of Cells	2304	4096	9216	16384
Number of Registers	192	256	384	512
Cell Rows $\times$ Columns	48×48	64×64	96×96	128×128

CHAPTER 4

FRAMEWORK FOR TASK SCHEDULING ON AN FPGA

This chapter presents the design environment for the algorithm. The first section describes the high level FPGA model and assumptions made about tasks and the system. Section 4.2 discusses notation and terminology.

4.1. System Model

Compaction of tasks on an FPGA involves movement of tasks from one place to the other on an FPGA. Inputs to the tasks that need to be moved have to be suspended and then the state of the logic blocks is stored by waiting until the results of the last input appear, or by waiting until a task reaches a checkpoint. Once the task is suspended, stored states have to be loaded back onto the portion of the FPGA at the task's destination and input to the task needs to be resumed for execution to continue. In this thesis, we have not addressed the problem of rerouting I/O to a task that is moved, assuming that sufficient I/O resources exist for this purpose.

Pre-emption involves pre-empting a task and later restoring the pre-empted task. In order to pre-empt a task, we need to store the state of the task and re-load the state in a way similar to that described above for compaction, except that the task is not moved from where it was originally allocated on the FPGA. Hence, rerouting I/O resources is not an issue here since the same I/O resources can be used.

We model an FPGA as a rectangular $r \times c$ array of reconfigurable logic blocks, or cells. Index cell (i, j) by its row and column, for $1 \leq i \leq r$ and $1 \leq j \leq c$. Cell (1, 1) is in the lower left corner and cell (r, c) is in the upper right corner. Interconnection resources comprise rows of links between rows of cells and columns of links between columns of cells; the FPGA has switching blocks where these rows and columns intersect. We make the following assumptions about the FPGA (consistent with those of Diessel and ElGindy [DE97-A]). The FPGA is partially reconfigurable with reconfiguration time proportional to the number of cells being reconfigured since cells are configured sequentially. **Configuration delay** is the time to configure a cell and its associated routing resources. A task is also modeled as a rectangular grid of specified width and length. The service time of a task is assumed to be the total processing time, which includes initial configuration delays and loading times, but excludes any time taken for suspension or migration. A task will specify the size of the h x w subarray needed for its execution, where $h \leq r$ and $w \leq c$. It can be placed on any h x w subarray of the FPGA. We assume adequate I/O resources to support any placement. A task will also specify its service time and deadline. Before pre-empting or compacting a task, we wait for the results of the last input or wait to reach a checkpoint before suspending it. This waiting time is negligible compared to the time to configure a task. Any task may be suspended with its inputs buffered and its configuration and state stored for future reloading on the FPGA. Note that resuming a task after pre-emption or moving the task incurs again the time to configure its subarray. I/O re-routing has not been addressed for a task that is moved.

A sequential controller queues tasks as they arrive. They reside in the system external to the FPGA. A task allocator, executing on the controller, attempts to schedule and place the next pending task on the FPGA using the algorithm described in Chapter 5. Call tasks that are currently executing as **active tasks** and tasks scheduled to start at some time in the future as **reserved tasks**.

4.2. Notations and Terminology

We will make use of the following notations and terminology.

Task: A task T is a seven-tuple T = (a, d, s, h, w, t_1, t_2), where

a = arrival time,

d = deadline,

s = service time (worst case),

h = height of the subarray that *T* requires,

w = width of the subarray that *T* requires,

t_1 = start time of the subarray allocated to the task, and

t_2 = finish time of the subarray allocated to the task.

The seven-tuple except t_1 and t_2 are known to the system when T arrives. The task starts at the beginning of time unit t_1 and finishes at the end of time unit t_2, that is, it runs for the interval [t_1, t_2]. The task satisfies its deadline if $t_2 \le d$. Let a(T), d(T), etc. denote the arrival time of task T, the deadline of task T, etc.

Real-time subarray: A real-time subarray S is a six-tuple (x, y, x′, y′, t_1, t_2), where (x, y) is the lower left cell of S. We refer to this cell as the base of S. Cell (x′, y′) is the upper right corner of S. Time t_1 denotes the time at which a task is scheduled to start on S and t_2 denotes the time when the task is scheduled to complete on S. If a subarray is neither allocated nor reserved, then its t_1 and t_2 serve as flags. S is of size $h \times w$, where $h = x' - x + 1$ and $w = y' - y + 1$.

Current time: Current time, t, is the time at a point of consideration.

Latest start time: Latest start time (LST) is the time by which a task must start execution in order to meet its deadline: LST = $d - s + 1$.

Laxity: The laxity of a task is the number of time units in the window from t to d in which the task can be inactive and still satisfy its deadline. The laxity of a reserved or unscheduled task is LST - t, or $d - s + 1 - t$ for (remaining) service time s. The laxity of an active task is $d - t_2$.

Active subarray: An active subarray is a real-time subarray allocated to a task that is currently executing. For current time t, this is a real-time subarray S for which $t_1 \leq t \leq t_2$.

Active subarray set: The active subarray set is a set of all active subarrays.

Reserved subarray: Reserved subarray is a real-time subarray allocated to a task for future execution, that is $t_1 > t$, where t is the current time.

Reserved subarray set: The reserved subarray set is the set of all reserved subarrays.

Free subarray: A free subarray is a subarray that is currently neither active nor reserved.

Non-allocable subarray: A non-allocable subarray for an incoming task T, due to a task T_k, is the subarray consisting of cells that cannot serve as the base of T due to conflict with T_k. For a real-time subarray (x, y, x', y', t_1, t_2) of size $h \times w$ in an array of size $r \times c$, the non-allocable subarray is $(x'', y'', x', y', t_1, t_2)$, where $x'' = \max\{1, x - h + 1\}$ and $y'' = \max\{1, y - w + 1\}$. Let $\xi_A(T, T_k)$ denote this subarray when T_k is an active task, and let $\xi_R(T, T_k)$ denote this subarray when T_k is a reserved task.

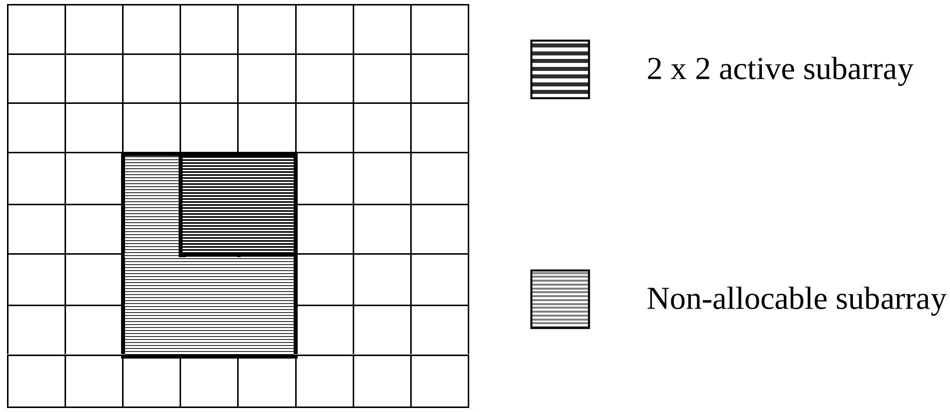

Figure 4.1: Non-allocable subarray for an incoming task of size 3×2 due to an active task of size 2×2.

See Figure 4.1 for an example, where (4, 4, 5, 5, t_1, t_2) is a 2×2 active subarray and (2, 3, 5, 5, t_1, t_2) is the non-allocable subarray for an incoming 3×2 task.

Non-allocable subarray set: The non-allocable subarray set for an incoming task T is the set of all non-allocable subarrays for T due to all active and reserved tasks.

Non-allocable border subarray: A non-allocable border subarray for an incoming task T of size h x w is the subarray consisting of cells that run along the top and right borders that cannot serve as the base of any free subarray. For every task, two non-allocable subarrays exist: the top h-1 rows in the horizontal direction and the rightmost w-1 columns in the vertical direction. Figure 4.2 shows the horizontal and vertical non-allocable border arrays for an incoming 2 x 3 task.

Top-cell & right-cell intervals: For active task T_k and incoming task T, the top cell interval extends in the row immediately above T_k from the right edge of T_k to the left from a distance one less than the sum of the widths of T_k and T (or until encountering the left edge of the FPGA). The right cell interval extends in the column immediately

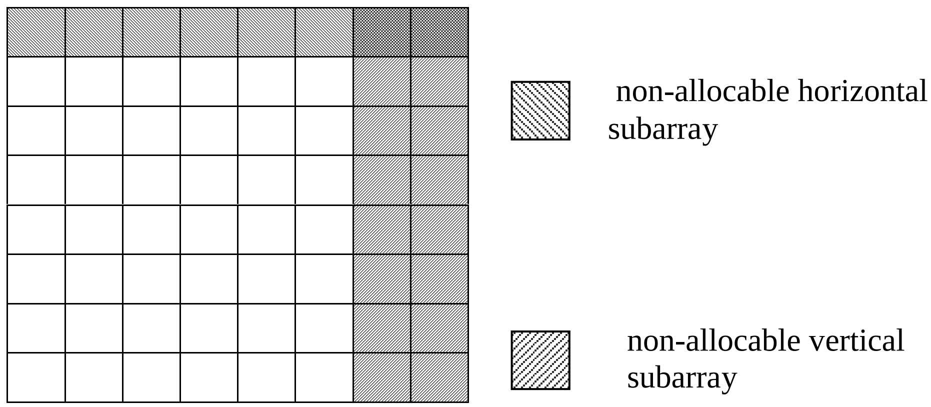

Figure 4.2: Non-allocable border subarrays for an incoming task of 2 x 3.

to the right of T_k from the top edge of T_k down for a distance one less than the sum of the heights of T_k and T (or until encountering the bottom edge of the FPGA). Figure 4.3 shows the top-cell and right-cell intervals for an incoming 3×4 task.

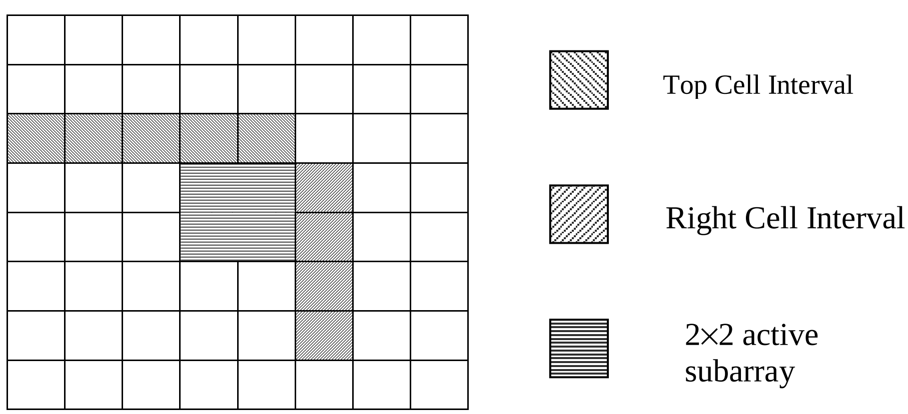

Figure 4.3: Top-cell and Right-cell intervals for an incoming task of 3×4.

CHAPTER 5

SCHEDULING ALGORITHM

This chapter details background ideas, the scheduling algorithm, and complexity analysis of the algorithm. The first two sections sketch the real-time task allocation scheme for meshes of Yoo and Youn [YY95] and the FPGA task compaction algorithm of Diessel and ElGindy [DE97-A] respectively. The third section describes our scheduling, allocation, and compaction method, while the last section presents complexity analysis of the algorithm.

This thesis investigates the techniques of reserving, pre-empting, and compacting tasks at run time in order to minimize waiting delays for the next pending task, maximize the utilization of the FPGA, and minimize the number of tasks rejected due to unavailability of subarrays that can accommodate the task and guarantee to finish execution within their deadlines. A brief description of the techniques follows. Reserving a task involves allocating a subarray to the task for future use, guaranteeing to finish it within its deadline. Pre-empting involves temporarily suspending a task from executing and allocating its subarray to the incoming task until it finishes its execution, then loading back the pre-empted task to the same subarray, where the pre-empted task and the incoming task can all finish execution within their deadlines. Compaction involves rearranging a subset of tasks that are currently executing and currently reserved on the FPGA in order to accumulate sufficient contiguous space for the incoming task.

We begin by overviewing the task compaction method of Diessel and ElGindy [DE97-A] for non-real-time tasks on FPGAs, then we describe the real-time task scheduling algorithm of Yoo and Youn [YY95]. Our algorithm, which integrates ideas from these two sources, follows.

5.1. Task Compaction Technique

Diessel and ElGindy [DE97-A] investigated the use of compaction on an FPGA. They described and assessed a one-way, one-dimensional, order preserving, task compaction heuristic by exploiting partial reconfiguration on an FPGA. It is one-way and one-dimensional in that it slides tasks from left to right along the rows of the FPGA cells. It is order preserving in that it does not change the relative order of tasks. Figure 5.1 contains an example of such a compaction.

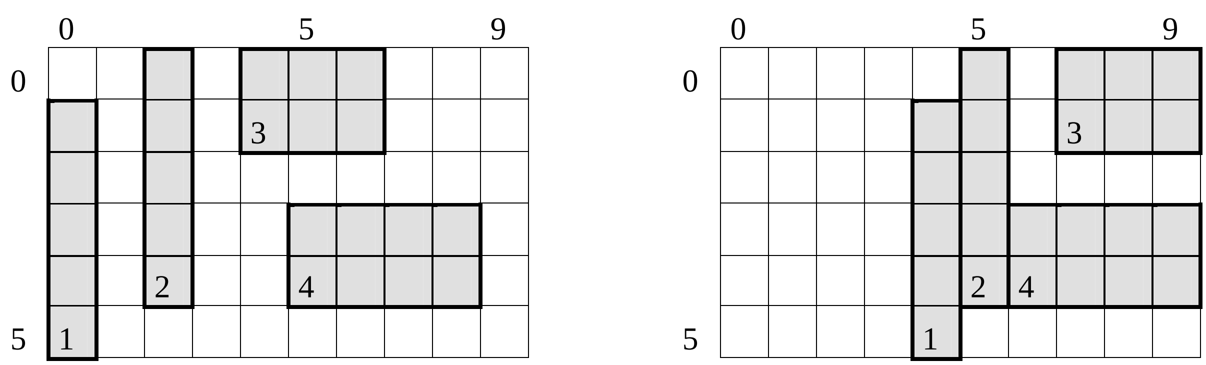

Figure 5.1: Placement of tasks on an FPGA before (left) and after (right) compaction.

They considered the same FPGA task model that we have adopted, except that they did not consider real-time tasks with deadlines. Since they ignored deadlines, their main concentration was to find an available subarray for an incoming task by manipulating the active tasks on an FPGA.

The compaction technique of Diessel and ElGindy works as follows. As the tasks arrive, a task allocator enqueues them. The task allocator attempts to find a subarray for the next pending task in the queue according to the Bottom Left Allocation method, which assigns an incoming task to the bottom leftmost free subarray of sufficient size, if possible. If a subarray cannot be found, then the task allocator checks the possibility of compacting the currently active tasks as follows.

To avoid searching all cells as possible base locations after a compaction, the task allocator examines cells in the intersection of top cell intervals and right cell intervals. This heuristic aims to minimize the cost of moving tasks in compaction [DE97-A].

To identify candidate cells to serve as the base of the new task, Diessel and ElGindy used a directed graph called the visibility graph having active tasks as its vertices. A task S dominates a task T if they occupy a common row and S is to the left of T. S directly dominates T if no task R exists such that S dominates R and R dominates T. The visibility graph has an edge from vertex S to vertex T if S directly dominates T. Assign to the edge from S to T the distance from task S to dominated task T. Sum the edge distances in a bottom-up fashion to compute the maximum distance a task can move to the right in a compaction. For every potential base b from the set B of candidate bases (intersection of top cell intervals and right cell intervals), search those subgraphs whose covered rows intersect the allocation site based at b. Choose the allocation site with minimum cost of freeing its site of active tasks and then compact the necessary tasks to the right.

Phase 4 of the algorithm in Section 5.3 adapts the algorithm of Diessel and ElGindy to a real-time setting, for example, handling the impact on reserved tasks of compacting active tasks.

5.2. Task Reservation and Pre-emption Techniques

Yoo and Youn [YY95] investigated the use of reservation and pre-emption techniques on multiprocessor systems. In their paper, they proposed an on-line scheduling and processor allocation scheme for real-time tasks on a two-dimensional mesh of processors. To minimize task rejection, their scheduling algorithm for real-time tasks works in three phases as follows. In Phase 1, when a task arrives, the task allocator tries to find a free submesh available to accommodate the incoming task for either immediate or future use, under the condition that this guarantees to finish the task within its deadline. If no suitable free submesh is available, then in Phase 2 the task allocator checks if the task can be reserved by rescheduling previously reserved tasks. If it can find a submesh for reserving the incoming task, where it can reschedule all previously reserved tasks so that they can finish within their deadlines, then it reserves this submesh for future use. If it cannot find such a submesh, then in Phase 3 it checks the possibility of pre-empting an active task, where all the affected tasks due to pre-emption are guaranteed to finish within their deadlines. If possible, then the task allocator pre-empts the task and allocates its submesh to the incoming task. Otherwise, the task allocator rejects the task.

Given that this was to run on a mesh of processors, the algorithm assumes sufficient local memory to store the context and memory information for a task that is pre-empted, so resuming a pre-empted task is straightforward and fast. In an FPGA, however, the cells do not have the memory to store one task's state and configurations while another is executing, so resuming a pre-empted task requires reloading configuration and state. The allocator must account for this time with respect to the task deadline when making pre-emption decisions for the FPGA, which does not arise for the mesh of processors. In fact, the FPGA setting demands consideration of task loading time in all scheduling decisions. Phases 1-3 of the algorithm in this paper (Section 5.3) adapt Yoo and Youn's algorithm to an FPGA setting, for example, accounting for task reloading time.

5.3. Algorithm

The task allocator seeks to allocate tasks as they arrive. The goal is to allocate incoming tasks such that they can execute within their deadlines. For this we employ a four-phase approach. When a task T arrives, Phase 1 executes. In this phase, the task allocator tries to schedule the incoming task T given a collection of active and previously reserved tasks. (If any phase fails and the laxity of T is greater than the time to execute the next phase, then execute the next phase; otherwise, reject T.) Phase 2 checks the possibility of rescheduling reserved tasks to schedule T as well as the previously reserved tasks to complete execution within their deadlines. Phase 3 tests the possibility of pre-empting an active task A, so that (i) T receives the subarray for A and (ii) T, A, and reserved tasks overlapping the subarray for A can be rescheduled to complete execution within their deadlines. Phase 4 checks the possibility of finding a subarray to allocate to T by compacting the active and reserved tasks to the right.

The allocator maintains two priority queues, AQ and RQ, holding information on the set of active tasks and set of reserved tasks, respectively. (Though priority queues are asymptotically more efficient, for the sizes studied in our simulations, however, lists were simpler and as fast.) Organize these queues according to increasing laxity. In rescheduling tasks and pre-empting tasks, the algorithm follows a heuristic guideline of minimum laxity first. The task allocator inserts a task into AQ at allocation time and deletes a task at release time. The task allocator inserts a task into RQ at reservation time and deletes a task at allocation time.

The algorithm describes the handling of a new task T of size $h \times w$ to be placed on an FPGA of size $r \times c$ with given AQ and RQ. Let m (n) denote the number of active (reserved) tasks at the current time. We will include a running example after each phase to display the actions of the phase.

5.3.1. Phase 1 — Allocate Directly

In Phase 1, the task allocator simply attempts to add T to the existing schedule, either starting T at the current time or reserving a subarray for a future time interval. To determine where and when to place T, the task allocator determines for each cell the earliest time at which the cell becomes free and remains free. Call this as the earliest available time for the cell and, for cell (i, j), denote it as E(i, j). Formally, $E(i, j) = \max\{t_2(i, j)+1, t\}$, where $t_2(i, j)$ is the finish time of the last task scheduled on cell (i, j) and t is the current time.

Procedure to generate E:

1. Initialize each element E(i, j) to t, where $1 \le i \le r-h+1$, $1 \le j \le c-w+1$, and t is the current time.

2. For each active task T_k,

   (a) determine non-allocable subarray $\xi_A(T, T_k)$ due to active task T_k, and,

   (b) for each cell $(i, j) \in \xi_A(T, T_k)$ and such that $1 \le i \le r-h+1$ and $1 \le j \le c-w+1$, if $E(i, j) \le t_2(T_k)$ (finish time of T_k), then $E(i, j) \leftarrow t_2(T_k) + 1$.

3. For each reserved task R_k,

   (a) determine non-allocable subarray $\xi_R(T, R_k)$ due to reserved subarray R_k, and

   (b) for each cell $(i, j) \in \xi_R(T, R_k)$ and such that $1 \le i \le r-h+1$ and $1 \le j \le c-w+1$, if $E(i, j) \le t_2(R_k)$ (finish time of R_k), then $E(i, j) \leftarrow t_2(R_k) + 1$.

Observe that cells in rows above $r-h+1$ and columns to the right of $c-w+1$ cannot serve as the base for T as T would not fit on the FPGA if based there. For each new task, the task allocator will recreate E since E depends on the current time and given set of active and reserved tasks.

Procedure for Phase 1:

1. Update RQ by deleting those tasks whose start time is less than or equal to t and adding them to AQ if their finish time is greater than or equal to t.
2. Update AQ by deleting those tasks whose finish time is less than t.
3. Construct E for the task T.

4. Search E from left to right, bottom to top until finding an E(i, j) value equal to the current time, t. If found, then allocate the subarray whose base is (i, j) to T and update AQ.
5. If no E(i, j) = t is found, then reverse the orientation of T from $h \times w$ to $w \times h$ and repeat Steps 3 and 4. Choose the minimum E(i, j) for the two orientations. If the value of E(i, j) is less than or equal to the latest starting time of T, then reserve the subarray with base (i, j) for T and update RQ. Otherwise, Phase 1 fails and the task allocator moves on to the next phase.

Example: We now introduce an example, which we will resume after each phase, to depict the actions of each phase. Overall, tasks arrive in the sequence shown in Table 5.1 and the FPGA on which these tasks are to run is of size 8×8.

Table 5.1: Task information for running example.

Task	Time available, a	Service time, s	Deadline, d	$h \times w$	LST	Laxity on arrival
T_1	1	7	13	6×4	7	6
T_2	2	9	16	4×3	8	6
T_3	3	4	14	3×3	11	8
T_4	4	5	15	4×3	11	7
T_5	5	8	21	4×4	14	9
T_6	6	5	15	5×4	11	5
T_7	7	4	13	4×3	10	3
T_8	8	6	15	3×5	10	2

Tasks T_1- T_3 can each start directly at the times they become available and tasks T_4 and T_5 cannot start immediately, but can be reserved. The task allocator uses only Phase 1 to process each of these tasks. At time 5, Table 5.2 depicts task status and Figure 5.2 depicts the placement of the active tasks. Recall that the base cell is the bottom leftmost cell of the subarray assigned to a task, and a service interval [BR96, HH95-B] indicates that the task starts at the beginning of time unit 2 and finishes at

Table 5.2: Task status at time = 5 in example.

Task	Active/ Reserved	Base cell	Service Interval
T_1	A	1, 1	[1, 7]
T_2	A	1, 5	[2, 10]
T_3	A	5, 5	[3, 6]
T_4	R	5, 5	[7, 11]
T_5	R	1, 1	[8, 15]

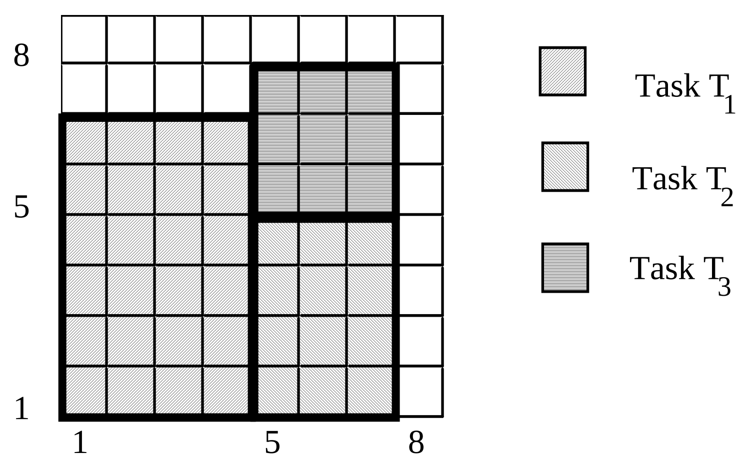

Figure 5.2: Placement of active tasks on FPGA at time = 5 in example.

the end of time unit 11, so the corresponding subarray is available up through time unit 1 and starting again in time unit 12.

5.3.2. Phase 2 — Reschedule Reserved Tasks

In this phase, the task allocator attempts to develop a new schedule for T and the reserved tasks so that each can complete execution by its deadline. It will run Phase 1 for the scheduling. Phase 2 will execute only if the laxity of task T is greater than the execution time of Phase 2 (shown in Section 5.2 to be O(n(m+n)rc) where m = number of active tasks and n = number of reserved tasks); otherwise, reject task T.

Procedure for Phase 2:

1. Let G denote the set of tasks in RQ whose laxity is greater than the laxity of T. Attempt to reserve T using the procedure in Phase 1 assuming that tasks in G have not yet been reserved.
2. If T can be reserved, then attempt to reserve all tasks in G in the order of increasing laxity using the procedure in Phase 1.
3. If T and the other tasks have been reserved successfully, then update RQ. Otherwise, Phase 2 fails, the tasks in G revert to their schedules at the start of Phase 2, and the task allocator moves on to Phase 3.

Example: We resume the example started in Phase 1 with the arrival of task T_6. At time 6, task T_6 arrives with service time 5, deadline 15, and size 5×4. It cannot be scheduled to execute in Phase 1, so the task allocator executes Phase 2. The task allocator checks the tasks in RQ and finds that T_5 has a greater laxity than T_6. It then schedules T_6 using Phase 1, while ignoring the reservation of T_5, then schedules T_5.

Table 5.3 depicts the new status of the tasks after rescheduling at time 6; Figure 5.2 still depicts the placement of the active tasks.

Table 5.3: Task status at time = 6 in example.

Task	Active/ reserved	Base cell	Service Interval
T_1	A	1, 1	[1, 7]
T_2	A	1, 5	[2, 10]
T_3	A	5, 5	[3, 6]
T_4	R	5, 5	[7, 11]
T_5	R	1, 1	[13, 20]
T_6	R	1, 1	[8, 12]

5.3.3. Phase 3 — Pre-empt an Active Task

Phase 3 will execute only if the laxity of task T is greater than the execution time of Phase 3 (shown in Section 5.4 to be O(mn(m+n)rc)); otherwise, reject task T. In this phase, the task allocator looks for an active task with (i) greater laxity than T, (ii) size at least as large as needed for T, and (iii) sufficient laxity that it can be pre-empted for the service time of T plus the time to reload its configuration and state and still meet its deadline. The allocator must also determine the ripple effect of pre-emption on reserved tasks.

Definition: An *affected task*, T_f, is a task that needs to be rescheduled due to the pre-emption of an active task T_k. An affected task is a reserved task whose subarray overlaps the subarray of T_k or overlaps the subarray of another affected task.

Pre-empting a task T_k can force rescheduling some or all affected tasks. Note that active tasks cannot be affected because of a lack of time and space overlap with T_k and rescheduled affected tasks [YY95].

Procedure for Phase 3:

1. For each task T_k in AQ whose laxity is larger than the laxity of T, test the following conditions. Process tasks in order of increasing laxity and stop when finding one that satisfies all three.

   a) Timing requirement: laxity(T_k) > laxity(T) and s(T) + t_2(T_k) + reload(T_k) ≤ d(T_k), where reload(T_k) is the time to load and restart T_k on the FPGA after T finishes.

   b) Space requirement: w(T) ≤ w(T_k) and h(T) ≤ h(T_k) (or w(T) ≤ h(T_k) and h(T) ≤ w(T_k)).

   c) Rescheduling requirement: All affected tasks must be rescheduled successfully.

   If the task allocator finds such a task T_k, then follow the steps below; otherwise, Phase 3 fails and the task allocator moves to Phase 4.

2. Update AQ to include T and update RQ for the affected tasks and the pre-empted task T_k. (Reserve T_k to run for its remaining service time after T finishes.)

3. The allocator pre-empts candidate task T_k and allocates a subarray of T_k to T for the time range equal to the service time of T.

Example: We resume the example continued in Phase 2 with the arrival of task T_7. At time 7, task T_7 arrives with service time 4, deadline 13, and size 4×3. It cannot be scheduled to execute in Phase 1 and the reserved tasks cannot be rescheduled to accommodate T_7 in Phase 2, so the task allocator executes Phase 3. The task allocator checks the tasks in AQ and finds that T_2 satisfies all three conditions. It then updates AQ and RQ, pre-empts T_2, then starts executing T_7 in a subarray (in this case the entire subarray) of T_2. Table 5.4 depicts the new status of the tasks at time 7 after pre-emption and Figure 5.3 depicts the placement of the active tasks.

Table 5.4: Task status at time = 7 in example.

Task	Active/ Reserved	Base cell	Service Interval
T_1	A	1, 1	[1, 7]
T_2	R	1, 5	[11, 14]
T_4	A	5, 5	[7, 11]
T_5	R	1, 1	[13, 20]
T_6	R	1, 1	[8, 12]
T_7	A	1, 5	[7, 10]

5.3.4. Phase 4 — Compact Tasks

Phase 4 will execute only if the laxity of T is greater than the execution time of Phase 4 (shown in Section 5.4 to be $O((m + n)^3)$); otherwise, reject task T. Let B denote the set of cells that are potential base sites for T. To minimize the number of cells to be relocated in a compaction, B contains the set of cells in the intersection of top cell

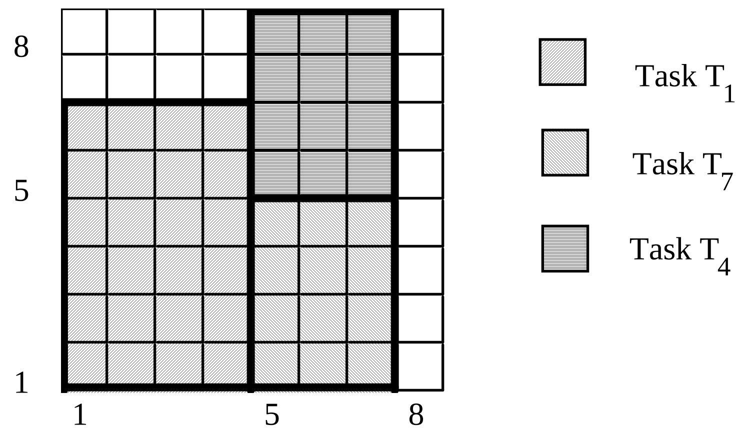

Figure 5.3: Placement of active tasks on FPGA at time = 7 in example.

intervals of active and reserved tasks (and the bottom row) and right cell intervals of active and reserved tasks (and the leftmost column); B excludes cells that are too close to the top end of the FPGA or right side of the FPGA to place T. Some cells in B may not be able to serve as a base for T since they may not be able to free up enough space for T by compacting other tasks to the right. To find a feasible subarray for T that involves a low cost of compaction, build a visibility graph of the active and reserved tasks. Since compaction of active tasks may have an impact on reserved tasks when they start, we consider both active and reserved tasks together in a single graph.

Procedure for Phase 4:

1. Build the visibility graph:

   a) Sort AQ and RQ into increasing base column order. If two or more tasks share a column, then sort them into increasing row order. For two or more tasks with the same base, sort them in order of increasing start time.

b) For each task in sorted order, insert a vertex into the graph as follows. Each vertex in the graph has associated with it the bottom and topmost rows and the time interval allocated to the corresponding task.

— For task T_k, insert vertex v_k in the graph.

— Perform a depth-first search of the graph. Continue the search to vertex v_g if the rows covered by task T_g overlap with the rows covered by task T_k and the time interval of T_g overlaps with the time interval of T_k. When no further overlap exists, create an edge from the last overlapping vertex v_g to v_k.

— Assign the distance T_g can move to the right to reach T_k to the edge from v_g to v_k. Store this as dist_edge(v_g, v_k).

— Backtrack and continue the search.

c) For each leaf v_k in the graph, compute the distance T_k can move to the right. (This is the distance to the right border of the FPGA, as no tasks to the right of T_k overlap the rows of T_k.) Store this distance as dist(v_k).

d) Proceeding bottom-up in the visibility graph, for each vertex v_g, compute the distance that T_g can move to the right after compaction as follows. For each child v_k of v_g, compute dist(v_k) + dist_edge(v_g, v_k); select the minimum such value as dist(v_g). For active task T_k, if the deadline of T_k is less than the sum of the current time, time taken for compaction, time to reload T_k, and remaining service time of T_k, then the distance that task can move to the right is zero. For reserved task R_k, if the deadline of R_k is less than the sum of the finish time of R_k and the time taken for compaction, then the distance that task can move to the right is zero. (This is a conservative estimate for reserved tasks, allowing for a ripple of delays propagating in time from the compaction.)

2. For each potential base $b \in B$, search those subgraphs whose covered rows intersect the allocation site based at b depth-first down to the leftmost tasks that intersect the allocation site. (If these leftmost tasks T_k can be moved based on dist(v_k), then so can tasks to the right.)

3. Check the possibility of moving each of these to free space for the incoming task. Find the minimum cost of freeing the allocation subarray and compacting the tasks to the right.

4. Scheduling the compaction:

   a) For active tasks: Compact tasks to the right starting from the rightmost task. Update finishing time of each task to account for compaction and reloading time.

   b) For reserved tasks: Change the location of the reserved subarrays. Update starting and finishing times of each task to account for delays due to compaction.

Example: We resume the example continued in Phase 3 with the arrival of task T_8. At time 8, task T_8 arrives with service time 6, deadline 15, and size 3×5. It cannot be scheduled to execute in Phase 1, the reserved tasks cannot be rescheduled in Phase 2 to accommodate T_8, and no task can be pre-empted in Phase 3 to place T_8, so the task allocator executes Phase 4. Figure 5.4 shows the visibility graph that the task allocator constructs, Table 5.5 depicts the new status of the tasks after compaction at time 8, and Figure 5.5 depicts the placement of the active tasks.

Table 5.5: Task status at time = 8 in example.

Task	Active/ Reserved	Base cell	Service Interval
T_2	R	1, 5	[11, 14]
T_4	A	5, 6	[7, 11]
T_5	R	1, 1	[13, 20]
T_6	A	1, 1	[8, 12]
T_7	A	1, 5	[7, 10]
T_8	A	6, 1	[8, 13]

5.4. Complexity Analysis

We now analyze the time complexity of the algorithm. Recall that the size of the FPGA is $r \times c$, the requested size of new task T is $h \times w$, the number of active tasks is m, and the number of reserved tasks is n. Phase 1 takes $O((m + n)rc)$ time to construct E because $r-h+1 = r$ and $c-w+1 = c$ in the worst case.

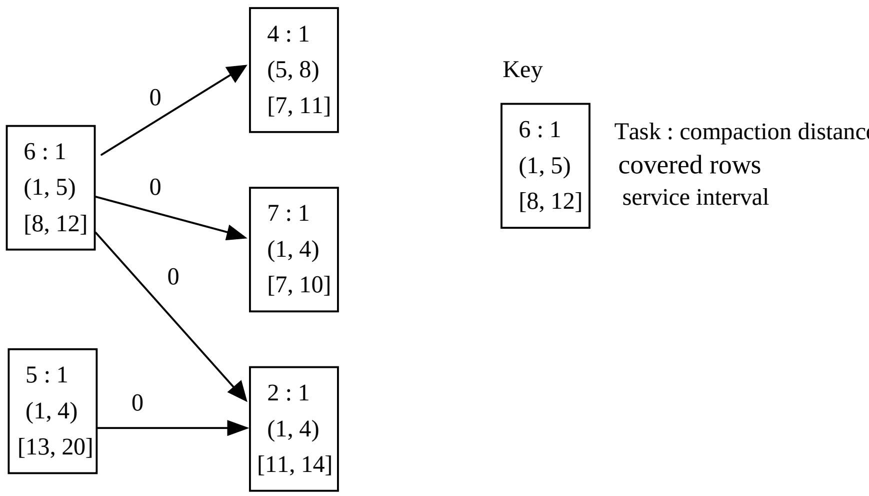

Figure 5.4: Visibility graph at constructed to place T_8 in example.

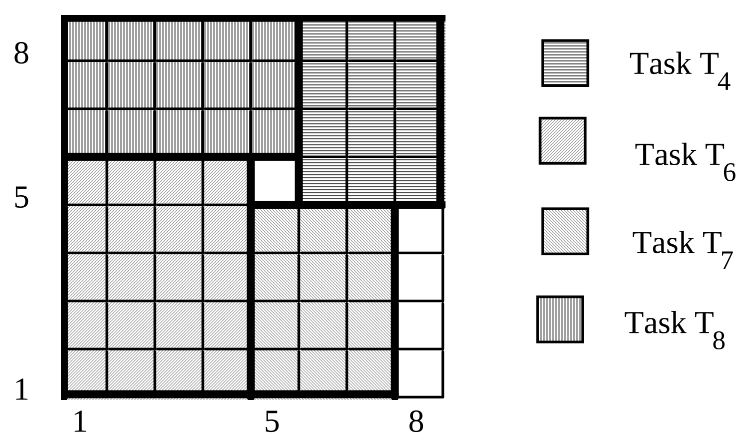

Figure 5.5: Placement of active tasks on FPGA at time = 8 in example

Also, this allows $O(rc)$ worst case size for each of $m + n$ non-allocable subarrays when constructing E. Constructing E dominates the time for Phase 1.

Phase 2 executes Phase 1 n times in the worst case (that is, all tasks in RQ have greater laxity). Hence, the complexity is $O(n(m + n)rc)$.

Phase 3 executes Phase 1 mn times in the worst case (that is, all tasks in AQ have greater laxity and must attempt to reschedule n affected tasks for each). Hence, the time complexity is $O(mn(m + n)rc)$.

In Phase 4, building the visibility graph takes $O((m + n)^2)$ time. The worst case size of B is $O(\min\{(m + n)^2, rc\})$. Checking the possibility of moving the tasks out of the way of the incoming task takes $O(m + n)$ time per base $b \in B$, which implies $O((m + n)^3)$ in the worst case. The overall time for Phase 4 is $O((m + n)^3)$.

CHAPTER 6

SIMULATION RESULTS

We evaluated the performance of the algorithm using software simulations. We report the results for varying task sizes, initial laxity, and inter-arrival times that would affect the performance of the algorithm on an FPGA.

The simulation model is as follows. Initially, an entire 64×64 FPGA is free and 10,000 tasks are generated and placed in a queue. The tasks are assumed to arrive at random intervals with random initial laxity and random sizes, according to the maximum inter-arrival time (initial laxity and task size, respectively) and distribution. The service period for each task was assigned randomly with a uniform distribution between 1 and 1000 time units. The time unit we considered is large enough so that the time needed for subarray allocation is negligible. We have considered the average instruction execution time as 1/10,000 of a time unit and the time to configure a single cell as 1/1000 of a time unit. (Configuration delay per cell is consistent with that of the experiments of Diessel and ElGindy [DE97-A], who investigated a range of configuration delays and observed that at mean configuration delays of less than 50% of the mean service period, compaction imparts a significant performance benefit.)

We report results collected from three independent runs to measure the performance under varying inter-arrival periods, initial laxities, and task sizes for uniform and increasing distributions. Each experiment involved fixing two of the maximum inter-arrival period, maximum initial laxity, and maximum task sizes, and varying the third. In all of these experiments, the fixed parameters are uniformly distributed and the varying parameter follows one of the uniform or increasing distributions. Tables 6.1 and 6.2 display results for varying ranges of inter-arrival times. For each phase, Table 6.1 shows the percentage of tasks reaching that phase that are allocated in that phase and Table 6.2 shows the task miss percentage, that is, the percentage of tasks unable to be scheduled and placed, after each phase. Tables 6.3 and 6.4 display results for varying ranges of initial laxities and Tables 6.5 and 6.6 for varying ranges of task sizes.

We observe that Phases 2-4 play a greater role as the FPGA is closer to saturation, as is expected. Contrasting the task miss percentage after Phase 1 with the task miss percentage after Phase 4 in Tables 6.2, 6.4, and 6.6 displays the utility of Phases 2-4. Figures 6.1-6.5 plot the results, and interpretations of the plots follow the tabular results. Since the major contributions of this thesis are in Phase 4, we focus on interpreting the performance benefits at the end of Phases 3 and 4. We follow the convention "Phase 4 over Phase 3" to imply "at the end of Phase 4 over at the end of Phase 3" throughout this chapter. All the results reported were taken at relative variations of saturating levels, caused due to the choice of fixed parameters in each experiment. The simulator handles the scheduling and compacting costs by the number of instructions executed and the average instruction time. We assume that the initial loading time is included in the service period. However, when a task is pre-empted or compacted, reloading costs, which are the costs incurred in storing the bit-stream data for each configuration by writing it to the external memory off the chip while pre-empting or compacting, and reading it back, and reconfiguration costs, which are costs incurred in loading the configuration data onto FPGA cells after the tasks have been moved around or pre-empted need to be considered. We neglected reloading times, but handled reconfiguration costs.

Table 6.1: Percentage of tasks reaching each phase that are allocated in that phase for varying inter-arrival time ranges; range of task sizes = [1, 32] and initial laxity = [1, 50].

Inter-arrival time [1, x]	Uniform 100	Uniform 500	Uniform 1200	Increasing 100	Increasing 500	Increasing 1200
Phase 1	75.10	81.50	91.80	77.50	85.40	94.30
Phase 2	16.18	16.21	12.19	17.82	20.27	12.10
Phase 3	10.63	16.57	18.05	9.15	22.68	18.16
Phase 4	9.8	24.17	15.6	17.22	31.89	16.09

Table 6.2: Task miss percentage after each phase for varying inter-arrival time ranges; range of task sizes = [1, 32] and initial laxity = [1, 50].

Inter-arrival time [1, x]	Uniform 100	Uniform 500	Uniform 1200	Increasing 100	Increasing 500	Increasing 1200
Phase 1	24.9	18.5	8.2	22.5	14.6	5.7
Phase 2	20.87	15.5	7.2	18.49	11.64	5.01
Phase 3	18.65	12.87	5.9	16.78	9.0	4.1
Phase 4	16.82	9.67	4.98	13.89	6.13	3.44

6.1. Effect of Inter-arrival Time on System Performance

We measure the performance of the proposed scheme for varying inter-arrival periods and evaluate the performance improvement of Phase 4 over Phase 3. The service time,

Table 6.3: Percentage of tasks reaching each phase that are allocated in that phase for varying initial laxity ranges; range of task sizes = [1, 32] and task inter-arrival times = [1, 500].

Initial Laxity range [1, x]	Uniform			Increasing		
	50	100	200	50	100	200
Phase 1	79.80	82.40	85.10	80.55	84.79	87.00
Phase 2	29.70	10.97	15.37	5.4	11.30	14.85
Phase 3	8.70	8.17	10.39	11.9	22.30	18.69
Phase 4	8.81	16.54	8.4	15.5	24.04	12.44

Table 6.4: Task miss percentage after each phase for varying initial laxity ranges; range of task sizes = [1, 32] and task inter-arrival times = [1, 500].

Initial Laxity range [1, x]	Uniform			Increasing		
	50	100	200	50	100	200
Phase 1	20.2	17.6	14.9	19.15	15.21	13
Phase 2	19.16	15.67	12.61	18.09	13.49	11.07
Phase 3	17.59	14.39	11.3	15.9	10.48	9
Phase 4	16	12.01	10.35	13.38	7.96	7.88

task size, and initial laxities generated for each task ranged uniformly between 1-1000 units, 1-32, and 1-100 units respectively. Here is the comparative analysis of the results for Phase 4 over Phase 3, for uniformly distributed inter-arrival times. Figures 6.1 and 6.2 chart the task miss percentages against inter-arrival times.

At maximum inter-task arrival periods below 100 time units, the FPGA was over-saturated with work as tasks arrived more frequently than space was available in which they could be allocated. The rate at which tasks were allocated in Phase 4 is dependent on the ability of the allocation method to find a suitable region for the task

Table 6.5: Percentage of tasks reaching each phase that are allocated in that phase for varying task size ranges; range of task inter-arrival times = [1, 500] and initial laxity = [1, 100].

Task size range [1, x]	Uniform			Increasing		
	10	32	64	10	32	64
Phase 1	94.91	83.85	75.28	87.70	73.10	60.20
Phase 2	29.86	21.98	10.47	22.60	15.76	7.28
Phase 3	23.8	13.49	5.10	18.90	12.18	3.25
Phase 4	51.1	25.68	4.76	36.50	20.10	1.40

Table 6.6: Task miss percentage after each phase for varying task size ranges; range of task inter-arrival times = [1, 500] and initial laxity = [1, 100].

Task size range [1, x]	Uniform			Increasing		
	10	32	64	10	32	64
Phase 1	5.09	16.15	24.72	12.3	26.8	39.8
Phase 2	3.57	12.6	22.13	9.52	22.56	36.9
Phase 3	2.72	10.9	21	7.72	19.8	35.7
Phase 4	1.33	8.1	20	4.9	15.8	35.2

at the head of the queue and also find a region so that all the tasks to be compacted meet their deadlines. In order to satisfy the next request in the queue, only a fraction of cells freed up as a result of tasks finishing, immediately satisfied the next request, as tasks arrived rapidly. The benefit due to compaction, however, increases for inter-arrival periods between 1 and 500 units as packing of the FPGA decreases and more tasks can be moved around for compaction while satisfying deadlines. When the inter-arrival periods are between 1 and 1200, tasks leave the chip more rapidly than they arrive. Thus, at maximum inter-arrival time of 1200, there is only slight benefit to compacting tasks to allocate more quickly. For exponentially distributed inter-arrival times, the interpretation is similar to the above, except that the percentage of tasks missed is comparatively less than that of uniform distribution, because in the former case tasks typically arrive further apart than in the latter case. We observe that the greatest improvements of Phase 4 over Phases 3 are 24.88% and 31% for uniform and exponential distributions, respectively.

6.2. Effect of Initial Laxity on System Performance

We investigated the dependence of scheduling performance on initial laxities in three different load regions: at saturation, where the benefit of Phase 4 over Phase 3 is small; in the range of loads where the FPGA is coming out of saturation, where the benefit of Phase 4 over Phase 3 is greatest; and in the range of loads that do not saturate the chip, where the benefit of Phase 4 over Phase 3 is decreasing. We measured the performance at maximum uniformly distributed inter-task arrival times of 500 units, maximum uniformly distributed task sizes of 32 units, and maximum uniformly distributed service time of 1000 units, while the initial laxities were randomly assigned with an exponentially increasing distribution in the ranges 1-50 units, 1-100 units, and 1-200 units. Figures 6.3 and 6.4 plot tasks miss percentages against initial laxities for uniform and increasing distributions, respectively. The performance measured was at three initial laxity ranges, all of which reflect saturated conditions because of service time (1-1000) and inter-arrival time (1-500), but the relative rate of saturation varies.

Initial laxity between 1 – 50 means that tasks need to be serviced quickly, and so heavily saturate the FPGA. Initial laxity between 1-100 means that there could be some wait period before they can be serviced, which creates moderate saturating conditions, while initial laxity of 1 – 1000 means that tasks could wait for a long period before being serviced, and hence the saturating levels decrease.

As we can observe from the chart, as the initial laxity increases from maximum saturating levels, the performance benefit of Phase 4 over Phase 3 rises to a maximum because more tasks can be compacted due to relative availability of more space on the FPGA, and relatively fewer deadline bottlenecks encountered for moving the tasks. When the initial laxities no longer cause saturation, the benefit of Phase 4 over Phases 3 is decreasing. Similar observations are made for initial laxities following exponentially increasing distribution, except that the tasks miss percentages in each phase are lower than with uniform distribution, as under exponential distribution the initial laxities are typically higher. We find that Phase 4 improves over Phase 3 by 16.5% and 24.04% for uniform and exponential distributions respectively, under moderate saturating levels.

6.3. Effect of Task Size on System Performance

We report the dependence of the results on task size in this section. We ran simulations with task service period, initial laxity, and inter-arrival period ranging uniformly from 1-1000 units, 1-100 units and 1-500 units, respectively, while task sizes have been varied uniformly and exponentially between 1-10, 1-32, and 1-64. Figures 6.5 and 6.6 plot the task miss percentages against task sizes. FPGA saturation occurs for task size ranges between 1-32 and 1-64, as only few large tasks can be accommodated on the FPGA. As the charts indicate, the performance benefit of Phase 4 over Phase 3 for task sizes between 1-10 is small, as smaller tasks are easily accomodated without compaction. However, the benefit increases tremendously as task sizes increases to a maximum of 32 cells per side, as more tasks can be completed

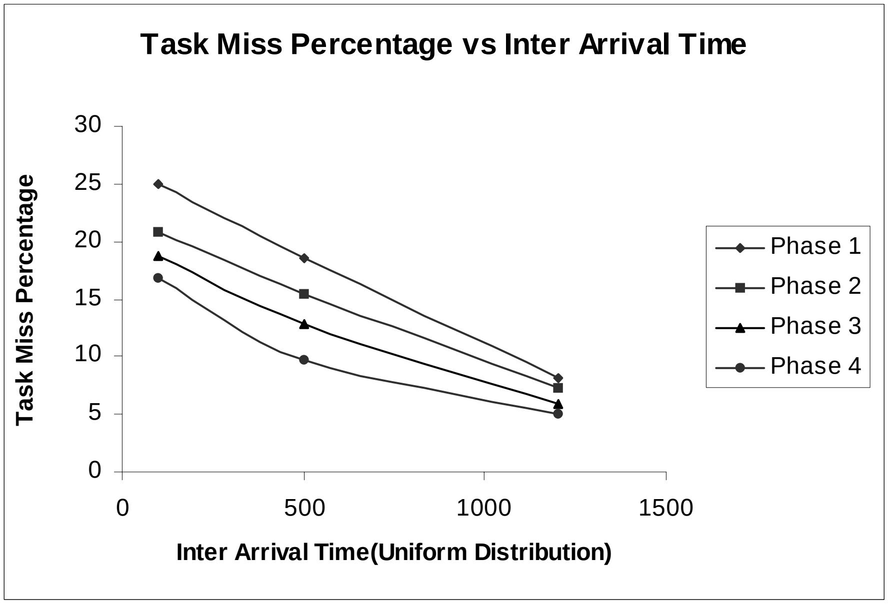

Figure 6.1: Task miss percentages for uniformly varying inter-arrival times

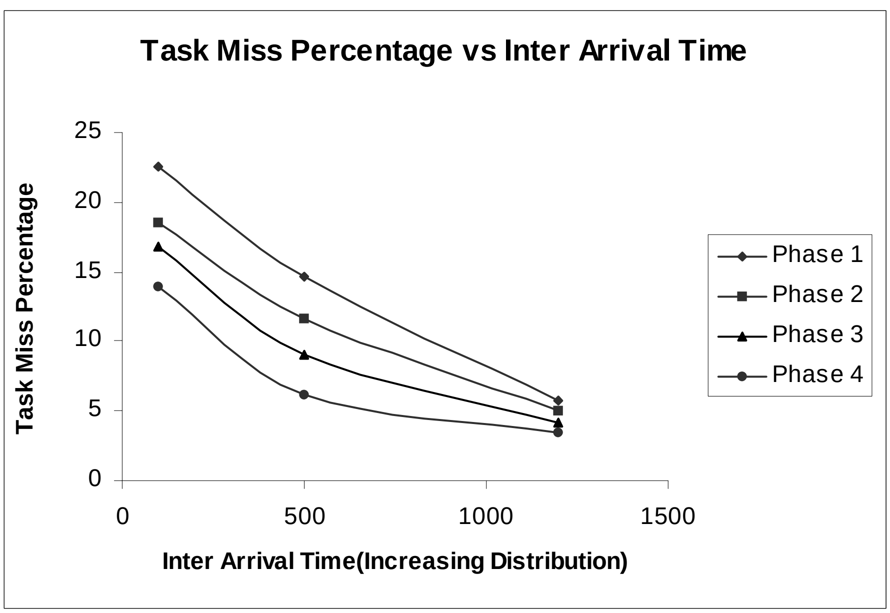

Figure 6.2: Task miss percentages for exponentially varying inter-arrival times.

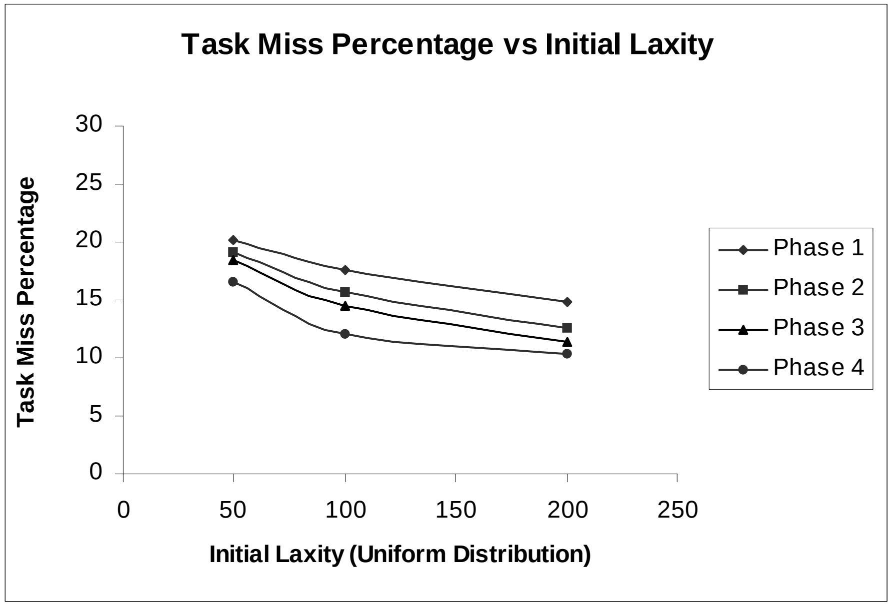

Figure 6.3: Task miss percentages for uniformly varying initial laxity

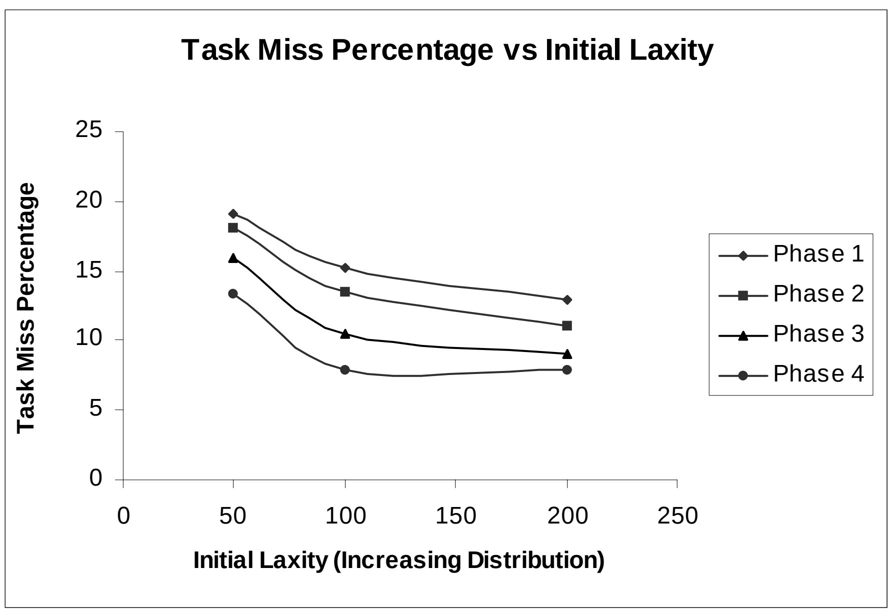

Figure 6.4: Task miss percentages for exponentially varying initial laxity.

Figure 6.5: Task miss percentages for uniformly varying task size

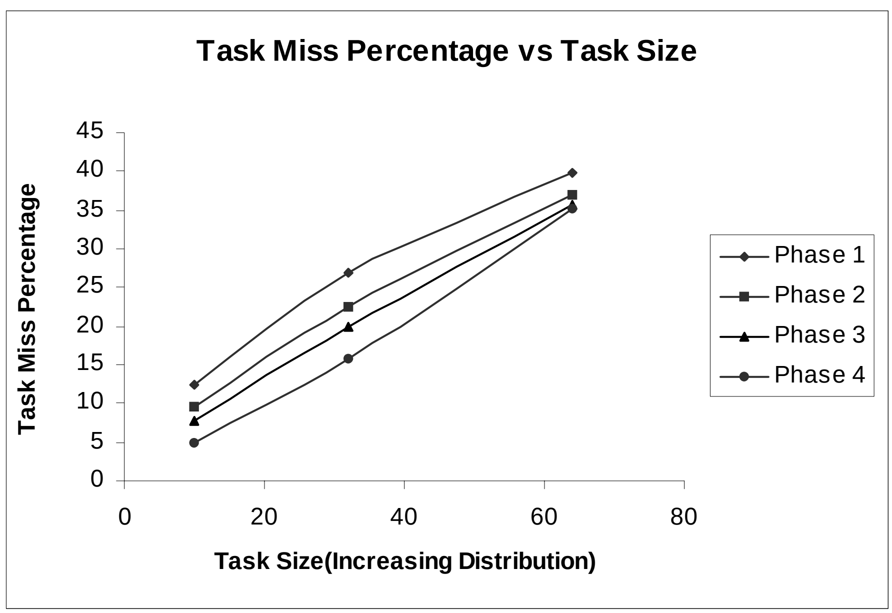

Figure 6.6: Task miss percentages for exponentially varying task size

without compaction. However, the benefit increases tremendously as task sizes increases to a maximum of 32 cells per side, as more tasks can be accommodated with compaction. The benefit decreases again as the maximum task size approached the chip size, as there will not be enough free cells to move the tasks around. The greatest improvements of Phase 4 over Phase 3 are 25.68% for uniform distribution and 20.20% for exponential.

CHAPTER 7

SUMMARY AND OPEN PROBLEMS

We have designed a task allocator that accounts for real-time constraints for an FPGA shared among multiple tasks or users. Using laxity as a priority criterion, the allocator processes tasks as they arrive, attempting to place a task to execute immediately or schedule it to execute in the future. If these fail, then it attempts to reschedule reserved tasks, pre-empt an active task, or compact active and reserved tasks. This work takes a step toward expanding the scope and utility of configurable computing.

Open areas for research include: incorporating constraints on task placement due to irregularities in FPGA resources; considering different constraints on I/O resources with respect to different task placements; and developing more efficient algorithms for the different phases.

REFERENCES

[AT6000] Atmel data sheet for AT6000 series: http://www.atmel.com.

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

[BDH97] Burns, J., Donlin, A., Hogg, J., Singh, S. and de Wit, M. (1997), "A Dynamic Reconfiguration Run-Time System", Proc. IEEE Symp. FPGAs for Custom Computing Machines, pp. 66-75.

[BR96] Brown, S. and Rose, J. (1996), "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, pp. 42-57

[C99] G. M. Chiu (1999), “An Efficient Submesh Allocation Scheme for Two-Dimensional Meshes with Little Overhead,” IEEE Transactions on Parallel and Distr. Sys , Vol. 10, No. 5, pp. 471-486.

[DE97-A] Diessel, O. and ElGindy, H. (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.

[DE97-B] Diessel, O. and H.ElGindy (1997), “Partial FPGA Rearrangement by Local Repacking,” Technical Report 97-02, Dept. of Comp. Sci. and Software Engr., Univ. of Newcastle, Australia.

[DEM00] Diessel, O., ElGindy, H., Middendorf, M., Schmeck, H. and Schmidt, B. (2000), "Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs", IEEE Proceedings-Computers and Digital Techniques, vol. 147, no. 3, pp. 181-188.

[DH91] Dutt, S. and Hayes, J.P. (1991), "Subcube Allocation in Hypercube Computers", IEEE Trans. Comput., vol. 40, no. 3, pp. 341-352.

[DKW99] Diessel, O., Kearney, D. and Wigley, G. (1999), "A Web-Based Multiuser Operating System for Reconfigurable Computing", Proc. 6th Reconfig. Arch. Workshop (Parallel and Distributed Processing; Lect. Notes Comp. Sci. #1586), pp. 579-587.

[DRDT] “Dynamically Reconfigurable Devices & Technology: http://dec.bournemouth.ac.uk/drhw_lib/technology.html#virtex .

[DW99] Diessel, O. and Wigley, G. (1999), "Opportunities for Operating Systems

Research in Reconfigurable Computing", Technical Report ACRC-99-018, School of Comput. and Info. Sci., Univ. of South Australia.

[EMS00] ElGindy, H., Middendorf, M., Schmeck, H. and Schmidt, B. (2000), " Task Rearrangement of Partially Reconfigurable FPGAs with Restricted Buffer", Proc. 10th Int'l Workshop Field-Programmable Logic and Applic. (Lect. Notes Comp. Sci. #1896), pp. 379-388.

[F95] B. Faucett (1995), "FPGAs as Configurable Computing Elements," Proc. Reconfig. Arch. Workshop.

[FP98] Fornaciari, W. and Piuri, V. (1998), "Virtual FPGAs: Some Steps Behind the Physical Barriers", Proc. Reconfig. Arch. Workshop (Lect. Notes Comp. Sci. #1388), pp. 7-12.

[G00] S. Guccione, "List of FPGA-based Computing Machines", http://www.io.com/~guccione/HW_list.html

[H98] Hauck, S. (1998), "The Roles of FPGAs in Reprogrammable Systems", Proc. IEEE, vol. 86, no. 4, pp. 615-638.

[HH95-A] J. D. Hadley and B. L. Hutchings (1995), "Design Methodologies for Partially Reconfigurable Systems," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, pp. 78-84.

[HH95-B] Hadley, J.D. and Hutchings, B.L. (1995), "Designing a Partially Reconfigured System", in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, J. Schewel, ed., Proc. SPIE, vol. 2607, pp.210-220.

[JTY99] Jean, J.S.N., Tomko, K., Yavagal, V., Shah, J. and Cook, R. (1999), "Dynamic Reconfiguration to Support Concurrent Applications", IEEE Trans. Comput., vol. 48, no. 6, pp. 591-602.

[LC89] K. Li and K. H. Cheng (1989), "Complexity of Resource Allocation and Job Scheduling Problems on Partitionable Mesh Connected Systems", Proc. 1st Symp. Patallel and Distr. Process, pp. 358-365.

[LS96] P. Lysaght and J. Stockwood (1996), "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays," IEEE Transactions on Very Large Sacle Integration (VLSI) Systems, Vol. 4, No. 3, pp. 41-50.

[M97] P. Mohapatra (1997), "Dynamic Real-time Task Scheduling on Hypercubes," Journal of Parallel and Distr. Comp., Vol. 46, No. 1, pp. 91-100.

[MD78] Mok, A.K. and Dertouzos, M.L. (1978), "Microprocessor Scheduling in a Hard-Time Environment", Proc. 7th Texas Conf. Comput. Sys.

[MH97] W.Magione-Smith and B.Hutchings (1997), “Configurable Computing: The Road Ahead,” Proc. Reconfig. Arch. Workshop (Microsystems Engineering Series), pp. 81-96.

[MHA97] W. Mangione-Smith and B. Hutchings, D. Andrews, A. Dehone, C. Ebeling, R. Hartenstein , O. Mencer, J. Morris, V. Prasanna, and H. Spaaneburg (1997), “Seeking Solutions in Configurable Computing,” IEEE Computer, pp. 38-43.

[MPR93] R. Miller, V. Prasanna, D.I.Reisis, and Q. Stout (1993), “Parallel Computations on Reconfgurable Meshes,” IEEE Transactions on Computers, Vol. 42, No. 6, pp. 678-692.

[NMN99] H. Nagano, A. Matsura, and A. Nagoya (1999), “An Efficient Implementation of Fractal Image Compression on Dynamically Reconfigurable Architecture,” Proc. 6th Reconfig. Arch. Workshop (Parallel and Distributed Processing; Lect. Notes Comp. Sci. #1586), pp. 670-678.

[RGS93] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli (1993), “Architecture of Field-Programmable Gate Arrays,” Proc. IEEE, vol. 81, no. 7, pp. 1013-1029.

[RL99] D. Robinson and P. Lysaght (1999), “Modelling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems Using the DCS CAD Framework,” Proc. 9th Int’l. Workshop Field-Programmable Logic and Applic. (Lect. Notes Comp. Sci. # 1673), pp. 41-50.

[SLC98] N. Shirazi, W. Luk, and P. Y. K. Cheung (1998), “Run-Time Management of Dynamically Reconfigurable Designs,” Proc. 8th Int’l. Workshop Field-Programmable Logic and Applic. (Lect. Notes Comp. Sci. #1482), pp. 59-68.

[SML98] S. M. Scalera, J. J. Murray, and S. Lease (1998), “A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing,” Proc. Reconfig. Arch. Workshop (Lect. Notes Comp. Sci. #1388), pp. 73-78.

[SO98] J. Spillane and H. Owen (1998), “Temporal Partitioning for Partially Reconfigurable-Field-Programmable Gate,” Proc. Reconfig. Arch. Workshop (Lect. Notes Comp. Sci. #1388), pp. 37-42.

[SP96] D. Sharma and D. Pradhan (1996), “Submesh Allocation in Mesh Multicomputers Using Busy-List: A Best-Fit Approach with Complete Recognition Capability,” Journal of Parallel and Distr. Comp., Vol. 36, No. 2, pp. 106-118.

[VCC] Virtual Computer Corporation: http://www.vcc.com

[VM97] J. Villasenor and W. H. Mangione-Smith (1997), "Configurable Computing," Scientific American, vol. 276, no. 6, pp. 66-71.

[WE99] M. Wojko and H. ElGindy (1999), "Configuration Sequencing with Self-Configurable Binary Multipliers," Proc. 6th Reconfig. Arch. Workshop (Parallel and Distributed Processing; Lect. Notes Comp. Sci. #1586), pp. 643-651.

[WH95] M. J. Wirthlin and B. L. Hutchings (1995), "DISC: The Dynamic Instruction Set Computer," in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, J. Schewel, ed., Proc. SPIE, vol. 2607, pp. 92-103.

[XILINX] Xilinx data sheets for XC3000, XC4000, XC6000, XC6200 series: http://www.xilinx.com.

[Y97] S. M. Yoo (1997), "An Efficient Task Allocation Scheme for 2D Mesh Architectures," IEEE Transactions on Parallel and Distr. Sys., Vol. 8, No.9 , pp. 934-942.

[YC99] S. M. Yoo, H. Choo, H. Y. Youn, C. Yu and Y. Lee (1999), "On Task Relocation in Two-Dimensional Meshes," Journal of Parallel and Distr. Comp., pp. 616-638.

[YY95] S. M. Yoo and H. Y. Youn (1995), "An On-Line Scheduling and Allocation Scheme for Real-Time Tasks in 2D Meshes," Proc. 7th IEEE Symp. Par. and Distr. Processing, pp. 630-637.

VITA

Shobharani Tatineni, daughter of Benarjee Tatineni and Vijaya Laksmi Tatineni, was born on the 26th day of July 1975, in Krishna District, Andhra Pradesh, India. She did her schooling at Siddartha High School, Hyderabad, subsequent to which she joined Chaitanya Junior College, Hyderabad, to pursue intermediate studies. She then entered the Department of Electrical and Electronics Engineering, Andhra University, in 1993 and obtained the degree of Bachelor of Electrical and Electronics Engineering in June 1997. In August 1997, she entered the graduate program in the Electrical and Computer Enginerring Department at Lousiana State University. During her enrollment in the graduate school, she served as a graduate research assistant at the Coastal Studies Institute in the Department of Oceanography and Coastal Sciences.