On Scheduling Dynamic FPGA Reconfigurations

A Partial Rearrangement Approach

Oliver Frank Diessel

B.Math., B.E. (Hons)

A thesis submitted in fulfilment of
the requirements for the degree of

Doctor of Philosophy

The Department of
Computer Science and Software Engineering
The University of Newcastle
Callaghan NSW 2308
Australia

January, 1998

Certificate of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

(Signed) _____

Oliver Frank Diessel

Acknowledgments

It has been an inspiration and a pleasure to be guided in this work by Hossam ElGindy. His vision, enthusiasm, and expertise motivated me as much as I appreciated and benefited from his open warmth and generosity. The foundations for this work were shaped by many enriching discussions with Bryan Beresford–Smith, who also inspired me to start.

I have had a great deal of assistance from the staff, students, and visitors to the Departments of Computer Science and Software Engineering and Electrical and Computer Engineering. In particular, I thank Mathew Wojko and Lachlan Wetherall for reviewing, discussing, and helping me clarify many aspects of this work. Thanks also to Peter Eades, Michael Hannaford, Arun Somani, and the CS&SE technical staff for providing infrastructure support. The financial support of a University of Newcastle Research Scholarship is gratefully acknowledged.

Finally, I would not have begun nor been able to complete this work without the love, support, and encouragement of my family and friends.

Contents

Abstract

The ability to partially reconfigure dynamically reconfigurable Field–Programmable Gate Arrays (FPGAs) at run–time allows them to be shared among multiple independent tasks. When the sequence of tasks to be performed is known in advance, the FPGA controller can optimize the use of shared resources. However, when the sequence is not predictable, or the task designs are not fixed, the controller needs to make allocation decisions on–line. Since on–line allocation suffers from fragmentation as variously sized tasks arrive and depart, tasks can end up waiting despite there being sufficient, albeit non–contiguous resources available to service them. The time to complete tasks is consequently longer and the utilization of the FPGA is lower than it could be.

This thesis proposes rearranging a subset of the tasks executing on the FPGA when doing so allows the next pending task to be processed sooner. Partial rearrangement proceeds in two steps. The first step identifies a rearrangement of the executing tasks that frees sufficient space for the waiting task, and the second schedules the movement of the tasks so as to minimize the delay in executing them. The scheduling strategies employed depend upon the methods available to move the tasks. Thus the problem of identifying the best rearrangement is linked by feedback through the schedule to the underlying hardware and its capabilities.

Current FPGA architectures allow tasks to be moved by reloading them. Simulation results indicate that significant reductions in allocation delays are possible when the FPGA is saturated with work and the time to load a task is relatively short. However, the reloading tasks face an I/O bottleneck that must be eliminated if partial rearrangements are to be practical for short–lived tasks. Techniques for copying tasks to their destinations over on–chip routing resources are therefore developed. These methods appear to be effective, even when link delays are long, giving hope that they may also be of use in boosting the performance of multiple–SIMD mesh connected computers.

Chapter 1

Introduction

This thesis proposes and assesses methods for rearranging a subset of the tasks configured on a Field–Programmable Gate Array (FPGA). The thesis considers moving the tasks by reloading them from off–chip storage and by copying them over on–chip routing resources. This chapter motivates and outlines the work described in the thesis.

The first section of this chapter describes the inspiration for developing space-shared multi–tasking FPGAs. The second section motivates the theme after summarizing the challenges facing the designers of space–shared FPGA systems. The contributions of this thesis and those of earlier investigators are reviewed in the third section. The fourth section then presents an outline of the thesis. The chapter concludes with a statement on related publications.

1.1 Historical Developments

Field–Programmable Gate Arrays occupy an increasingly important niche in the price/performance spectrum of computing hardware. At one end of the spectrum, a general purpose processor makes use of the von Neumann model of computation to execute a program by stepping through a sequence of stored instructions. The strength of this model lies in the flexibility of performing different computations by altering the instruction sequence. Unfortunately, the rate at which instructions are executed and the number of bits operated upon by each instruction limit the speed of processing. Specialized architectures and techniques that exploit parallelism are therefore sought when applications require levels of performance that cannot be delivered by sequential processors. For the majority of applications however, they

are perfectly adequate, and large production volumes thus allow manufacturers to market popular processors for relatively low prices.

At the other end of the spectrum, application–specific integrated circuits (ASICs) are designed to compute a particular function, thereby gaining a performance advantage over general purpose processors in a number of ways. First, components such as decoders and multiplexors that are intended to support arbitrary instruction streams but which take time for signals to pass through can be removed from the circuit. Second, by eliminating the need for hardware to fetch, decode, or cache instructions, additional chip area can be allocated to performing computational tasks. Third, instructions that are performed serially at both the bit and word level on the sequential processor can be performed in parallel on the different bits of a word and the different stages of a pipeline. However, once fabricated, ASICs are fixed in function, and their function is limited to the one intended at the design stage. ASICs thus sacrifice flexibility to achieve maximum performance. Moreover, the complexity of designing and testing ASICs leads to long manufacturing lead times and high non–recurring engineering costs. However, for large production volumes, the cost of ASICs can be lower than for general purpose processors.

Current FPGAs, on the other hand, consist of two–dimensional arrays of programmable logic elements and interconnections on a single chip that can be mass–produced and then customized in software to suit specific purposes. The function of the FPGA is programmed by configuring the function of individual logic cells and the interconnections between them. As with ASICs, performance advantages over general purpose processors can be gained by performing instructions in parallel on the bits of a word and the stages of a pipeline. However, as with general purpose processors, supporting flexibility comes at the cost of electrical delays due to configurable components and increased area due to the size of these components and the need to provide sufficient resources to implement circuits of varying complexity. Nevertheless, for small to medium volumes, these costs are offset by significantly lower implementation costs than for ASICs.

Historically, FPGA circuits were configured by programming anti–fuse, UV–EPROM, or EEPROM [53], and were therefore static. The introduction of FPGAs with configuration memory implemented in static RAM allowed the configuration to be modified by altering memory contents during normal operations [31]. However, altering the FPGA function necessitated halting the device and loading a new configuration for the entire chip. Since configuration data had to be loaded

serially, the time needed to reconfigure a circuit was large and depended upon the area of the chip. FPGAs that provide random access to configuration memory were recently introduced to overcome the excessive reconfiguration delay [2, 69]. Such FPGAs are able to reduce reconfiguration delays in two important ways. First, parts of the FPGA not being reconfigured may continue to operate and are therefore not delayed at all. Second, the time to configure a circuit depends upon its size since the configuration bit stream need only load relevant cells.

FPGAs that provide random access to configuration memory while the device is active are known as dynamically or partially reconfigurable. Dynamically reconfigurable FPGAs are used by decomposing the application or FPGA task into a number of mutually time exclusive modules or subtasks that are loaded at run–time as needed [37]. When a module is no longer needed, its resources can be reused by modules that are required but not yet loaded. The range and number of applications reported for dynamically reconfigurable FPGAs is rapidly increasing. Examples of applications so far reported are digital signal processing [54], image processing [28], neural processing [33], video coding [67], simulation [30], string searching [32], and postscript processing [61]. Application areas that may benefit from dynamic reconfiguration include video–on–demand hardware, mobile computing, data encryption, content–based searching, and multiprocessor cache coherence protocols [52]. As more ambitious systems are developed, it is conceivable that it will become possible and desirable for related or even disparate functions to share the one hardware platform.

Partial reconfiguration recycles resources that are not currently used for circuits that are currently needed. A good example of partial reconfiguration is the DISC system, which makes use of a well–defined global context to implement relocatable FPGA tasks of arbitrary size [68]. Tasks that occupy the entire width and an arbitrary number of rows of the FPGA may be located at any row, allowing physical placement of hardware to be determined at run–time. When a new task is to be executed on the FPGA and there is insufficient contiguous space for it, the least recently used tasks are removed from the system to allow for the new task to be placed. The effective area of the FPGA is thus increased by simulating many FPGAs or a much larger FPGA on a small one. While the well–defined global context simplifies DISC task design, it restricts the use of the FPGA to a single task or user at a time so as to prevent tasks from contending for the use of control lines that span the length of the chip. Interest in exploiting partial reconfiguration

to share the FPGA amongst multiple simultaneous tasks and/or users is, however, growing [11, 25, 55].

Switching between configurations to time–share an FPGA among several tasks or users is under investigation for the Garp system [34], which consists of a multi–tasking MIPS processor and an FPGA coprocessor integrated onto a single die. Garp tasks are also prevented from contending for the use of globally spanning control lines by executing one at a time. However, the system supports multiple users by providing instructions to save and restore configurations in array memory when a context switch occurs. The number of multiple users supported in this fashion is limited by the amount of on–chip memory. However, current dynamically reconfigurable FPGAs already use several bytes of valuable space per cell to store configuration and register data, and saving FPGA contexts off–chip becomes infeasible as FPGA sizes grow. A purely time–shared approach to multi–tasking therefore limits the maximum number of users.

While DISC and Garp allow the low–level parallelism inherent in applications to be exploited, much of the FPGA resource may remain idle because tasks are processed sequentially. This inefficiency is exacerbated as FPGA sizes grow. In order to utilize the unused portions of devices and to reduce response times by processing tasks in parallel, future FPGA systems need to support the partitioning of available resources amongst independent tasks that can be processed simultaneously. This technique, known as space–sharing, allows each of the multiple tasks to execute without interruption within its own partition as if it were the sole application executing on an FPGA that is just large enough to support it.

1.2 Motivation — The Space–Sharing Challenge

The design and implementation of space–shared FPGAs poses many challenging problems. Multi–tasking systems not only need to support individual users with design tools and monitors, they also need to manage the resources shared by all users. At the lowest operating system level, the sharing amongst several users of logic elements, wires, and input/output (I/O) pins needs to be managed. At a higher level, access to resources needs to be scheduled according to task priorities. Some of the challenging optimization problems that need to be solved to support multiple simultaneous tasks are listed below.

Hardware support

How should the hardware be made partitionable so that multiple independent tasks can execute? Can multiple simultaneous I/O streams be supported? If so, can multiple tasks be configured and controlled simultaneously? Do multiple clocks need to be supported?

Task design

The operating system should be responsible for the choice of task location; a task must therefore be relocatable. How should relocatable tasks be designed so that they do not interfere with neighbouring tasks? What are the global requirements for supporting arbitrary task location?

Run–time binding

Since tasks need to be relocatable, there is a need for some run–time binding. Task partitioning, placement, and routing need to adapt according to temporal and spatial constraints. How much preprocessing of the final design can be done? What needs to be done on–the–fly? How can it be done efficiently?

Task relocation

In order to maximize utilization and to be fault tolerant, the system also needs to support preemption and to implement garbage collection. How can the overheads of the operating system be minimized? Should operating system functions be implemented as dynamically reconfigurable tasks? If so, how?

Effective solutions to these problems will make space–shared FPGA systems easier to use, more powerful, and more cost effective. These factors in turn will contribute to the attractiveness and more general use of systems employing reconfigurable hardware.

This thesis examines one strategy for boosting performance by defragmenting space–shared FPGAs. Performance loss due to fragmentation comes about as follows. When the sequence of tasks to be performed by an FPGA is known in advance the designer can optimize the use of resources off–line and design an appropriate static controller. However, when the set of tasks to be executed is not closed, the controller needs to make task placement (cell allocation) decisions on–line. The layout of a task is assumed to comprise a contiguous block of logic cells and interconnections that are known before the task is to be configured onto the

array. Thus "allocating" a task is the process of deciding where on the array to place the layout. On–line allocation of contiguous resources is by nature suboptimal. Since users want rapid response, the goal of allocation is to allocate as quickly as possible because task execution times are fixed. However, when the task sequence and/or execution times are not known, it is impossible for the allocator to place a task so as to guarantee minimum impact on subsequent tasks because contiguous allocation schemes suffer from fragmentation as variously sized tasks are allocated and deallocated. Tasks consequently end up waiting in a queue despite there being sufficient, albeit non–contiguous resources available to service them. The response times of tasks are therefore longer and the utilization of FPGA resources is lower than they could be. There is thus a need for a means of minimizing the consequences of fragmentation.

1.3 Contributions

This thesis proposes rearranging, at run–time, a subset of the tasks executing on the FPGA so as to aggregate sufficient contiguous space for the next waiting task when doing so allows it to be allocated sooner. The rearrangement goals are to minimize the delay to the next waiting task, to minimize the delays to executing tasks that have to move, and to minimize the time needed to complete the rearrangement.

A partial rearrangement at run–time involves moving a subset of the tasks executing on the FPGA while the remaining tasks continue to execute. When the task at the head of the pending queue cannot be allocated immediately because of fragmentation, a procedure that attempts to identify a rearrangement of a subset of the currently allocated tasks to accommodate the waiting task is invoked. If such a rearrangement is found, then a schedule for moving the executing tasks that allocates the waiting task as soon as possible and minimizes the delays to executing tasks that are to be moved is computed. The thesis considers three task movement models: moving the tasks by reloading them from off the chip; copying the tasks on–chip over nearest neighbour links; and copying the tasks over segmentable buses. The scheduling methods needed are unique to each model.

The main contributions of this thesis are:

- Two new heuristics for the NP–complete problem of identifying rearrangements of FPGA tasks that will accommodate an additional task. These solu-

tions, respectively known as local repacking and ordered compaction, apply to rectangle packing situations which permit repacking. In particular, they apply to the problem of allocating mesh of processor tasks.

- A proof that it is NP–hard to schedule arbitrary rearrangements of FPGA tasks so as to minimize delays to tasks when they are moved by reloading.
- A depth–first ordered search heuristic for scheduling local repackings by reloading.
- An optimal algorithm for scheduling ordered compactions by reloading.
- An optimal algorithm for the ordered compaction of unit sized one–dimensional tasks over segmentable buses.
- A heuristic for the ordered compaction of arbitrarily long one–dimensional tasks over segmentable buses that bounds the maximum delay to a task.
- An assessment and comparison of the costs and benefits of FPGA task rearrangement methods for varying configuration delays when tasks are reloaded from off–chip and copied on–chip.

The use of task rearrangement to reduce fragmentation was first investigated for multi–stage interconnection networks as part of the PASM project, with efforts directed at modelling and performing the movement of a single task [57, 58]. Several task rearrangement methods that moved some or all of the executing tasks were subsequently proposed for MIMD hypercubes, with efforts directed at devising optimal edge–disjoint migration algorithms (see, for example, [13, 36]). Algorithms for rearranging the tasks on a star graph were described in [40]. The first results for the mesh, which is a natural model for a generic FPGA architecture, were reported by Youn, Yoo, and Shirazi [70]. Youn et al. proposed a complete, parallel task rearrangement method that moved all of the tasks executing on the mesh. They also proposed a sequential method for performing partial rearrangements. Their results suggest that modest performance improvements are possible with the complete method, but that the partial method is not particularly beneficial, especially as communication overheads rise. In contrast to Youn's findings, this thesis indicates that, depending upon the operating conditions, substantial performance benefits are possible with partial rearrangements irrespective of the communication overhead.

There are alternative approaches to overcoming fragmentation which are not discussed in this thesis. They include scaling tasks down or up in size to influence task execution times or to fit them into available blocks, and partitioning the waiting task at run–time into appropriately sized subtasks that can be placed immediately.

1.4 Outline

Chapter 2 presents the hardware and operating system models upon which this thesis is based. The chapter first describes the space–shared FPGA and segmentable bus models. Next, it outlines the allocation of tasks with partial rearrangement. Then, it details the three models for moving tasks considered: by reloading, by copying over nearest neighbour links, and by copying over segmentable buses.

Chapter 3 describes two proposals for identifying and scheduling task rearrangements on FPGAs when tasks can only be loaded one after another from off the chip. The chapter begins with a statement of the FPGA task rearrangement problem and the solution goals. It then describes the proposals for identifying partial task rearrangements on conventional FPGAs and describes how optimal allocation sites for the next pending task can be found. After proving the NP–hardness of task rearrangement scheduling, the chapter presents polynomial time scheduling methods for each of the rearrangement identification methods. The chapter compares the time complexity and summarizes the empirical performance of the algorithms before concluding with a description of possible improvements.

Chapter 4 describes the use of nearest neighbour links and segmentable buses to move tasks in parallel on the chip. First of all, a method for performing ordered compactions, described in Chapter 3, using nearest neighbour links to move the tasks is described. The remainder of the chapter then presents and analyzes algorithms for the ordered compaction of unit length and arbitrarily sized one–dimensional FPGA tasks over a segmentable bus. The further development of these methods, including the extension to two dimensions, is considered at the conclusion of the chapter.

Chapter 5 reports on an empirical study into the impact of the proposed methods on simulated FPGAs. To begin with, the effectiveness of the arbitrary task rearrangement scheduling algorithm is examined. Then, the effects of varying the system load, the configuration delay, and the task size distribution when tasks are reloaded are investigated. Finally, the effect of varying the link delay when tasks are

moved over nearest neighbour links is assessed To conclude the chapter, the results are compared with those of Youn et al. for the mesh, and areas for improvement are identified.

Chapter 6 concludes the thesis with a review of the results and their significance as well as a description of the unresolved issues and directions for further study.

1.5 Related Publications

Chapters 3 and 5 contain material that has appeared in [20] and [21]. Chapters 4 and 5 contain material reported in [19] and [23]. A summary of the results will appear in [22].

Chapter 2

Architectural Models

This chapter describes the hardware platforms and operating system environment for the algorithms in this thesis. First, it describes the partitionable array models considered. Next, it presents the multi–tasking operating environment that supports task rearrangements. Finally, it details the various models investigated for moving tasks.

The space–shared FPGA model is a multi–tasking model in which the programmable resources are partitioned among multiple independent tasks. Ideally, each task executes in its own partition as if it were the sole task executing on an FPGA just large enough for its own needs. Each task is served by its own controller, which has the responsibilities of loading, interrupting, and reconfiguring interdependent subtasks, and of directing the flow of communications to and from the task.

The first section of the chapter presents a high–level model for a partitionable Single Instruction, Multiple Data (SIMD) machine that forms the basis for the hardware architectures discussed in this thesis. Because of its generality, this model is also suitable for describing partitionable FPGAs. Indeed the fundamental similarities between SIMD and FPGA computing models are touched upon in this and the second section, which describes the dynamically reconfigurable FPGA, reconfigurable mesh, and segmentable bus models used in the thesis.

The algorithms of Chapter 3, which use reloading to move tasks, are designed for the dynamically reconfigurable FPGA model. In Chapter 4, two methods for moving tasks on–chip are considered. Since nearest neighbour links are commonly found on current FPGA architectures, the dynamically reconfigurable FPGA model applies when these are used to rearrange the tasks. However, to move tasks over

buses, dynamic switching is needed. Since this is not a feature of currently available FPGAs, a segmentable bus model which incorporates the distinctive features of the reconfigurable mesh is developed.

The third section outlines the multi–tasking operating system environment that allows tasks to be rearranged. The description focuses on the task allocation function with particular emphasis on allocation with task rearrangement but is necessarily high–level so as to remain implementation independent.

Detailed models for moving tasks by reloading them and by copying them over nearest neighbour links and buses are provided in the final section of the chapter.

2.1 A Space–Shared Multi–Tasking Model

The Single Instruction, Multiple Data (SIMD) model of parallel computation is characterized by a large number of simple compute processors all executing, at the same time, the same instruction, which is broadcast by a centralized control processor. The model achieves high performance through massive data parallelism.

Of the many network topologies that have been investigated, the SIMD array has gained supremacy, not just for its design simplicity, but also because many real world problems naturally map to it. Indeed, the first parallel machine built, the ILLIAC IV [6], was a SIMD mesh. Like ILLIAC, the MPP [8] was a bit–slice mesh, and the MasPar MP–1 [49], a word–width processor array, achieved commercial success.

As early as 1968 [6], designers of massively parallel computers recognized that effective use of the processing elements in such machines can be achieved through the ability to partition a large–scale multiprocessor into independent SIMD subsystems. The ILLIAC IV supported partitioning at the processor and array level. The 64 bit–wide processing elements were able to be split into two 32 bit–wide or eight 8 bit–wide elements, and the array could be split into two or four independently controlled, equally sized machines. The Connection Machine [66] was also partitionable into one, two, or four machines of equal size, each of them controlled by its own front end processor. However, this coarse type of partitioning is static and limiting.

The partitionable multiple–SIMD model, first proposed by Nutt in 1977 [50], allows for the set of compute processors to be shared by multiple control units. Systems adhering to this model make effective use of the compute processors by

adjusting the sizes of processor partitions to the sizes of the tasks, thereby allowing several tasks to be executed in parallel. Possible implementations were subsequently investigated by the MOPAC [41], PASM [60], and GPA [12] projects.

An overview of this model is illustrated in Figure 2.1. The compute resource consists of N interconnected processing elements (PEs) that are controlled by a set of M control processors (CPs). The CPs in turn are under the global control of a host. The host orchestrates the operation of the CPs, each of which broadcasts instructions for the PEs under its control over the CP–PE interconnection network. It is assumed that partitions involve non–overlapping sets of consecutive or contiguous processors and behave like a dedicated machine of the corresponding size [59].

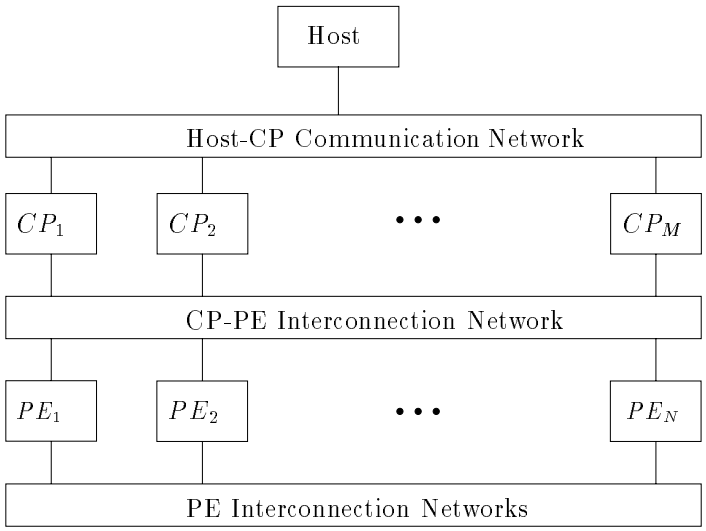


Figure 2.1: Nutt's multi–SIMD model of parallel computation.

2.2 Compute Resource Models

2.2.1 Dynamically Reconfigurable FPGA Model

Bolotski, DeHon and Knight describe a model of computational arrays that unifies SIMD arrays and FPGAs [10]. Their model consists of a grid of array elements with interconnection resources linking array elements together. Each array element performs a simple function on some state and some inputs from the array and either updates its own state to record the result of its computation or shares the results with other elements in the array. An instruction specifies the computation performed

by each array element. The instruction is also used to specify communications and state manipulation. In their model, the transform from input values to outputs is modelled as a look up table addressed by the input value. The instruction is modelled as either the programming of the look up table or as additional inputs. Ideally, each array element should be able to perform a different instruction each cycle. However, the instruction bandwidth that would be needed for reasonably sized arrays would be too large. Both FPGAs and SIMD therefore weaken the model. While SIMD arrays compromise in the spatial variation of instructions to allow a high rate of instruction dispatch, FPGAs compromise in the rate of instruction dispatch to allow the instructions to vary spatially through the array. FPGAs achieve high performance by pipelining data through multiple stages, each of them performing different logic functions simultaneously.

In broad terms, an FPGA task is a sequential or combinational circuit that is laid out in space rather than time. The logic functions of an FPGA circuit are performed by configurable logic cells that are interconnected through wires and programmable routing switches. A "program" not only instructs the cells which functions to perform but also determines how they are to be interconnected by setting the routing switches. The circuit is configured before the data arrives and remains configured until a new configuration is loaded.

The cells of common dynamically reconfigurable FPGAs [2, 69] are laid out in a two–dimensional grid and are usually directly connected with their neighbours to the north, south, east, and west via nearest neighbour links. In addition, FPGAs usually incorporate some network of bus segments for fast communications over longer distances. Routing switches are used to connect cells to bus segments and to interconnect bus segments for turns or longer paths.

Definition 1 *A space–shared FPGA of width* W *and height* H *is a two–dimensional grid of configurable cells and routing resources denoted* $G^2[(1,1),(W,H)]$ *with bottom–left cell labelled* $(1,1)$ *and top–right cell labelled* (W,H).

Current FPGA cells typically consist of no more than a few logic gates, or a small look up table, a flip–flop, and some multiplexors for configuring the cell function. Cells typically operate on two or four bits of input and produce a single output bit. The output of the cell is a boolean combination of the inputs, flip–flop contents, and constants.

Configuration involves loading the look up table and/or memory selectors for the multiplexors associated with each cell to select the cell's function. Several bytes of configuration data per cell are serially loaded as a configuration bit–stream via pins on the periphery of the chip. The configuration of the routing switches is also determined by the program.

With current technology, the time to configure a cell is an order of magnitude greater than the signal delay through it or along a wire. The signal delay along interconnections composed of several bus segments is approximately linear in the number of switches along the path from the source cell to the destination cell [39].

It is assumed that an FPGA task and the used routing resources surrounding its perimeter can be modelled as a rectangular subarray of arbitrary yet specified dimensions. Some internal fragmentation therefore results when task designs cannot be optimized to a rectangular shape. The size of a task is assumed fixed for the duration of its execution.

Definition 2 *The FPGA task* $t[l_1, l_2]$ *with* $l_1, l_2 \in Z^+$ *requires an array of size* $l_1 \times l_2$ *to execute.*

Tasks are assumed to be independent. However, when a task is decomposed into several reconfigurable subtasks, they are allocated to the largest bounding box required throughout the task's instantiation. In this way, routing conflicts and interference with other tasks are avoided.

Tasks are assumed to be deadline–free and to have unknown service periods. However, it is possible to check whether or not tasks for which service periods are known can be rearranged without exceeding deadlines.

No limit is placed upon the number of tasks that can execute simultaneously. To support multi–tasking, the FPGA should be able to support multiple simultaneous I/O streams. The idealized model considered in this thesis assumes any number of I/O streams can be supported without slowdown. The relatively small and fixed number of I/O pins on FPGA packaging necessitates the use of time multiplexing for I/O. Techniques for doing so are being investigated by MIT's virtual wires project [3].

Each task is allocated a subarray of the required size within a larger partitionable array. Usually a subarray will simply be referred to as an array as well.

Definition 3 *The orientation* $\mathrm{or}(t) = (x, y)$ *of a task* t *specifies the number of cell*

columns x and rows y allocated to the task from the array. Given the orientation of a task, its width $w(\text{or}(t)) = x$ and height $h(\text{or}(t)) = y$ are known.

Tasks may be rotated and relocated. Task $t[l_1, l_2]$ may be oriented such that $\text{or}(t) = (l_1, l_2)$ or such that $\text{or}(t) = (l_2, l_1)$. If $\text{or}(t) = (l_1, l_2)$, then it may be allocated to any array $G^2[(x, y), (x + l_1 - 1, y + l_2 - 1)]$ where $1 \leq x \leq W - l_1 + 1$ and $1 \leq y \leq H - l_2 + 1$. If tasks make use of hierarchical routing networks, then they might not in practice be relocated anywhere. The FPGA abstraction assumes the routing interface to all cells is identical.

The dynamically reconfigurable FPGA model assumes the I/O architecture permits random access to the configuration memory of a single logic cell or routing switch in a single step. Moreover, it is assumed that a cell or switch can be configured in a constant amount of time.

It is assumed that the time needed to configure a subarray

$$t_{\text{conf}}(G^2[(x_1, y_1), (x_2, y_2)]) = CD \times (x_2 - x_1 + 1)(y_2 - y_1 + 1) \tag{2.1}$$

is proportional to the configuration delay per cell CD and the size of the subarray since, at worst, cells are configured sequentially. Since the delay properties of commercially available chips are isotropic and homogeneous, CD is assumed to be constant, i.e., the time needed to configure a task and route I/O to it is independent of the task's location and orientation.

The logic cells are assumed to have storage for a single task context. Configuration memory contents are assumed to be overwritten when a cell is configured.

2.2.2 Reconfigurable Mesh Model

The reconfigurable mesh [1, 38, 47] is a more traditional SIMD model of computation that is also based on a two–dimensional grid of processors architecture (see Figure 2.2).

The model is distinguished by its reconfigurable bus system. Internal to each processing node is a set of locally controlled short circuit switches that allow the interprocessor wires to be connected together to form a communications bus. A different connection configuration can be established during each machine cycle, and all processors participating in a bus configuration have access to the data available on it. The model is said to display connection autonomy because the connection configuration of a PE can be set according to local state information. The 15 possible connection configurations are depicted in Figure 2.3.

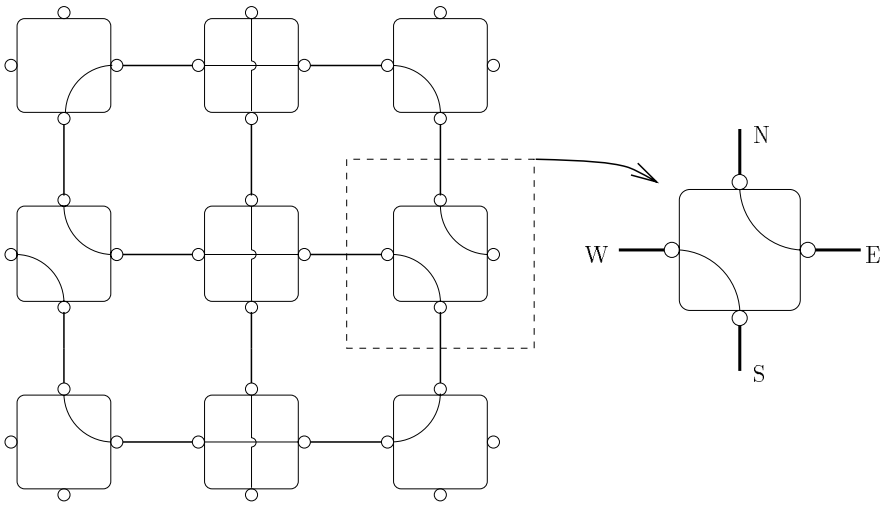


Figure 2.2: A reconfigurable mesh of size 3×3.

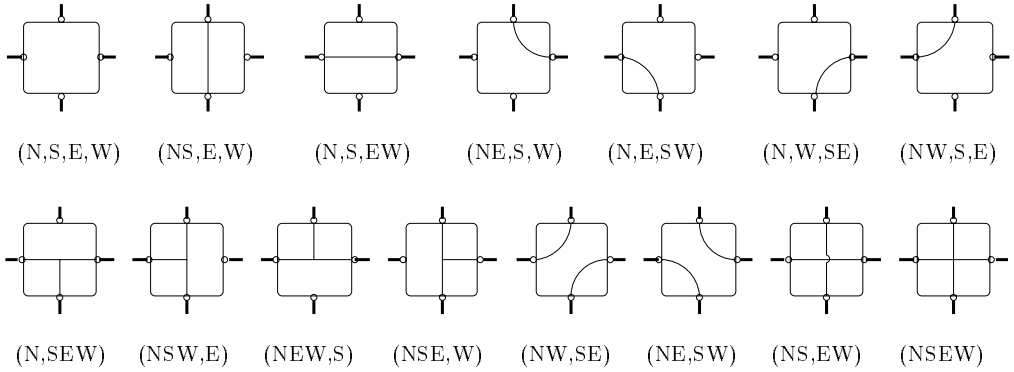


Figure 2.3: Reconfigurable mesh connection configurations.

Reconfigurable mesh processors operate synchronously, in one machine cycle setting a connection configuration, sending (receiving) a datum to (from) each I/O port, and performing an arithmetic, logic or control operation. When a connection is set, signals received by a port are simultaneously available to any port connected to it. For example, if processors connect their northern and southern I/O ports by closing the appropriate switches as in the configuration (NS,E,W), data "broadcast" onto the "column buses" so formed can be read by all of the processors in a column. The model allows concurrent reading from a bus but requires exclusive writing to the bus. If in a single cycle multiple broadcasts are to be made over multiple buses, those buses are required to be disjoint. The model usually assumes the delay along a bus is a constant independent of length. For buses of bounded length this assumption is reasonable. Techniques for coping with buses of unconstrained length have also been proposed [9, 24].

2.2.3 Segmentable Bus Model

The segmentable bus model used in this thesis adopts the central assumptions of the reconfigurable mesh model while simplifying the hardware considerably. The model makes use of the ability to establish a new bus configuration each cycle and to communicate in one cycle irrespective of distance. The former is not difficult to implement, and the adoption of the latter is motivated by the assumption that the buses formed will not branch and will span the width of the FPGA chip at most. For a review of the computational power of this model see [65].

In Chapter 4, a hybrid model comprising a linear array of FPGA cells as the compute resource and a linear array of switching elements for the configurable interconnect is used to investigate the complexity and power of using segmentable buses to rearrange tasks. See Figure 2.4.

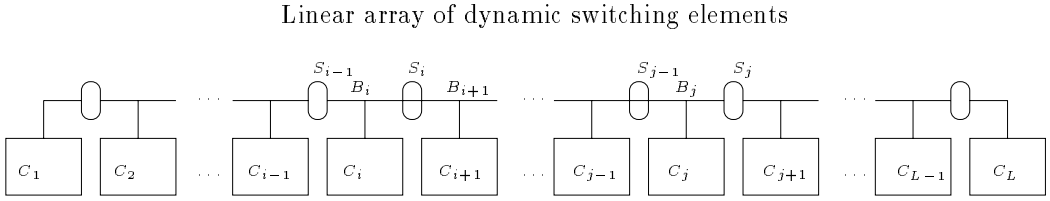


Figure 2.4: The segmentable bus FPGA model.

The one–dimensional grid or linear array of L cells $G^1[1, L]$ is labelled $C_1, \ldots, C_L$ from left to right. Each cell C_i is connected to a bus segment of unit length labelled B_i directly to its north. Between each pair of consecutive bus segments B_i and B_{i+1} there is a short circuit switch labelled S_i which can be opened or closed at the commencement of each communication cycle or step.

Cells C_i and C_j with $i < j$ communicate in a single cycle in the following manner:

1. Switches S_{i-1} and S_j open, and switches $S_i, S_{i+1}, \ldots, S_{j-1}$ close, and
2. C_i and C_j communicate via their northern ports and the bus formed between them.

Opening the switches S_{i-1} and S_j isolates the bus spanning cells C_i and C_j from the rest of the bus system, thereby allowing additional buses to be formed to the left of S_{i-1} and to the right of S_j in the same cycle.

Setting a bus configuration and transferring a cell's contents is assumed to take a single step irrespective of transfer distance. The length of a step is therefore normalized to the longest period needed to establish the switch settings and propagate a cell's contents across the width of the FPGA at most.

Logic cells are assumed to have storage for two task contexts between which they can switch in a single step. Storage for the cell configuration and state that is switched out of context may be accessed from the bus segment attached to the cell while the cell executes a second task context.

2.3 Task Management with Partial Rearrangement

Overall management of tasks is accomplished in the following way. Refer to Figure 2.5 for an overview. Note that the model applies equally well to one–dimensional arrays.

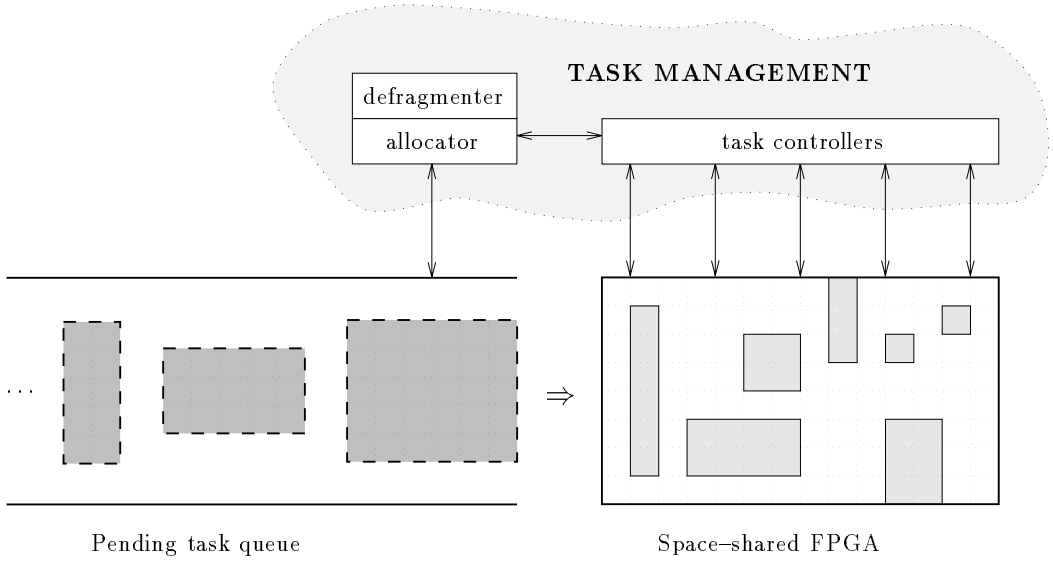


Figure 2.5: An overview of task management with partial rearrangement.

Requests for service are queued in arrival order by a sequential host. A task allocator, executing on the host, attempts to find an allocation site to satisfy the next pending request when it arrives. If the allocator succeeds in finding a suitable site, it associates a controller with the new task and its partition and allows the controller to assume responsibility for loading the task and establishing I/O to it.

When the allocator fails to find a suitable allocation site for the next pending request, it invokes a defragmenter, which determines whether or not the request

can be satisfied with partial rearrangement. If allocation with rearrangement is possible, the defragmenter performs the rearrangement and returns the allocation site thereby created to satisfy the request. If on the other hand allocation with rearrangement is not possible, then all requests are blocked until allocation is attempted once more.

Subsequent allocation attempts are made whenever a task completes and there is a request for service pending. If some executing tasks can be rearranged to accommodate the task, then a schedule for suspending and moving them is computed. The defragmenter then coordinates the partial reconfiguration of the FPGA by signalling individual task controllers to suspend a task's operation, save the task's context, move the task and its context to a new location, and to resume the task's operation.

For the sake of fairness and simplicity, requests for service are processed in first–come first–served (FCFS) order. However, the methods discussed in this thesis do not depend upon the scheduling method. Non–FCFS methods with better performance, such as back–filling [44], could therefore be used.

The task allocator uses bottom–left allocation, which is a first fit method [71]. The bottom–leftmost free block large enough to satisfy the request is allocated to the task. The advantages of the first fit method are that it is simple and that it has complete recognition capability, i.e., it recognizes all possible allocation sites. Many other contiguous allocation schemes have been proposed. For example, the two–dimensional buddy method [43] and frame–sliding [14] are more efficient than first fit, as originally proposed, but they suffer from high fragmentation, and have incomplete recognition capability. The busy–list method [18] is efficient, and it attempts to reduce fragmentation by using a best–fit approach. However, it is considerably more complicated to implement. It should be noted that partial rearrangement can be successfully used with any allocation method.

Definition 4 *Let* $T = \{t_i[l_{1,i}, l_{2,i}] : 1 \leq i \leq n\}$ *be a set of tasks allocated to an FPGA* $G^2[(1,1),(W,H)]$. *The arrangement of tasks* $A(G^2[(1,1),(W,H)]) = \{a(t_i) : t_i \in T\}$ *is the set of non–overlapping orthogonally aligned rectangular allocations* $a(t_i) = G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]$ *in the array* $G^2[(1,1),(W,H)]$. *The allocation for task* t_i *is said to be based at the cell allocated to the bottom–left corner of the task* $(x(\mathrm{bl}(t_i)), y(\mathrm{bl}(t_i)))$ *and to extend to the cell allocated to the top–right corner of the task* $(x(\mathrm{tr}(t_i)), y(\mathrm{tr}(t_i)))$.

2.4 Task Rearrangement

Rearranging the tasks executing on an FPGA requires moving them. Moving a task involves: suspending input to the task and waiting for the results of the last input to appear or waiting for the task to reach a checkpoint; storing register states if necessary; reconfiguring the portion of the FPGA at the task's destination; loading stored register states if necessary; and resuming the supply of input to the task for execution. The problem of rerouting I/O to a task that is moved is not addressed in this thesis.

Since tasks are assumed to be without deadlines, any task is considered preemptable and may therefore be suspended with its inputs being buffered and necessary internal states being latched until the task is resumed. The time needed to wait for the results of an input to appear or for the task to reach a checkpoint is considered to be proportional to the size of the task which in the absence of feedback circuits is the worst case. However, since with current technology the time to configure a cell and associated routing resources is typically an order of magnitude greater than the signal delay of a cell or the latency of a wire, the latency of the design is considered negligible compared with the time needed to configure the task.

A task cannot be moved without some cost. The approach in this thesis is to distinguish between the minimum possible cost of moving a task and the actual cost of moving it. The minimum cost is the time needed to suspend, move and resume the task, which is unavoidable. However, the actual cost needs to account for the time a task is suspended while other tasks are being moved. The difference between the actual and minimum costs represents a scheduling delay that is to be minimized for all tasks.

2.4.1 Moving Tasks by Reloading

In Chapter 3, the effectiveness of reconfiguring the destination region of a task by reloading the configuration stream with a new offset is investigated. This approach naturally re–incurs the cost of configuring the task, given above in Equation 2.1, but is applicable to any dynamically reconfigurable device.

Moving tasks by reloading them is inherently sequential. Tasks are reloaded one after another, cell by cell. In Chapter 4, techniques for using on–chip resources to overcome this bottleneck are developed. It should also be noted that if I/O to tasks is performed using direct addressing, then tasks not being moved may be

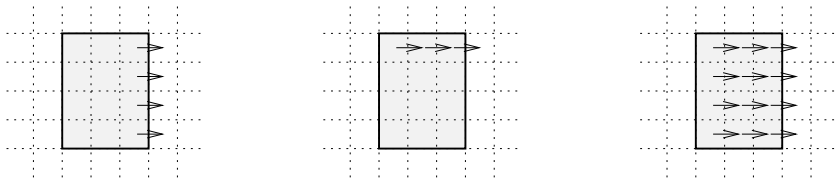
delayed by the configuration stream of tasks being moved.

2.4.2 Moving Tasks over Nearest Neighbour Links

The time required to move a task over the links connecting neighbouring cells is significantly less than that required to reload it. Assume a task is to be moved d cells to the right along the rows of the FPGA and that there are no tasks in its way to impede its movement. See Figure 2.6 for an illustration.

Definition 5 *The configuration bits for a cell and the cell's state are collectively referred to as a task element.*

All task elements in the rightmost column of task cells can be moved simultaneously by writing them onto port E and having them read by the neighbouring cells from port W. Moreover, the task elements in each row of task cells can be moved simultaneously as well by having all cells write their task element to port E, and having all cells that are to receive a task element read it from port W. The task elements of each row are said to be pipelined to the right. The task can thus be moved one column of cells to the right in each move cycle, and d cycles in total are needed to move the task to its destination.



(a) moving a column of cells at a time (b) moving a row of cells at a time (c) moving all cells at once

Figure 2.6: Moving a task over nearest neighbour links.

To move a task it must be halted. Cells then repeatedly send their task elements to their neighbours until the task reaches its destination. All task elements reach their destination in the same cycle. Once the task reaches its destination it is resumed again.

The time to move a task is proportional to the distance the task has to move. The actual time spent moving depends upon the time needed to transfer a task element to the neighbouring cell, which depends upon the amount of configuration and state information that needs to be transferred, as well as the bandwidth of the link. Current FPGAs allow configuration bits to be loaded one byte or word

at a time while their nearest neighbour links are usually one bit wide. However, transferring configuration data over a nearest neighbour link may be faster than loading it from off–chip because absolute addresses may not need to be decoded and wire lengths are shorter. The time spent moving a task also needs to take into account the time required to halt and resume the task. This task or design latency is considered negligible compared with the time needed to load the task. Assuming tasks are not moved too often, this cost can therefore be ignored.

2.4.3 Moving Tasks over a Segmentable Bus

Moving a linear array task over a segmentable bus involves switching the task out of context and configuring the buses needed to move the task elements over a sequence of steps. During each step, the source (destination) cells of a move then write (read) their task elements to (from) the bus segments to which they are connected. When all the elements of a task have reached their destination, the task is switched back into context and task execution is resumed.

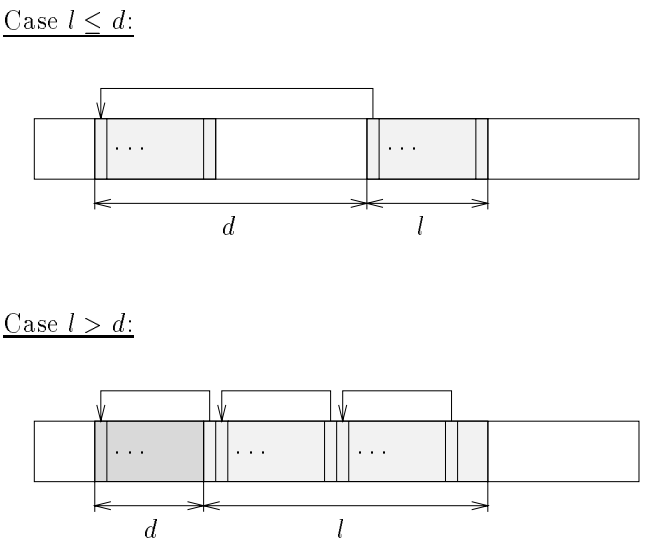


Figure 2.7: Moving a linear array task over a segmentable bus.

The time required to move the task is derived in the following lemma.

Lemma 1 *A one–dimensional task of length l can be moved d cells to the left in $\Theta(\min(l, d))$ steps over a segmentable bus. This is optimal.*

Proof: Refer to Figure 2.7. When $l \leq d$, all task elements must be moved over and past the bus segment connected to the leftmost cell occupied by the task. Since

the buses used to move task elements are required to be disjoint, at most one task element can be moved per cycle or step. At least l steps are therefore required to move the task. Moving them one at a time achieves the lower bound.

When $l > d$, d task elements need to be moved to the left of the leftmost cell occupied by the task. The need to form disjoint buses implies d steps are needed for these movements. Furthermore, the cell initially occupied by the leftmost task element cannot be the destination of a task element move during one of these cycles. Therefore $d + 1$ steps at least are required to complete the task move. To complete the move in a minimum number of steps, task elements whose index mod $(d + 1)$ is equal to i are moved in parallel during step i. This movement maximizes the use of the segmentable bus and is therefore optimal. ∎

Chapter 3

Partial FPGA Rearrangement

This chapter proposes and evaluates methods by which feasible rearrangements can be identified and scheduled when tasks are moved by reloading them from off–chip memory.

The purpose of partial FPGA rearrangement is to allocate waiting tasks as quickly as possible so as to reduce task response time. This benefit to the waiting task does not come without some cost to the executing tasks since they must be interrupted in order to be moved. It is therefore desirable to delay as few executing tasks as possible and to minimize the maximum amount any single task is delayed — the net effect of the delays to executing tasks should not outweigh the net benefit to the waiting tasks. The current FPGA model, described in Chapter 2, moves tasks by reloading their configuration bit streams with new offsets. This approach limits the time needed to complete the rearrangement since the time to move a task is assumed to be proportional to its area, and the tasks that need to be moved must be reloaded one after another. The time to complete a rearrangement is therefore proportional to the total area of the tasks moved. Since the rate at which tasks can be allocated is limited by the rate at which allocations can be found or rearrangements can be performed, it is desirable to complete rearrangements as quickly as possible. These factors contribute to the formulation of the FPGA rearrangement problem as follows.

FPGA REARRANGEMENT PROBLEM

INPUT: A set of executing tasks $T = \{t_1, \ldots, t_n\}$, an arrangement $A(G^2[(1,1),(W, H)]) = \{a(t_i) : t_i \in T\}$ of the executing tasks, and a waiting task $t_{n+1}[l_{1,n+1}, l_{2,n+1}]$ that cannot be allocated to the array without overlapping the allocation of some other task in T.

OUTPUT: A new arrangement $A'(G^2[(1,1),(W,H)]) = \{a'(t_i) : 1 \leq i \leq n+1\}$ of

the tasks, if possible, and a schedule $p : T \rightarrow Z_0^+$ for moving the tasks in $\{t_i : a(t_i) \neq a'(t_i)\}$ such that:

1. the allocation for t_{n+1} is freed of executing tasks in minimum time,
2. the maximum delay to executing tasks that must be moved is minimized, and
3. the time needed to complete the rearrangement is minimized.

The statement of the problem suggests two subproblems are to be solved. First, a new arrangement of the executing tasks that accommodates the unallocated or waiting task is needed. Second, a schedule for moving the tasks from their initial to their final allocations is to be found. The work involved in solving these problems represents an overhead to the system. An additional requirement therefore, is to find efficient solutions.

In Section 3.1 it is shown that the problem of identifying a rearrangement of the executing tasks that accommodates the waiting task is NP–complete. A reasonably quick approximate approach to finding suitable or feasible rearrangements is therefore needed. Two heuristic solutions are presented in this chapter. The first is based on the idea of repacking the tasks in a suitable local area of the array using a two–dimensional strip–packing algorithm. Local area candidates are identified by means of a quadtree decomposition of the free cells. Since strip–packing algorithms are approximation algorithms, some feasible local areas remain unidentifiable by this method. The second method constrains the range of task movements and thus the number of initial arrangements that can be successfully rearranged. This second method identifies all possible sites for the waiting task that moves a subset of the executing tasks closer together in one direction.

Each feasible rearrangement must be evaluated for the time needed to free the allocation site, for the time needed to complete the rearrangement, and for the maximum delay to moving tasks. The problem of identifying the best rearrangement is thus linked by feedback to the problem of scheduling the rearrangement of the tasks. While the geometric problem of identifying possible rearrangements of the tasks is independent of the hardware model, the choice of optimal rearrangement varies according to the different scheduling techniques and outcomes possible with different technologies. In this chapter, task movements are constrained to be performed sequentially by reloading tasks from off the chip.

In Section 3.2, scheduling the rearrangement of tasks on the FPGA so as to minimize the delays to the waiting and moving tasks is shown to be NP–complete. A greedy heuristic that minimizes the increase in delay to tasks with each sequencing choice is thus developed. The time complexity and performance of this heuristic depends upon the depth to which the state–space is explored before making a choice. Methods for on–chip task movements and their effect on rearrangement scheduling are examined in the next chapter.

Section 3.3 presents a comparison of the time complexity of partial rearrangement heuristics and summarizes the results of the experimental investigation of allocation performance reported on in Chapter 5.

This chapter concludes with a chapter summary and avenues for improvement in Section 3.4.

3.1 Identifying Feasible Rearrangements

A feasible rearrangement of the executing tasks is a new arrangement of the executing tasks that allows the waiting task to be allocated as well. Although it is assumed that tasks may be rotated before allocation, the rearrangements considered in this thesis do not rotate the executing tasks.

In this section, it is first shown that the problem of identifying feasible rearrangements is NP–complete. Two heuristics are subsequently presented to overcome this intractability. Local repacking hierarchically decomposes and assesses the array for suitable subarrays of tasks that, when repacked using known methods, may accommodate the next waiting task. This method may translate tasks in several directions. Ordered compaction, on the other hand, makes use of a novel scanning procedure to identify subsets of tasks that can be squeezed together in one direction to make room for the waiting task.

3.1.1 Identifying Feasible Rearrangements is NP–Complete

In [42], Li and Cheng show that it is NP–complete to decide the RECTANGLE PACKABILITY problem, which is to determine whether or not a set of oriented rectangles can be orthogonally packed without overlap into a larger containing rectangle. Their proof was by reduction from the PARTITION problem [29]: the sizes of the elements of a given PARTITION instance determine the widths of corresponding rectangles having height 1/4. These can be packed into an ar-

ray of width one half the total size of the PARTITION elements and height 1/2 in polynomial time if and only if P = NP. The following theorem thus follows.

Theorem 1 *([42]) RECTANGLE PACKABILITY is NP–complete.*

Corollary 1 *REARRANGEMENT FEASIBILITY, the problem of deciding whether a set of executing tasks can be feasibly rearranged to accommodate the next waiting task is NP–complete.*

Proof: By equivalence with RECTANGLE PACKABILITY. A procedure for deciding RECTANGLE PACKABILITY can decide whether or not the set of executing tasks taken together with the next waiting task can be packed into the array. Similarly, an algorithm for deciding REARRANGEMENT FEASIBILITY can be used iteratively to determine RECTANGLE PACKABILITY. The problems are therefore computationally equivalent. ∎

Since the problem of deciding REARRANGEMENT FEASIBILITY is NP–complete, it is unlikely to have a polynomial time solution. The corresponding optimization problem, that of finding a feasible rearrangement, is therefore also unlikely to be easy. Consequently, heuristic solutions are sought. The remainder of this section presents two such solutions.

3.1.2 Local Repacking

The idea behind local area repacking is to repack the tasks initially allocated to some rectangular region of the array so as to accommodate the waiting task within the subarray as well. A hierarchical decomposition of the array known as a free area tree is used to keep track of the number of free cells within each subarray. In so doing, regions that contain sufficient free area to accommodate the waiting task can be quickly identified, and rearrangements of the tasks they contain can be attempted. Two–dimensional bin packing algorithms with good performance bounds are used for this last step: the tasks, viewed as rectangles, are packed from scratch into an infinitely long strip whose width is determined by the length of one side of the subarray. If the tasks are packed using total height less than the length of the other side of the subarray, then the rearrangement is feasible, and its cost is then assessed.

Following a definition of a free area tree and a description of its use, algorithms for constructing and searching it are presented. The subsection concludes

with a brief review of known strip–packing algorithms available for use by the method.

3.1.2.1 Free Area Trees

A free area tree is a type of quadtree [56, 72] that need not necessarily be defined over a square grid and whose leaves may have just one rather than three siblings. Each node of the tree, which represents a portion of the array, stores the number of free cells contained within the region and pointers to its children. If the array covered by a node is completely free, or if it is entirely allocated to a single task, then it is not further decomposed. Otherwise, the array represented by the node is partitioned evenly into two or four disjoint subarrays, depending upon its size, and represented by child nodes. A formal definition of a free area tree follows.

Definition 6 *Array* $G_1^2[(x_1, y_1), (x_2, y_2)]$ *is said to intersect array* $G_2^2[(x_3, y_3), (x_4, y_4)]$ *iff* $(x_1 \leq x_4)$ *and* $(x_3 \leq x_2)$ *and* $(y_1 \leq y_4)$ *and* $(y_3 \leq y_2)$. *The intersection of arrays*

$$G_1^2[(x_1, y_1), (x_2, y_2)] \cap G_2^2[(x_3, y_3), (x_4, y_4)] = G^2[(\max(x_1, x_3), \max(y_1, y_3)), (\min(x_2, x_4), \min(y_2, y_4))]$$

if $(x_1 \leq x_4)$ *and* $(x_3 \leq x_2)$ *and* $(y_1 \leq y_4)$ *and* $(y_3 \leq y_2)$, *otherwise it does not exist and is defined to be* $\emptyset$.

Definition 7 *The area of the array* $G^2[(x_1, y_1), (x_2, y_2)]$ *is*

$$\mathrm{ar}(G_1^2[(x_1, y_1), (x_2, y_2)]) = (x_2 - x_1 + 1)(y_2 - y_1 + 1).$$

By definition, $\mathrm{ar}(\emptyset) = 0$.

The free area of the array $G^2[(x_1, y_1), (x_2, y_2)]$ *is the number of unallocated cells* $\mathrm{fa}(G^2[(x_1, y_1), (x_2, y_2)])$ *in the array.*

For the arrangement of tasks $A(G^2[(1, 1), (W, H)]) = \{a(t_i) : 1 \leq i \leq n\}$ *the free area*

$$\mathrm{fa}(G^2[(x_1, y_1), (x_2, y_2)]) = \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)]) - \sum_{i=1}^{i=n} \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)] \cap G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]).$$

Definition 8 *The predicate* $\mathcal{P}(G^2[(x_1, y_1), (x_2, y_2)])$ *is defined to be true if some task* t_i *exists such that some but not all of the cells in* $G^2[(x_1, y_1), (x_2, y_2)]$ *are allocated to*

it, i.e., if for some allocated task $t_i, 0 < \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)] \cap G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]) < \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)])$.

Definition 9 *(After [72]) The free area tree* $F[(x_1, y_1), (x_2, y_2)]$ *covering* $G^2[(x_1, y_1), (x_2, y_2)]$ *is defined recursively as follows:*

1. $F[(x, y), (x, y)]$ *is a leaf node.*

2. $F[(x, y_1), (x, y_2)]$ *with* $y_1 < y_2$ *is a node.*
   If $\mathcal{P}(G^2[(x, y_1), (x, y_2)])$, *then* $F[(x, y_1), (x, y_2)]$ *has two children:*

   (a) $F[(x, y_1), (x, \lfloor (y_1 + y_2)/2 \rfloor)]$, *and*

   (b) $F[(x, \lfloor (y_1 + y_2)/2 \rfloor + 1), (x, y_2)]$.

3. $F[(x_1, y), (x_2, y)]$ *with* $x_1 < x_2$ *is a node.*
   If $\mathcal{P}(G^2[(x_1, y), (x_2, y)])$, *then* $F[(x_1, y), (x_2, y)]$ *has two children:*

   (a) $F[(x_1, y), (\lfloor (x_1 + x_2)/2 \rfloor, y)]$, *and*

   (b) $F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, y), (x_2, y)]$.

4. $F[(x_1, y_1), (x_2, y_2)]$ *with* $x_1 < x_2$ *and* $y_1 < y_2$ *is a node.*
   If $\mathcal{P}(G^2[(x_1, y_1), (x_2, y_2)])$, *then* $F[(x_1, y_1), (x_2, y_2)]$ *has four children:*

   (a) $F[(x_1, y_1), (\lfloor (x_1 + x_2)/2 \rfloor, \lfloor (y_1 + y_2)/2 \rfloor)]$,

   (b) $F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, y_1), (x_2, \lfloor (y_1 + y_2)/2 \rfloor)]$,

   (c) $F[(x_1, \lfloor (y_1 + y_2)/2 \rfloor + 1), (\lfloor (x_1 + x_2)/2 \rfloor, y_2)]$, *and*

   (d) $F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, \lfloor (y_1 + y_2)/2 \rfloor + 1), (x_2, y_2)]$.

Figure 3.1(a) depicts the arrangement of a pair of tasks on a rectangular array. The array is partitioned to show the regions delimiting the extent of the leaf nodes in the free area tree representation of the arrangement. The free area tree corresponding to the arrangement of Figure 3.1(a) is illustrated in Figure 3.1(b). The left to right order of nodes on each level corresponds to the order in which the children of a node are listed in Definition 9.

When invoked, the local repacking method commences by building the free area tree for an arrangement of tasks on the array. Next the tree is searched for nodes that contain more free cells than are needed by the waiting task. For each such node, a repacking of the tasks allocated to the array covered by the node is attempted. These tasks are found in linear time by checking for intersections with

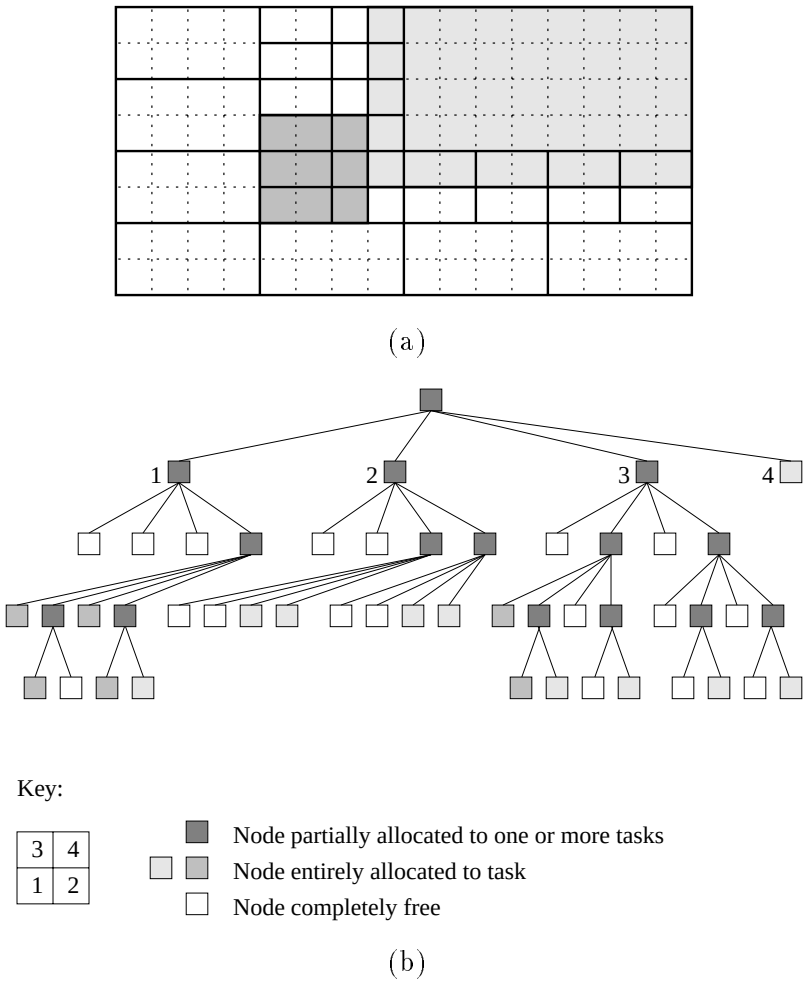


Figure 3.1: (a) The arrangement of a pair of tasks on an array with free area tree leaves marked. (b) The free area tree for the arrangement of (a).

the node's array. If the new arrangement accommodates the waiting task within the array covered by the node as well, then the rearrangement of the tasks to achieve the packing can be scheduled in order to evaluate its optimality.

Tasks which only partially intersect the array covered by a node need to be handled in some way. Should they be included in the packing, moved elsewhere, or left where they are to be packed around? The approach adopted in this work is to attempt to repack these tasks completely into the rectangular array covered by the node as well. This approach avoids further searching and avoids the complexity of packing into arbitrary rectilinear polygons. At each node, therefore, the area available for the waiting task needs to account for the total area of tasks that are only partially covered by the region.

Definition 10 *If the allocation for task t_i intersects the array $G^2[(x_1, y_1), (x_2, y_2)]$, then the uncovered area*

$$\begin{aligned}\mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)]) \;=\; & \mathrm{ar}(G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]) - \\ & \mathrm{ar}(G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)] \cap G^2[(x_1, y_1), (x_2, y_2)]).\end{aligned}$$

Otherwise, $\mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)]) = 0$

Definition 11 *The attached area*

$$\mathrm{aa}(F[(x_1, y_1), (x_2, y_2)]) = \sum_{i=1}^{i=n} \mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)])$$

is the total number of cells allocated to the uncovered portion of tasks that are partially intersected by the free area tree node $F[(x_1, y_1), (x_2, y_2)]$.

If the free area less the attached area at a node exceeds the area of the waiting task, then a packing into the array covered by the node of all the tasks intersected by it and the waiting task is attempted.

3.1.2.2 Building the Free Area Tree

The free area tree is expanded iteratively by inserting each of the executing tasks into an initially empty root[1]. The procedure *InsertTaskIntoFAT* of Figure 3.2 updates the free and attached area for the current node and task and recurses with those children that are partially intersected by the task. It expands the tree by creating the children that don't already exist.

Except for step 4(b), each step requires constant time. Since the cost of step 4(b) can be attributed to the descendents of the node input to the procedure, the time spent updating each node in the tree per invocation is thus constant.

Dyer has analyzed the size of the quadtree representation of square objects in square images [26]. He showed that $O(2^{p+2} - p + q)$ nodes at worst are needed to represent a square of size $2^p \times 2^p$ in an image of size $2^q \times 2^q$. This expression accounts for the perimeter of the square, the logarithm of its diameter, and the height of the root of the tree above the expanded subtree covering the object. If $m = \max(W, H)$, then it follows that $O(m)$ nodes of the free area tree are updated per task insertion since no task can be larger than the array. For n tasks therefore, $O(mn)$ time is needed to build the free area tree. The worst case is attained by

[1] An empty node has free area set to the area of the array covered by the node and attached area set to zero.

Procedure **InsertTaskIntoFAT**

Input A pointer to a node in a free area tree and a pointer to the descriptor of a task that is to be inserted into the tree.

Output A free area tree that has been expanded or modified to account for the task inserted into it.

begin

1. Compute the area of intersection (AI) between the node array and the task.
2. Update the free area for the node.
3. If the area of the task is greater than AI (the node does not wholly contain the task), then
   (a) Update the attached area for the node.
4. If the area of the node is greater than AI (the node is not completely allocated to the task), then
   (a) If the children of the node have not yet been created, then
      i. Create child nodes with free area initialized to the child's area and attached area set to zero.
   (b) For each child that intersects the task:
      i. Recurse with the task descriptor and a pointer to the child node.

end

Figure 3.2: Procedure *InsertTaskIntoFAT*.

any arrangement of tasks that occupy entire rows of a square array, for example, $A(G^2[(1,1)(W,W)]) = \{a(t_i) = G^2[(1,i),(W,i)] : 1 \leq i \leq n \leq W\}$.

It would be possible to reduce the $O(mn)$ time required to build the free area tree if it were updated as allocations, deallocations, and rearrangements occurred. Insertions or deletions into the free area tree require $O(m)$ time. However, a local repacking can potentially rearrange all of the tasks executing on the array, thereby entirely changing the structure of the tree. Nevertheless, rebuilding the tree after each rearrangement would represents a saving over building it each time the feasibility of rearrangement were to be checked.

3.1.2.3 Searching the Free Area Tree

It is desirable that the free area tree be searched in some way that allows promising regions to be discovered early in the search. Ideally the region that is known to cost least to repack should be discovered first. Searching the tree breadth–first allows schedules affecting successively fewer tasks to be discovered and allows the search to be abandoned at a time when the marginal benefit of finding arrangements with lower allocation and execution delays is offset by the growing allocation delay due to the search. A "deepest layer first" search examines those nodes that affect the least number of tasks but have the least chance of accommodating the waiting task first of all. An ideal search therefore starts somewhat higher in the tree and works its way up.

The local repacking method reported upon in Chapter 5 implements a depth–first search of the free area tree and abandons the search once the first feasible arrangement is found.

3.1.2.4 Repacking the Tasks

The search of the free area tree identifies those subarrays that might accommodate the waiting task if the tasks allocated to it are rearranged. Well–known strip–packing algorithms can be used to check whether such an arrangement exists.

Given a set of oriented rectangles and a two–dimensional bin of a given width and unbounded height, the strip–packing problem is to find a minimum height non–overlapping orthogonal packing of the rectangles into the bin [5]. This variant of the two–dimensional bin–packing problem is NP–complete. Much attention has therefore been given to finding polynomial time approximation algorithms, i.e., fast algorithms that come within a constant times the height used by an optimal packing

[15]. For L an arbitrary list of rectangles, let $\mathrm{OPT}(L)$ denote the minimum possible bin height into which the rectangles in L can be packed, and let $\mathrm{A}(L)$ denote the height actually used by a particular algorithm when applied to L [16]. An *absolute performance bound* β for A is a bound of the form

$$\mathrm{A}(L) \leq \beta \mathrm{OPT}(L).$$

On the other hand, in *asymptotic performance bounds* of the form

$$\mathrm{A}(L) \leq \beta \mathrm{OPT}(L) + \gamma,$$

the constant β is intended to characterize the behaviour of the algorithm as the ratio between $\mathrm{OPT}(L)$ and the maximum height rectangle in L goes to infinity. The height of the tallest rectangle is usually normalized to 1, whereby any other choice would only affect the constant γ.

So as to minimize the frequency with which an algorithm fails to find a feasible arrangement when such an arrangement is possible, its absolute performance bound should be as small as possible. The algorithm should also be efficient so as to keep the scheduling component of the allocation delay to a minimum.

Sleator proposed an $O(n \log n)$ time strip–packing algorithm with

$$\mathrm{A}(L) \leq 2\mathrm{OPT}(L) + 0.5h_{\mathrm{tall}}$$

where h_{tall} is the height of the tallest rectangle [62]. Since $h_{\mathrm{tall}} \leq \mathrm{OPT}(L)$, $\mathrm{A}(L) \leq 2.5\mathrm{OPT}(L)$ in the worst case. Asymptotically, however, $\mathrm{A}(L) \Rightarrow 2\mathrm{OPT}(L)$ as $h_{\mathrm{tall}} \Rightarrow 0$. Sleator suggested using his algorithm together with the Split–Fit algorithm of Coffman et al. [16] that has better asymptotic performance. Their algorithm, which has time complexity $O(n \log n)$, is characterized by the equation

$$\mathrm{A}(L) \leq 1.5\mathrm{OPT}(L) + 2$$

when the height of the tallest rectangle is normalized to 1. Should even better asymptotic performance be desired the $O(n \log n)$ time 5/4 algorithm of Baker et al. [4] has the characteristic equation

$$\mathrm{A}(L) \leq \frac{5}{4}\mathrm{OPT}(L) + \frac{53}{8}h_{\mathrm{tall}}.$$

Sleator's suggestion can be extended to include an $O(n \log n)$ stacking algorithm, due to Coffman and Shor [17] that has good asymptotic average case performance.

For their algorithm, the expected height of a unit width strip–packing of rectangles from the uniform model (side lengths in [0,1]) is

$$\mathrm{E}[\mathrm{A}(L)] = n/4 + \Theta(\sqrt{n}),$$

i.e., $\Theta(\sqrt{n})$ space is wasted.

Chapter 5 reports on the effectiveness of using Sleator's algorithm to attempt the repacking. Given the node $F[(x_1, y_1), (x_2, y_2)]$ has been identified as a likely candidate, a "strip" of width $(x_2 - x_1 + 1)$ is tried first. While the orientation of the allocated tasks relative to the width of the strip needs to be preserved to obtain the performance of known strip–packing algorithms, a packing with each orientation of the waiting task is attempted. A feasible rearrangement results if the height of the packing is less than $(y_2 - y_1 + 1)$. Otherwise, the orientation of the strip is flipped so that its width is considered to be $(y_2 - y_1 + 1)$, and a packing within a height of $(x_2 - x_1 + 1)$ is attempted. As mentioned in Section 3.1.2.1, the algorithm attempts to pack tasks that are partially intersected by the subarray into the array as well. If, however, a partially intersected task couldn't possibly fit because one of its sides is too long, the repacking is aborted.

3.1.3 Ordered Compaction

Ordered compaction is a second approach to identifying feasible rearrangements. The ordered compaction heuristic places the waiting task at a favourable location and moves those tasks that initially occupy the allocation site off to one side. Ordered compaction therefore has the effect of moving the subset of the executing tasks that is to be compacted closer together while preserving their relative order. Without loss of generality, ordered compaction to the right is considered. In VLSI circuit compaction this technique has previously been referred to as "ploughing" [51], which is a graphic term for describing the effect — a subset of the executing tasks is ploughed to the right parallel to the rows of the array so that the waiting task can be inserted into the enlarged free space abutting the leftmost tasks moved. Figure 3.3 depicts an example of a right ordered compaction.

In this section, it is shown that in order to minimize the time to complete a compaction it is best to attempt to place the waiting task adjacent to a pair of tasks such that one abuts the allocation site on its left and the other abuts the allocation site below. The number of potential allocation sites worth checking is thus significantly reduced. It is then shown how the feasibility of a site can be

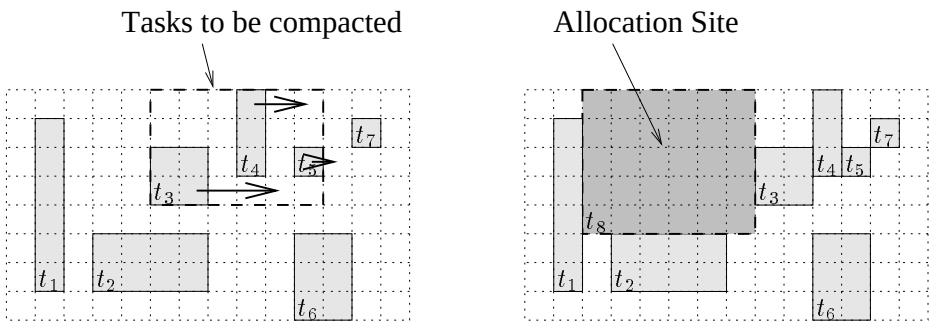


Figure 3.3: An example of a right ordered compaction. The initial arrangement on the left shows the tasks to be compacted so as to allocate a waiting task of size 6×5. The final arrangement on the right indicates the allocation site for the waiting task.

decided by searching a visibility graph defined over the executing tasks. The time needed to free the allocation site of executing tasks is also determined during the decision process.

In the following, the right ordered compaction of tasks for a given orientation of the waiting task is discussed. It is assumed that $\text{or}(t_{n+1}) = (w, h)$. However, it is necessary to consider both orientations of the waiting task and to consider compacting the executing tasks to the left, top, and bottom of the array in order to find the best allocation site. In each case the method is identical, albeit with orientations and directions switched in the natural way. For the remainder of this section the term *compaction* is used to refer to right ordered compaction.

3.1.3.1 Identifying Potential Allocation Sites

The following definitions, which appear illustrated in Figure 3.4, are used to pinpoint the minimum cost locations for placing the base of the waiting task t_{n+1} if it is to be allocated in the neighbourhood of t_i.

The first definition arises from considering where the waiting task can be based were it placed as close as possible on the right of t_i without intersecting it.

Definition 12 *For the waiting task t_{n+1}, assumed to be oriented such that $\text{or}(t_{n+1}) = (w, h)$, and for each executing task $t_i, 1 \leq i \leq n$, the right cell interval for t_i, $\text{rci}(t_i, t_{n+1})$ consists of the set of possible base locations for t_{n+1} were some cell in its leftmost column placed adjacent to and in the same row as a cell in the rightmost column of t_i.*

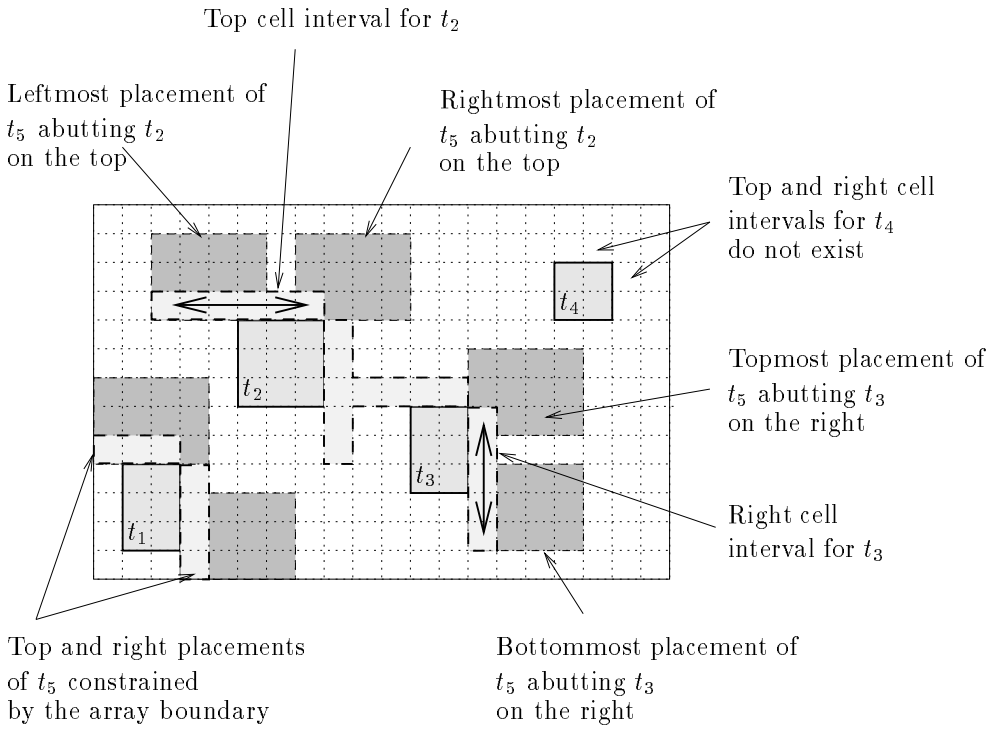


Figure 3.4: Definition of top and right cell intervals for four executing tasks and the darkly shaded waiting task, t_5, with $\mathrm{or}(t_5) = (4, 3)$.

The existence and extent of the right cell interval for t_i given $\mathrm{or}(t_{n+1})$ is constrained by the boundaries of the array but disregards the intersection of t_{n+1} with other executing tasks.

Definition 13 *The right cell interval*

$$\begin{aligned}\mathrm{rci}(t_i, t_{n+1}) \quad = \quad & G^2[(x(\mathrm{tr}(t_i)) + 1, \max\{1, y(\mathrm{bl}(t_i)) - h + 1\}), \\ & (x(\mathrm{tr}(t_i)) + 1, \min\{y(\mathrm{tr}(t_i)), H - h + 1\})]\end{aligned}$$

exists iff $x(\mathrm{tr}(t_i)) + 1 \leq W - w + 1$.

Definition 14 *The right cell intervals for* t_{n+1} *is the set*

$$\mathcal{R}(t_{n+1}) = \{\mathrm{rci}(t_i, t_{n+1}) : 1 \leq i \leq n, x(\mathrm{tr}(t_i)) \leq W - w\} \cup G^2[(1, 1), (1, H - h + 1)],$$

which includes an interval that is defined with respect to the left edge of the array.

Similar definitions can be made regarding the placement of the waiting task in the vicinity of the other edges of executing tasks. Those that can be made with respect to the topmost row of an executing task follow.

Definition 15 *For the waiting task t_{n+1} the top cell interval for t_i is the set of possible base locations* $\mathrm{tci}(t_i, t_{n+1})$ *were some cell in the bottommost row of the waiting task placed adjacent to and in the same column as a cell in the topmost row of t_i.*

The top cell interval

$$\begin{aligned}\mathrm{tci}(t_i, t_{n+1}) \quad = \quad & G^2[(\max\{1, x(\mathrm{bl}(t_i)) - w + 1\}, y(\mathrm{tr}(t_i)) + 1), \\ & (\min\{x(\mathrm{tr}(t_i)), W - w + 1\}, y(\mathrm{tr}(t_i)) + 1)]\end{aligned}$$

exists iff $y(\mathrm{tr}(t_i)) + 1 \leq H - h + 1$.

The top cell intervals for t_{n+1} is the set

$$\mathcal{T}(t_{n+1}) = \{\mathrm{tci}(t_i, t_{n+1}) : 1 \leq i \leq n, y(\mathrm{tr}(t_i)) \leq H - h\} \cup G^2[(1,1), (W - w + 1, 1)],$$

which includes an interval defined with respect to the bottom edge of the array.

The cells at the intersection of the set of right and top cell intervals for t_{n+1} are of particular interest.

Definition 16 *The set of cells at the intersection of the set of right and top cell intervals for t_{n+1} is denoted $\mathcal{I}(t_{n+1}) = \mathcal{R}(t_{n+1}) \cap \mathcal{T}(t_{n+1})$.*

Definition 17 *Let the set $\mathcal{B}(t_{n+1})$ denote the union of the set of cells in $\mathcal{I}(t_{n+1})$ with the bottommost cells of each* $\mathrm{rci}(t_i, t_{n+1}) \in \mathcal{R}(t_{n+1})$ *and the leftmost cells of each* $\mathrm{tci}(t_i, t_{n+1}) \in \mathcal{T}(t_{n+1})$.

Figure 3.5 illustrates the set $\mathcal{B}(t_{n+1})$ for the example of Figure 3.3.

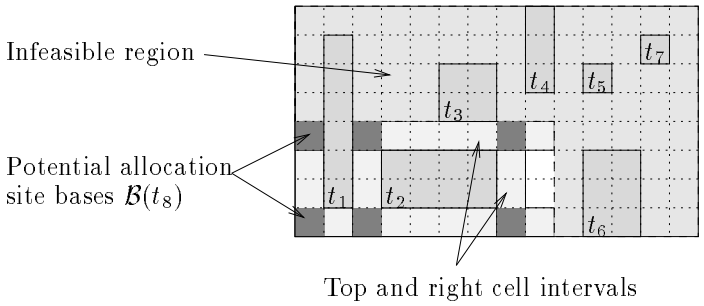


Figure 3.5: Potential allocation site bases for a waiting task t_8 with $\mathrm{or}(t_8) = (6, 5)$ appear shaded darkly.

Theorem 2 *If the waiting task t_{n+1} can be allocated by right ordered compaction, then the time needed to complete the compaction is minimized for an allocation site based at some cell in $\mathcal{B}(t_{n+1})$.*

Proof: The proof considers the time needed to free the allocation site for all possible base locations of the waiting task. The assumption is that the time needed to complete the compaction is at least proportional to the area of tasks that need to be moved out of the allocation site.

The right cell interval for t_i is the leftmost column in t_i's neighbourhood where t_{n+1} can be placed without intersecting t_i. Refer to Figure 3.6. Were the placement of t_{n+1} to intersect t_i, t_i would have to be moved to the right of the allocation site for t_{n+1} by the right ordered compaction strategy, thereby increasing the time needed to complete the compaction. Placing t_{n+1} to the right of the right cell interval for t_i does not reduce the cost of freeing the area needed by t_{n+1}. Indeed, it could increase the cost by intersecting additional tasks on the right boundary of the allocation site. For example, see task t_k in Figure 3.6. These additionally intersected tasks would need to be moved as well, thereby increasing the time needed to complete the compaction.

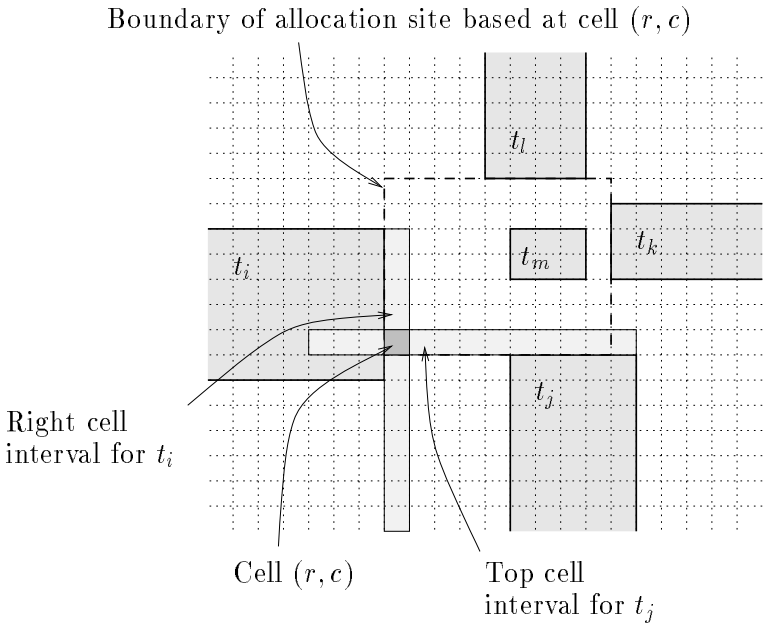


Figure 3.6: Allocation sites based at the intersection of right and top cell intervals are locally optimal. Minor displacements from these local optima can force additional tasks to have to be removed from the allocation site.

If the right cell interval for t_i intersects the top cell interval for t_j at cell (r, c) say, then if t_{n+1} were based at (r, c), it would, as described above, be constrained from being placed further to the left or right without potentially increasing the time needed to free the allocation site. The waiting task would also be constrained from being located above or below the top cell interval for t_j by a similar argument since a slightly lower placement of the site would intersect t_j, forcing it to be moved, and a slightly higher placement would force the movement of any tasks that become intersected at the top edge of the site. In Figure 3.6, for example, task t_l is intersected if the allocation site is based above the top cell interval for t_j.

If the right cell interval for t_i is not intersected by a top cell interval, then it is possible for an allocation site based at a cell in the interval to intersect another task in one way only. The site could intersect a task t_j, to the right of and in the vicinity of t_i, whose top edge is flush with, or above the top edge of t_i. See Figure 3.7 for an example. Basing the waiting task at the bottommost cell of the right cell interval avoids the need for compaction if the site does not intersect such a task. On the other hand, no more time is needed to complete the compaction for a site based at this cell than at any other cell in the right cell interval for t_i since each location forces t_j to have to move.

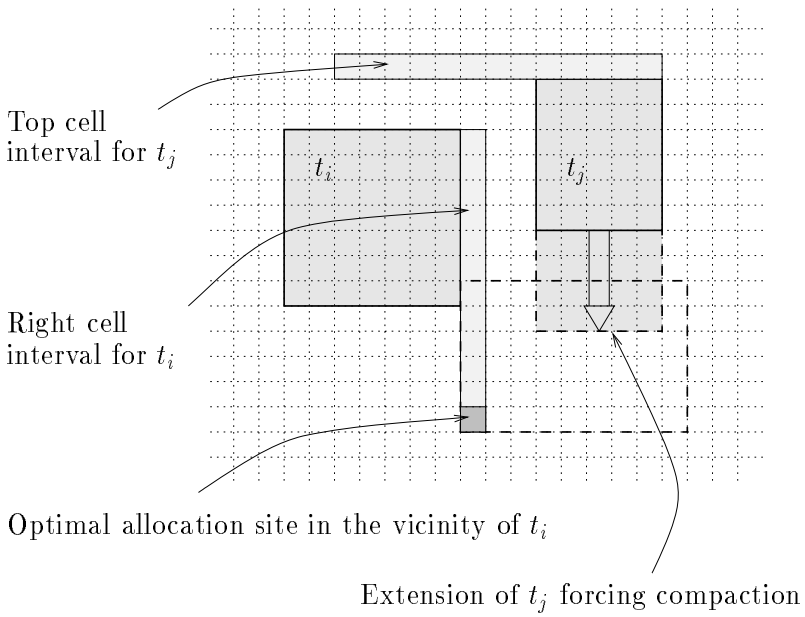


Figure 3.7: Right cell intervals that are not intersected may offer opportunities for allocating the waiting task without compaction. In any case, checking the bottommost cell is optimal.

A similar argument can be used to show that it is only necessary to check the leftmost cell of each top cell interval when the interval is not intersected by a right cell interval. ∎

Constructing the set $\mathcal{B}(t_{n+1})$ of potential bases for the waiting task requires $O(n^2)$ time if each member of the set of right cell intervals is used to check for intersections against each member of the set of top cell intervals. Since $O(n^2)$ potential base locations have to be identified, this is optimal in the worst case. The time complexity of the average case can be improved by processing the tasks in sorted order.

3.1.3.2 Assessing Allocation Site Feasibility

Allocation sites based at cells in $\mathcal{B}(t_{n+1})$ are not guaranteed to be feasible because it may not be possible to compact the executing tasks within such a site to the right due to lack of space. An efficient way of assessing feasibility is to build a visibility graph of the executing tasks.

Definition 18 *(After [63]) A task v is said to dominate a task t if, for some cell (r_v, c_v) of v and some cell (r_t, c_t) of t, $r_v = r_t$ and $c_v > c_t$. Where v dominates t, v is said to directly dominate t if there is no task u such that v dominates u and u dominates t. A visibility graph is a directed graph having the collection of executing tasks as vertex set and for each pair of tasks t and v it contains an edge from t to v iff v directly dominates t.*

Figure 3.8 depicts the visibility graph for the example of Figure 3.3.

The visibility graph is built in $O(n^2)$ time in the following way. The list of executing tasks is sorted into increasing base column order, whereby if two or more tasks share a column, they are sorted into increasing row order. For each task a graph vertex is created and inserted in sorted order. A vertex already in the graph has associated with it the bottommost and topmost rows intersected by tasks in its subgraph. Vertex insertion can therefore be done in linear time by a depth first search of vertices not visited before to determine whether or not the task dominates some task in the subgraph. The distance from the parent to a newly added child is associated with each inserted edge. After the graph has been built, the maximum distance the task can be moved to the right is stored at each vertex. The distance the terminal nodes can be moved is given by their base columns and their widths.

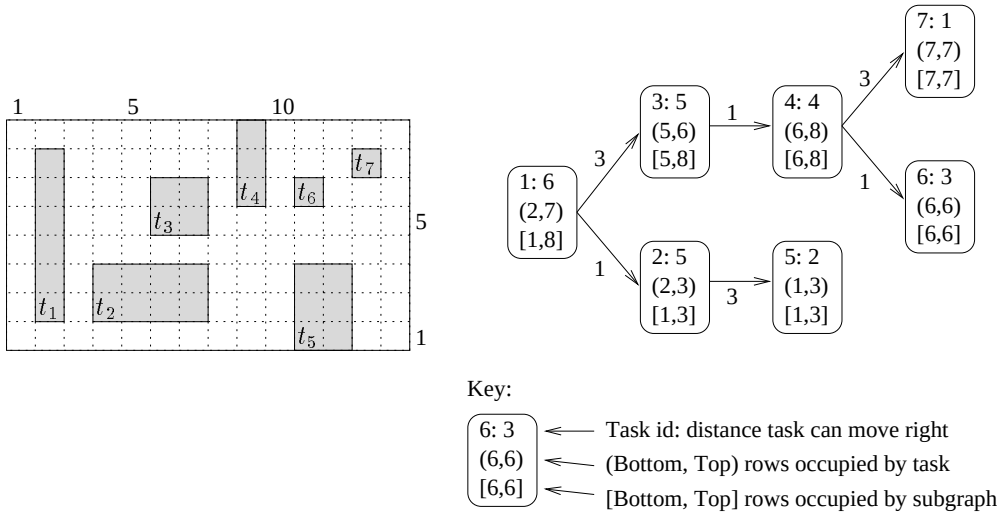


Figure 3.8: An arrangement of tasks on the left, with its visibility graph depicted on the right.

The distance non–terminal nodes can be moved can be computed in $O(n)$ time by summing the edge distances in a bottom–up fashion. This final step saves time during searching by eliminating the need to traverse subgraphs for allocation sites that cannot be freed of executing tasks.

It is possible to reduce the $O(n^2)$ time needed to build the visibility graph if it is updated as allocations, deallocations, and rearrangements occur. Node insertions or deletions into the visibility graph can be performed in linear time because at most n arcs need to be updated. Since ordered compactions only affect the distances nodes can be moved, not the graph topology, their impact can be propagated in linear time as well.

For each potential base $b \in \mathcal{B}(t_{n+1})$, the subgraphs that span rows intersected by the allocation site were it based at b are searched. The leftmost tasks that intersect the potential allocation site can be identified by a depth first search. Once they have been found, the feasibility of moving them right the required distance can be checked. These checks require $O(n)$ time in total because it is possible that each task needs to be examined. If the potential base admits a feasible rearrangement, the set of tasks that need to be moved can be listed in linear time by searching the subgraphs of the leftmost tasks intersecting the allocation site.

The order in which potential allocation site bases are searched influences the efficiency of the ordered compaction method. It is desirable to search the bases

as they are identified in a left to right sweep across the array because tasks that intersect potential allocation sites based closer to the left edge have a better chance of being accommodated on the right. To this end, it is useful to check potential allocation site bases in the order generated when right cell intervals are chosen in increasing column order. However, sites closer to the left may involve moving a greater total area of tasks than sites further to the right. These potentially less costly sites become more sparse as the sweep progresses because it becomes more difficult to compact the intersected tasks. The search for the best allocation site could therefore be abandoned when the cost of further searching exceeds the marginal benefit of finding less costly compaction schedules.

The array geometry suggests there are many avenues for improving the performance of the algorithms by parallelizing them. In particular, searching the visibility graph could be done locally were the graph embedded in a mesh. The best site would then be identified by a reduction operation.

The ordered compaction method experimented and reported upon in Chapter 5 abandoned the search when the first feasible allocation site was identified.

3.2 Scheduling FPGA Rearrangements

In this section, the problem of scheduling FPGA task rearrangements so as to minimize delays for executing tasks is shown to be NP–complete by a reduction from the PARTITION problem. It is then shown that the problem of optimally scheduling the tasks can be viewed as a search for an optimal path in a state–space tree, and an optimal heuristic search procedure that is based on the A* algorithm is derived. Since this exact algorithm can use exponential time and space to find an optimal path, a polynomial time depth–first search heuristic is also presented. A simple local cost estimator that results in reasonable performance for both heuristics is described. For ordered compaction, an algorithm that minimizes the delay to executing tasks at the cost of delaying the allocation of the waiting task is described.

3.2.1 FPGA Rearrangement Scheduling is NP–Complete

Definition 19 *Given two arrangements of a set of FPGA tasks, an initial arrangement* $A(G^2[(1,1),(W,H)]) = \{a(t_i) : t_i \in T\}$ *and a final arrangement* $A'(G^2[(1,1),(W,H)]) = \{a'(t_i) : t_i \in T\}$, *the intersection set of task* t_i, $I(t_i) \subseteq T - \{t_i\}$, *is the set of tasks in the initial arrangement that are intersected by* t_i *when it is placed into*

its final position, i.e., $I(t_i) = \{t_j : a(t_j) \cap a'(t_i) \neq \emptyset\}$.

Given an initial and a final arrangement of a set of FPGA tasks, a method for rearranging the tasks, i.e., moving the tasks from their initial to their final partitions, is sought that minimizes the delay (defined below) to tasks subject to the following constraints. These constraints arise as a consequence of the FPGA model and the scheduling goals, which are listed at the introduction to this chapter.

C1: A task must be removed from its initial position on the array before it can be placed into its final position. The removal of a task from the array is instantaneous.

C2: Only one task at a time can be placed. A task can only be placed into its final position and its placement cannot be interrupted. The time needed to place a task is equal to its size $s(t_i) = w(\text{or}(t_i)) \times h(\text{or}(t_i))$.

C3: Any tasks in $I(t_i)$ that have not yet been removed from the array at the instant the placement of t_i commences are simultaneously removed from the array.

C4: The waiting task t_{n+1}, which is assumed to be initially removed from the array and therefore without an initial position, is the first task placed into its final position.

Definition 20 *The elapsed time between the removal of a task from the array and the commencement of its placement represents a delay to the task.*

Let $r(t_i)$ be the time t_i is removed from the array, $p(t_i)$ be the time the placement of t_i commences, and $d(t_i) = p(t_i) - r(t_i)$ be the delay to t_i. The sequencing constraints can then be formulated in the following way:

$$
\begin{aligned}
& r(t_i) \leq p(t_i) && \text{(C1)},\\
& p(t_i) > p(t_j) \Rightarrow p(t_i) \geq p(t_j) + s(t_j) && \text{(C2)},\\
& \forall t_j \in I(t_i), r(t_j) \leq p(t_i) && \text{(C3)},\\
& r(t_i) \leq \min\{p(t_i), \{p(t_j) : t_i \in I(t_j)\}\} && \text{(C1 \& C3), and}\\
& r(t_{n+1}) = p(t_{n+1}) = 0 && \text{(C4)}.
\end{aligned}
$$

The problem now is to determine the complexity of finding a schedule $p : T \rightarrow Z_0^+$ that minimizes $\max\{d(t_1), d(t_2), \ldots, d(t_n)\}$.

FPGA REARRANGEMENT SCHEDULING

INSTANCE: A set $T = \{t_1, \ldots, t_{n+1}\}$ of tasks and a delay bound $D \in Z^+$. For each task $t_i \in T$, a size $s(t_i) \in Z^+$ and an intersection set $I(t_i) \subseteq T - \{t_i\}$.

QUESTION: Is there a schedule $p : T \rightarrow Z_0^+$ subject to C1 through C4 with $\max\{p(t_j) - p(t_i) : t_j \in I(t_i)\} \leq D$ for all i?

Theorem 3 *FPGA REARRANGEMENT SCHEDULING is NP–complete.*

Proof: It is easy to see that FPGA REARRANGEMENT SCHEDULING is in NP since a non–deterministic algorithm need only guess a schedule and then check in polynomial time that the placement constraints and the delay bound are met. To show that the FPGA REARRANGEMENT SCHEDULING is NP–complete, the well–known PARTITION problem is reduced to it [29].

Let the non–empty set $A = \{a_1, \ldots, a_n\}$ with size $s(a_i) \in Z^+$ for each $a_i \in A$ constitute an instance of PARTITION, and let $S = \sum_{i=1}^{n} s(a_i)$. Then construct an instance of the FPGA REARRANGEMENT SCHEDULING problem consisting of $n + 3$ tasks with $D = 4S + \lfloor S/2 \rfloor$ such that the delay bound can be met if and only if the set A can be partitioned into two subsets $A' \subset A$ and $A - A'$ such that $|A'| = |A - A'| = \lfloor S/2 \rfloor$.

Let $t_i = [w(\mathrm{or}(t_i)), h(\mathrm{or}(t_i))]$ denote a task oriented with width $w(\mathrm{or}(t_i))$, height $h(\mathrm{or}(t_i))$, and size $s(t_i) = w(\mathrm{or}(t_i)) \times h(\mathrm{or}(t_i))$. Set

$$\begin{aligned} t_1 &= t_{n+3} = [2S, 2], \\ t_2 &= [S, 1], \end{aligned}$$

and construct a further n tasks corresponding to the items in A,

$$t_{i+2} = [s(a_i), 1], 1 \leq i \leq n.$$

The initial arrangement of the tasks for a particular instance is illustrated in Figure 3.9. Task t_1 initially occupies the second and third rows from the bottom of an array of width $2S$ and height 5. Tasks t_2 through t_{n+2} are arranged in sequence from left to right along the fourth row from the bottom of the array.

The final arrangement of the tasks of the example is shown in Figure 3.10. The bottom–left corner of task t_{n+3} is aligned with the bottom–left corner of the array, and tasks t_1 through t_{n+2} have been shifted up a row.

From the initial and final arrangements it can be seen that the intersection set of t_1 is $I(t_1) = T - \{t_1, t_{n+3}\}$, of t_{n+3} is $I(t_{n+3}) = \{t_1\}$, and of all other tasks

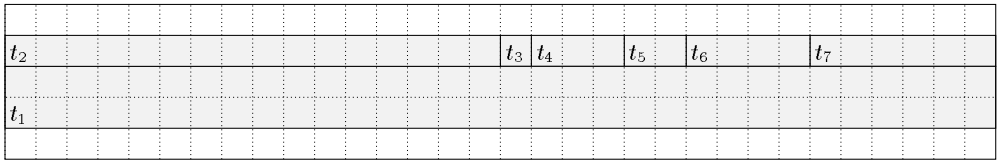


Figure 3.9: The initial arrangement of the tasks for an instance of PARTITION with $A = \{1, 3, 2, 4, 6\}$.

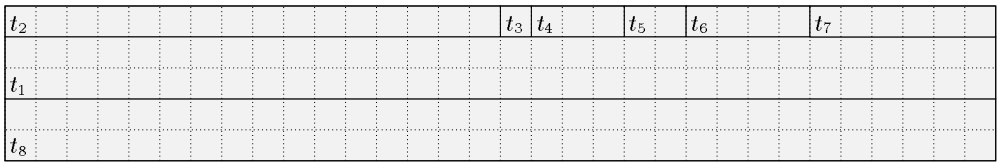


Figure 3.10: The final arrangement of the tasks for an instance of PARTITION with $A = \{1, 3, 2, 4, 6\}$.

is empty. The tasks $t_{i+2}, 1 \leq i \leq n$, their intersection sets, and the intersection set of task t_1 can be constructed in a linear scan of the set A. The magnitude of S can be established at the same time, whereupon the tasks t_1, t_2, and t_{n+3} can be constructed in constant time. The initial and final arrangement of tasks need not be computed since the intersection information is conveyed by the intersection sets. The transformation from an instance of PARTITION to an instance of FPGA REARRANGEMENT SCHEDULING can thus be done in linear time.

As t_{n+3} is placed first of all, t_1 is removed from the array at time $p(t_{n+3}) = 0$. If t_1 is placed next, then the remaining tasks are removed from the array at time $p(t_1) = p(t_{n+3}) + s(t_{n+3})$ and can only be placed after t_1 has been placed. In order to minimize the delay to the remaining tasks, they are best placed in non–decreasing order of size. But this ordering places t_2 last of all and delays it for a total of $5S$ (the size of tasks t_1 and t_3 through t_{n+2}). If this delay is to be reduced, some tasks from $T - \{t_1, t_{n+3}\}$ must therefore be moved before t_1 is placed. However, tasks totaling no more than $\lfloor S/2 \rfloor$ in size can be placed after t_{n+3} and before t_1 if the delay on t_1 is not to exceed D.

Since $s(t_2) = S$, it must be placed after t_1, and placing t_2 before any other tasks that are also chosen to be placed after t_1 forces those tasks to be delayed by $5S$ at least since they would be delayed by the time taken to place tasks t_1 and t_2. Any correct schedule must therefore place t_2 last of all. Placing tasks with a total size of less than $\lfloor S/2 \rfloor$ after t_{n+3} and before t_1 leaves tasks with a total size of more

than $\lfloor S/2 \rfloor$ to be removed from the array with t_2 when t_1 is placed, thereby delaying task t_2 by more than $4S + \lfloor S/2 \rfloor$.

Therefore, if the tasks t_3 through t_{n+2}, can be partitioned into a pair of disjoint sets of size $\lfloor S/2 \rfloor$ in total, then a schedule satisfying the bound D can be found. Since there is a one–to–one correspondence between these tasks and the elements of A in the given PARTITION instance, it can be seen that if and only if a partition of the set exists, a schedule meeting the bound can be found.

Similarly, if the bound of the FPGA REARRANGEMENT SCHEDULING problem corresponding to an instance of the PARTITION problem can be met, then one of the possible partitions is given by the elements corresponding to the tasks placed respectively between tasks t_{n+3} and t_1 and between t_1 and t_2 in the schedule. If a valid partitioning of the set A does not exist, then no schedule can meet the bound. ∎

Corollary 2 *With the constraint of placing the waiting task first of all, scheduling the ordered compaction of FPGA tasks to minimize delays to executing tasks is NP–complete in one or two dimensions.*

Proof: The construction of the proof of Theorem 3 orderly rearranges the tasks, so it is clear the proof holds in two dimensions.

The proof is easily adapted to one dimension by converting the two–dimensional arrangements of Figures 3.9 and 3.10 to arrangements in one dimension. The conversion maps the cells of each array as they appear in row major order, from the bottom of the array to the top, to the cells of the linear array as they appear from the left to the right. ∎

Corollary 3 *Without the constraint of placing the waiting task first of all, FPGA rearrangement scheduling is NP–complete.*

Proof: Consider an arrangement in which the tasks representing the elements of the PARTITION set completely fill a single row, and an additional task, which also fills a single row, is to be rearranged such that the two rows containing the tasks are to be exchanged. In this case there is a cyclic dependency between the solitary task and the tasks of the PARTITION set. Delays to tasks are therefore minimized by moving tasks totaling half the area of the PARTITION set first, followed by the solitary task, and then the remainder of the PARTITION set. ∎

3.2.2 FPGA Rearrangement Scheduling as Heuristic Search

The FPGA rearrangement scheduling problem may be thought of as a search for a task reconfiguration sequence that minimizes the maximum delay to tasks. With n tasks to rearrange after configuring the waiting task, there are $n!$ different ways of sequencing the rearrangement. Each of these can be viewed as a path from the root of a tree to a leaf, in which a node $c_i, 0 \leq i \leq n$, represents the ith sequencing choice. From the specification of the problem, the waiting task t_{n+1} is chosen to be placed at the root c_0. The initially executing tasks are then chosen to be reconfigured in the sequence $c_1, c_2, \ldots, c_n$. The state of the search at any node c_i can be deduced from the unique path $c_0, c_1, c_2, \ldots, c_i$ taken from the root to c_i. The sizes of the tasks determine the times at which a choice can be carried out, and thus the time at which tasks are suspended as they become intersected. It is therefore also possible to determine which tasks have not yet been suspended or relocated, and by how much the placed tasks have been delayed. In FPGA rearrangement scheduling, each path has a cost associated with it, which is the maximum of the execution delays to the tasks when they are relocated in the sequence given by the path. The FPGA rearrangement scheduling problem is to find a cost–minimal path, which is known as a solution path.

At a node, the search for a cost–minimal path proceeds by calculating the cost associated with each arc leaving the node. This process is called expanding the node. After a node has been expanded, a decision is made about which node to expand next. For the search for a solution path to be efficient, as little as possible of the tree is expanded. Searching for a cost–minimal path blindly in a breadth–first or depth–first manner is impractical because there are $n - i$ possibilities for the next sequencing choice at node c_i — one for each task remaining to be placed into its final position. However, the search can be made more efficient through the use of heuristic information to guide the choice. The idea is to expand the node that seems most promising. Such a search is called an *ordered search* or *best–first search* [7]. One way of judging the promise of a node is to estimate the cost of a solution path which includes the node being evaluated. This estimate, made by an evaluation function, is based on the current state and knowledge about the problem domain. How well the evaluation function discriminates between promising and unpromising nodes determines the effectiveness of the search.

3.2.3 Optimal Heuristic Search — the A* Algorithm

A well–known optimal ordered search algorithm applicable to finding minimal–cost paths in directed acyclic graphs is the A* algorithm [7]. Its distinctive feature is its definition of the evaluation function f^*. In a tree, the evaluation function $f^*(c_i)$ estimates the minimal cost of a path from the root to a leaf passing through node c_i by summing the exact cost of reaching the node from the root, $g(c_i)$, and an estimate $h^*(c_i)$ of the minimal cost of reaching a leaf from c_i. It can be shown that A* is guaranteed to find a solution path if h^* is a nonnegative under–estimator of the minimal cost of reaching a leaf from the node being evaluated and all arc costs are positive. Although $h^*(c_i)$ is required to be a lower bound on $h(c_i)$, the actual cost of reaching a leaf from c_i, the more nearly h^* approximates h the better the algorithm performs. Algorithm A_1 is said to be more informed than algorithm A_2 if, whenever a node $c_i, 0 \leq i < n$, is evaluated, $h_1^*(c_i) > h_2^*(c_i)$. If algorithm A* is more informed than algorithm A, then A* never expands a node that is not also expanded by A. It is in this sense that A* is considered optimal. The procedure *EFRS* of Figure 3.11 finds a solution path based on the A* algorithm.

It remains for the nature of the evaluation function f^* to be described. The cost of reaching a node $g(c_i)$ is given by the maximum of the delays to the relocated tasks, which is known. A simple estimator of the minimal–cost path to reach a leaf from the node is also available: in calculating $h^*(c_i)$, ignore the executing tasks, and determine the maximum amount by which the suspended tasks could be delayed. This approach is examined in more detail below. With this estimate in hand, the minimum cost of a path to a leaf through c_i is then given by $f^*(c_i) = \max\{g(c_i), h^*(c_i)\}$.

Ignoring the list of executing tasks allows the suspended tasks to be optimally scheduled in polynomial time. This fact is proved next by proving a slightly stronger result. Since the suspended tasks are scheduled such that the maximum delay to them is minimized, $h^*(c_i)$ is guaranteed to be an underestimate of the actual delays incurred by them when tasks that remain to be moved are considered as well.

Lemma 2 *If $r(t_i)$ is the time at which an FPGA task t_i is removed from the array and $s(t_i)$ is its size, then the suspended tasks are optimally scheduled in non–decreasing $r(t_i)+s(t_i)$ order if none of them causes additional tasks to be suspended.*

[2] In Procedures *EFRS* and *AFRS*, if t_i is suspended, then its source list is the list of suspended tasks, otherwise it is the list of executing tasks.

Procedure **ExactFPGARearrangementScheduling (EFRS)**

Input A list of n tasks to be rearranged, and a description of the waiting task. For each task, its size and intersection set is given.

Output A sequence in which the tasks ought to be rearranged so as to minimize the maximum execution delay to tasks.

Note The algorithm is based on the A* heuristic search algorithm [7].

begin

1. Create an open state with the waiting task placed onto a list of reconfigured tasks. Place the tasks intersected by the waiting task onto a list of suspended tasks, and place the remaining tasks onto a list of executing tasks.
2. Calculate f^* for this state, and place the state on the list of open states.
3. While a solution path has not been found:
   (a) Remove that state from the list of open states for which f^* is minimal, and save it as the current state.
   (b) If the list of suspended tasks and the list of executing tasks for the current state are both empty, then iterate (a solution path has been found).
   (c) For each task t_i not yet relocated:
      i. Create an open state copy of the current state with task t_i removed from its source list[2] and appended to the list of reconfigured tasks. Remove the remaining executing tasks intersected by t_i from the list of executing tasks, and insert them into the list of suspended tasks.
      ii. Update f^* for the open state, and place it on the list of open states.
   (d) Discard the current state.
4. Report the sequence in which tasks were reconfigured.

end

Figure 3.11: Procedure *ExactFPGARearrangementScheduling (EFRS)*.

Proof: Consider the expressions for the delay to suspended tasks t_i and t_j assuming $r(t_i) + s(t_i) \leq r(t_j) + s(t_j)$ and neither causes additional tasks to be suspended. If at time τ, t_i commences reconfiguration before t_j, then t_i is delayed for $\tau - r(t_i)$ time units, and t_j is delayed for at least $(\tau + s(t_i)) - r(t_j)$ time units. On the other hand, were t_j scheduled first, it would be delayed for $\tau - r(t_j)$ time units, and t_i would be delayed for at least $(\tau + s(t_j)) - r(t_i)$ time units. It is assumed that $s(t_i) - r(t_j) \leq s(t_j) - r(t_i)$, hence $\tau + s(t_i) - r(t_j) \leq \tau + s(t_j) - r(t_i)$. Clearly, $\tau - r(t_i) \leq \tau + s(t_j) - r(t_i)$ as well, so the delays to tasks when t_i is scheduled first are no greater than the delay to t_i were t_j scheduled first.

To see that the ordering is suboptimal if additional tasks are suspended, consider the example in which $r(t_i) = r(t_j) = 1$ and $1 < s(t_i) < s(t_j)$. Let us assume that t_i is reconfigured at $\tau = 2$ and that it causes a task t_k with $s(t_k) \geq s(t_j)$ to be suspended when it is placed; thus $r(t_k) = 2$. The specified ordering schedules t_j to be reconfigured after t_i at time $2 + s(t_i)$ and t_k to be reconfigured at $2 + s(t_i) + s(t_j)$, giving delays of $1, 1 + s(t_i)$, and $s(t_i) + s(t_j)$ respectively for t_i, t_j, and t_k. However, the maximum delay can be reduced by scheduling t_j to be placed first of all at $\tau = 2$. Task t_i is then reconfigured at $2 + s(t_j)$, and t_k is only suspended at $2 + s(t_j)$ and reconfigured at time $2 + s(t_j) + s(t_i)$. The delays to t_i, t_j, and t_k are then $1 + s(t_j), 1$, and $s(t_i)$ respectively, which are all less than $s(t_i) + s(t_j)$. ■

3.2.4 Local Versus Global Choice of the Most Promising Node

The running time of the A* algorithm potentially requires exponential time and space because it attempts to make a globally optimal choice of the most promising node at each step. This section presents a simplification of the algorithm that achieves an acceptable solution most of the time for lower cost. The idea is to make a locally optimal choice of the next node to expand by always expanding the most promising successor of the last node expanded. Such a search is known as an *ordered depth–first search* [7]. The procedure *AFRS* of Figure 3.12 implements the algorithm.

If the evaluation function f^* of the exact algorithm is used by algorithm *AFRS* as well, it is useful to keep the list of suspended tasks in sorted order, and therefore to implement it using a priority queue with $\Theta(\log n)$ insertion and deletion time. Step 1 of *AFRS* then requires $O(n \log n)$ time in the worst case, since $O(n)$ tasks may be intersected by the waiting task. Step 2(b)i will also require $O(n \log n)$

Procedure **ApproximateFPGARearrangementScheduling (AFRS)**

Input The list of n tasks to be rearranged, and the waiting task. For each task, its size and intersection set is given.

Output An approximately minimal–cost sequence for rearranging the tasks.

Note The algorithm is an ordered depth–first heuristic search [7].

begin

1. Create a current state with the waiting task placed onto a list of reconfigured tasks. Place the tasks intersected by the waiting task onto a list of suspended tasks, and place the remaining tasks onto a list of executing tasks.
2. Repeat n times:
   (a) Initialize $f^*_{\min}$ to a large value.
   (b) For each task t_i not relocated yet:
      i. Create an open state copy of the current state with task t_i removed from its source list[2] and appended to the list of reconfigured tasks. Remove the remaining executing tasks intersected by t_i from the list of executing tasks, and insert them into the list of suspended tasks.
      ii. Calculate f^* for the open state, and save it and a new value for $f^*_{\min}$ if $f^* < f^*_{\min}$.
   (c) Copy the saved state to the current state.
3. Report the sequence in which tasks were reconfigured.

end

Figure 3.12: Procedure *ApproximateFPGARearrangementScheduling (AFRS)*.

time, and Step 2(b)ii requires a scan of the suspended task list in $O(n)$ time. The running time of algorithm *AFRS* is therefore $O(n^3 \log n)$.

Note that Step 2 examines all possible next states from the previously expanded state and closes all but the best. It was felt that the estimator may not look far enough ahead to be useful when the number of tasks to be rearranged is large. The experimental evaluation reported upon in Chapter 5 therefore compares the performance of *AFRS* and *EFRS* with one– and two–state lookahead. The drawback with looking two states ahead in Step 2 is that it adds another factor of n to the time complexity of the algorithm.

The time complexity of the approximate FPGA rearrangement scheduling procedure might be reduced by a parallel algorithm.

Local repacking was evaluated experimentally using *AFRS* with two–state lookahead for rearrangement scheduling. The results are summarized in the following section and are described in detail in Chapter 5. For ordered compaction, a more straightforward scheduling method was used. A description of the method concludes this section.

3.2.5 Scheduling Ordered Task Movements with Minimum Delay

Given a set of tasks that are to be orderly compacted it is possible to move the tasks without delay according to the visibility graph if a task is not moved until all tasks in its subgraph that must move have moved. This scheduling policy minimizes delays to executing tasks by suspending each task that is to be moved just for the period needed to reload it, and by moving it onto a region of the FPGA that does not overlap any other executing tasks. Scheduling the compaction is straightforward and requires time linear in the number of tasks to be compacted. The only drawback with the policy is that it moves the tasks occupying the allocation site last of all and therefore does not minimize the time needed to free the allocation site. However, this may not be a serious disadvantage since it is the rate at which compactions are completed that determines the rate at which waiting tasks can be allocated.

3.3 Evaluation of Partial Rearrangement Heuristics

This section compares the time complexity and allocation performance of the local repacking and ordered compaction heuristics. Section 3.3.1 compares the

time complexity, and Section 3.3.2 summarizes the results and findings of the experimental investigation described in detail in Chapter 5. The section concludes with a brief discussion of the findings, which motivates the work of the following chapter.

3.3.1 Time Complexity

For an FPGA of width W and height H with $m = \max\{W, H\}$ and n executing tasks, the local repacking heuristic requires $O(mn)$ time to build the free area tree. With $O(m)$ nodes, the tree can be searched in $O(mn \log n)$ time for the existence of a feasible rearrangement. Ordered compaction, on the other hand, needs $O(n^2)$ time to identify potential allocation sites and build the visibility graph. Each of the potential sites can be checked in $O(n)$ time, which leads to a time complexity of $O(n^3)$ for ordered compaction to determine whether a feasible compaction exists or not. These costs can be reduced by dynamically maintaining the free area tree and visibility graph.

In the worst case it is difficult to know which method requires more time. However, in practice only a few nodes at the root of the free area tree are searched, which means $O(mn)$ time is spent building the tree, and a few searches at a cost of $O(n \log n)$ time each are performed. For ordered compaction, the visibility graph needs to be built, and if the potential allocation sites are checked in a left to right sweep, the search can be abandoned after checking the right cell interval on the left array border, which usually suffices to determine compaction feasibility. The cost for ordered compaction is therefore more likely to be $O(n^2)$, which is unlikely to be greater than the $O(mn)$ time needed by local repacking.

Without the constraint of placing the waiting task first of all, ordered compaction needs $O(n)$ time to schedule the rearrangement so as to minimize the delays to the executing tasks whereas local repacking requires $O(n^3 \log n)$ time with one–state lookahead, or $O(n^4 \log n)$ time with two states of lookahead. When the waiting task is to be placed first of all, both methods need to use the approximate scheduling method.

3.3.2 Empirical Performance

Analytical assessments of the ability of the local repacking and ordered compaction heuristics to identify rearrangements whenever they might be feasible are beyond the scope of this thesis. Instead, a simulation study of the performance of the methods was carried out and is reported upon in detail in Chapter 5. This

section presents the main results and conclusions of that study in order to complete the analysis of these methods and to motivate the work on parallel task movements described in the following chapter.

Simulation experiments were carried out to compare the performance of the static first fit allocation method with dynamic allocation methods employing local repacking and ordered compaction whenever first fit failed. Local repacking used procedure *AFRS* with two–state lookahead, while ordered compaction adopted a scheduling strategy that minimized the delay to executing tasks. Both local repacking and ordered compaction were programmed to abandon the search for feasible rearrangements when the first feasible rearrangement was found. This rearrangement was then scheduled.

Several experiments were conducted to compare the performance of the different allocation/rearrangement methods. The first experiment investigated the effect on performance of varying task loads with nominal configuration delays. Both local repacking and ordered compaction performed significantly better than first fit when the FPGA was saturated with work as tasks arrived more quickly than they could be processed. Local repacking appeared to perform marginally better than ordered compaction when tasks were small, whereas ordered compaction performed better when task sizes grew to encompass the entire array. When the FPGA was saturated with work, partial rearrangement reduced the mean allocation delay by up to 24%. The response times were correspondingly lower and the utilization correspondingly higher. When tasks arrived less frequently than they could be processed, the benefits of rearrangement were insignificant.

A second experiment investigated the effect on performance of varying the configuration delay in the saturated operating region. Both methods performed well when the mean configuration delay was very low (less than 1% of the mean service period). However, local repacking began performing worse than first fit at mean configuration delays less than 5% of the mean service period. By comparison, ordered compaction sustained mean configuration delays corresponding to approximately 10% of the service period before performing worse than first fit. The very high execution delays experienced by tasks using the local repacking method is the main factor contributing to its poor performance. Ordered compaction, which delays no task longer than is needed to move it, and which may move tasks with less total area, delays tasks much less. It therefore sustains better performance than first fit over a larger range of configuration delays.

3.3.3 Discussion

Partial rearrangement has the potential to offer considerable performance advantages with acceptable computational effort. Unfortunately, these benefits may be jeopardized by execution delays which render partial rearrangement ineffective at configuration delays that are relatively small compared with the mean task service period.

These execution delays need to be substantially reduced if partial rearrangement is to become more broadly applicable. Since the ratio of the configuration delay to the computational latency of cells is unlikely to change significantly, unless heuristics that reduce the total area of tasks involved in rearrangements are found, new approaches to moving tasks are needed.

The main bottleneck with the current approach is that tasks are reloaded from off–chip in a sequential process that takes time proportional to the area of each task. If it were possible to use the configurable interconnect for moving tasks, then it would be possible to eliminate the sequential reload step and to reconfigure a number of target cells at a time. Furthermore, several tasks might make use of the available bandwidth to move at the same time. Additional performance gains could be expected from reductions in execution delays to individual tasks and rearrangement schedule lengths. Techniques for moving tasks on–chip are described and analyzed in the next chapter.

3.4 Conclusions

Partial FPGA rearrangement aims to reduce allocation delays for waiting tasks by reducing the fragmentation of free cells through the movement of executing tasks. In order to minimize the delay to waiting tasks, the rearrangements should be performed as quickly as possible. When tasks are reloaded, this aim equates to minimizing the total area of the set of tasks to be moved. So as not to offset the benefits of rearrangement, the delay to executing tasks that are to be moved needs to be minimized as well.

Considering identifying feasible rearrangements of FPGA tasks is NP–complete, two heuristics were developed in this chapter. Local repacking uses a quadtree decomposition of the FPGA to identify subarrays that may be capable of accommodating the waiting task if they are repacked. Known strip–packing algorithms are used to attempt the repacking. Ordered compaction, on the other hand,

searches a visibility graph of the executing tasks to determine whether tasks can be moved together in one direction so as to fit the waiting task in the resulting gap — the method can also be used to identify allocation sites when compaction is not required. The fixed costs associated with constructing the data structures needed to search for feasible rearrangements could be reduced by dynamically maintaining them. However, the efficiency of both methods depends upon the search strategy used. It is not clear how the search effort can be minimized.

Scheduling rearrangements so as to minimize the delays to the executing tasks that must be moved was shown to be NP–complete. Exact and approximate scheduling heuristics based on state-space search strategies were therefore developed, and a simple linear time cost function was proposed to guide the order in which nodes are expanded. The more constrained task movements of ordered compaction allow a scheduling method that minimizes the delay to the moving tasks to be used.

It is difficult to distinguish between the heuristics for identifying rearrangements on the basis of worst case performance. It is likely that the cost of identifying feasible ordered compactions is no more than the cost of identifying feasible local repackings. However, the existence of a linear time scheduling algorithm for ordered compaction gives it a scheduling advantage. Moreover, analytical performance bounds on procedure *AFRS* are needed.

It is felt that parallel solutions offer scope for further reducing the time complexity of identifying and scheduling rearrangements.

An experimental assessment of the performance of the methods indicates dynamic allocation by partial rearrangement can be of significant benefit when the configuration delay is a small fraction of the mean service period and when tasks arrive more quickly than they can be processed. Local repacking appears to be slightly more effective than ordered compaction when task sizes are small, however, both methods become ineffective with modest increases in the configuration delay.

In order to increase the range of application, it is proposed to move tasks on–chip as a means of overcoming the I/O bottleneck of reloading configuration bit streams from off the chip. This proposal is developed in the next chapter.

Chapter 4

On–Chip Compaction

The results of the experimental evaluation of partial rearrangements reported upon in Chapter 3 demonstrate that partial rearrangements are an effective means of reducing allocation delays in heavily loaded systems. Unfortunately, the costs associated with reloading tasks from off–chip mitigate the benefits when tasks are short–lived. The costs of rearranging the tasks must therefore be reduced if the technique is to be more broadly applicable. This chapter investigates techniques for using on–chip routing resources to move tasks for less cost.

Of primary interest is the development of fast algorithms for performing arbitrary two–dimensional partial rearrangements using dynamically reconfigurable buses to move the tasks. However, this goal was found to be too ambitious; more modest results are therefore reported. The main contributions of this chapter are to demonstrate the potential benefits of performing task movements on–chip, and to report on the progress towards developing effective one–dimensional ordered compaction scheduling algorithms.

Currently available dynamically reconfigurable FPGAs typically have communication links connecting cells to their nearest neighbours. The first section of this chapter describes the use of these links to perform ordered compactions and then summarizes the results of an experimental study to assess the performance of on–chip compaction.

In the following section, the focus shifts to the use of segmentable buses, which have the potential to improve upon the use of nearest neighbour links when tasks are moved beyond their boundaries, that is, when the final allocation for a task does not overlap its initial allocation. Methods for orderly compacting linear array tasks are investigated because it is hoped that they will provide some of the insights

needed to develop two–dimensional methods and because notable FPGA coprocessor applications that could make use of compaction to improve performance are based on one–dimensional architectures.

The chapter concludes with a summary of the findings and describes avenues for future research.

4.1 Compacting with Nearest Neighbour Links

This section describes the use of nearest neighbour links to perform two–dimensional ordered compactions as described in the previous chapter. The description focuses on the method for scheduling the compaction given that a rearrangement has already been identified. The results of a simulation study to assess the performance of on–chip compaction over nearest neighbour links, which is reported upon in detail in Chapter 5, are then summarized.

4.1.1 Scheduling Ordered Compactions

It is assumed that the methods for identifying and assessing potential allocation sites described in Section 3.1.3 and the method for moving a task over nearest neighbour links described in Section 2.4.2 are used. Given that a set of tasks to be orderly compacted has been chosen, compaction by nearest neighbour links proceeds as follows.

The tasks to be compacted are simultaneously halted and switched out of context. Cells containing task elements that are to be moved then send them to their right neighbours. Cells receiving a task element from the left check whether it has reached its destination and pass it onto the right if not. These steps are repeated until all task elements reach their destination. When a task arrives at its destination, the task elements are switched back into context to resume execution.

Individual tasks are delayed from executing for the time they are in motion, which is proportional to the distance that they move. The time needed to free the allocation site is consequently proportional to the distance the leftmost column of the leftmost task occupying the allocation site must move. The best allocation site thus minimizes this distance and is equal to the number of steps in the ordered compaction schedule. Figure 4.1 illustrates the steps in a schedule to compact the tasks in the example of Figure 3.3 over nearest neighbour links.

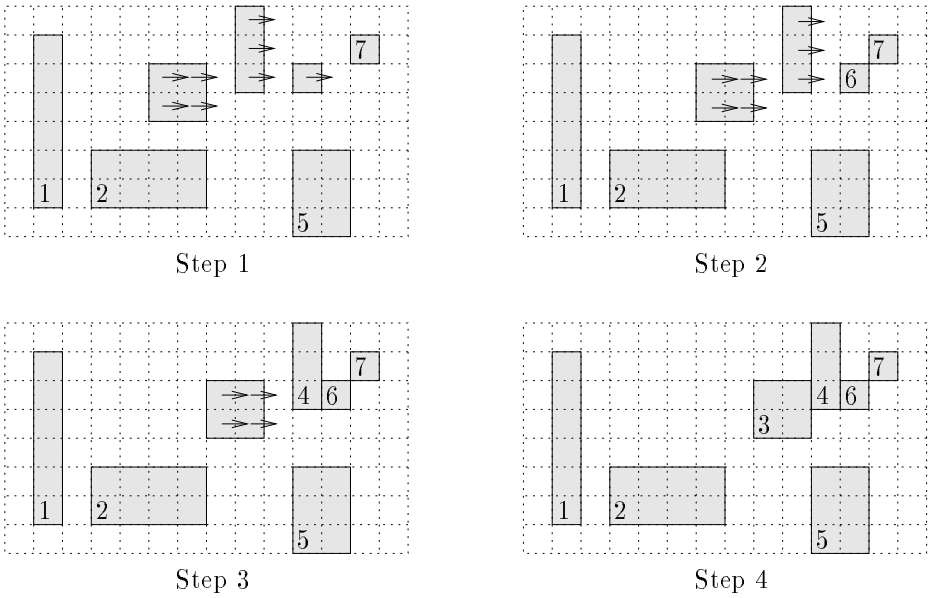


Figure 4.1: Compacting the set of tasks from Figure 3.3 using nearest neighbour links.

4.1.2 Performance Evaluation

A simulation study to investigate the performance of ordered compaction using nearest neighbour links to move tasks is reported upon in detail in Chapter 5. In this section, the main results and conclusions of that study are summarized.

Ordered compaction was used to try to partially rearrange the tasks executing on the FPGA whenever a first fit allocation method failed to find an allocation site for the next waiting task. Instead of moving tasks by reloading them, as considered in the previous chapter, tasks were moved over nearest neighbour links. The delay to a moving task was calculated by scaling the distance it moved by the configuration delay per cell. The effect of moving tasks without cost was also examined for comparative purposes. The time to load the tasks onto the FPGA was calculated by taking the product of the configuration delay per cell and the task's area. It was assumed that a task could be suspended and resumed in a single clock cycle.

The results indicate that ordered compaction using nearest neighbour links eliminates the degradation in performance that occurs when tasks are moved by reloading. Compaction over nearest neighbour links reduces the allocation delay because execution delays are small compared with those that occur when tasks are reloaded. At high configuration delays, the cost of using nearest neighbour links to perform ordered compactions was found to be negligible. These performance im-

provements are achieved by reducing the time needed to move tasks and to complete rearrangements.

4.1.3 Hardware Enhancements

In order to implement the proposed method efficiently, several hardware enhancements to current FPGAs are required.

1. A mechanism for efficiently halting and resuming a subset of the executing tasks that does not affect the remaining tasks is needed.

2. There is a need to support the pipelining of task elements over nearest neighbour links. That is, it should be possible to instruct the cells in specified regions of the FPGA to pass task elements from left to right, and to instruct them to stop doing so at the appropriate time. A local method is desirable.

4.2 Compacting with Segmentable Buses

Ordered compaction over nearest neighbour links performs better than reloading tasks from off–chip. However, when tasks are to be moved large distances, they are suspended longer, and the benefits of compaction may once again be eroded. In order to extract maximum benefit from compaction, longer point–to–point connections are needed. The use of segmentable buses offers the possibility of communicating between arbitrary points in constant time, thereby reducing the time to move tasks. For the remainder of this chapter, techniques for compacting tasks using segmentable buses are discussed.

The investigation of techniques for performing one–dimensional compaction is primarily motivated by the desire to gain insight into the problem or performing arbitrary two–dimensional rearrangements on–chip. However, one–dimensional methods could also find practical use in current and proposed FPGA applications. The throughput and utilization of essentially one–dimensional architectures such as DISC [68] and Garp [34] could be increased by partially rearranging the tasks residing on the array. A DISC or Garp machine can be rearranged by performing a one–dimensional compaction on all columns simultaneously.

Ordered compaction in one dimension proceeds, as in two dimensions, in two steps. First, a feasible rearrangement of the executing tasks is identified, and

second, the rearrangement of the tasks is scheduled. This section discusses one–way ordered compaction, which compacts the executing tasks in one direction only. The simple extension of the techniques described here to two–way compactions is discussed at the conclusion of the section. Without loss of generality, the ordered compaction of tasks to the left is considered.

The idea behind left ordered compaction is to choose a suitable site for the waiting task and to move the executing tasks occupying it to the left as little as necessary to free the site. Moreover, the spatial order of the moved tasks is preserved. In general, these tasks will be moved to cells allocated to other tasks, which will be moved as well. The left ordered compaction protocol identifies a sequence of tasks that can be moved in minimal time to free space for the waiting task and attempts to minimize the maximum execution delay to the moving tasks. Figure 4.2 depicts an example of a left ordered compaction

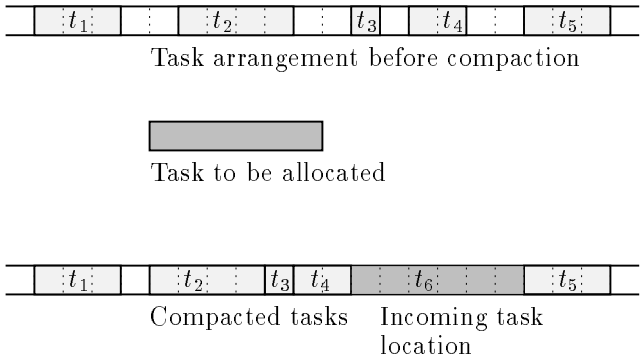


Figure 4.2: An example of a left ordered compaction to accommodate the waiting task t_6.

After deriving a lower bound on the time to move a task, a procedure for identifying an optimal allocation site is described. The compaction scheduling problem is then formalized and conjectured to be NP–complete. An optimal polynomial time algorithm for scheduling the compaction of unit length tasks without delays is presented and discussed. A heuristic for compacting arbitrary length tasks on a linear array is then described. The heuristic frees the space for the waiting task in minimum time and bounds the delay to moving tasks as well as the schedule length. The concluding remarks discuss the extension of these methods to two dimensions.

4.2.1 Notation

The following notation is used in this section. The linear array $G^1[1, L]$ is assumed to be processing a set of n tasks $T = \{t_1, \ldots, t_n\}$. Each executing task t_i has an associated length l_i with $1 \leq l_i \leq L$ and is assumed to be allocated to a block of l_i contiguous cells based at the cell C_{b_i} on the left with $b_i \in [1, L - l_i + 1]$. Task t_i is composed of l_i task elements labelled $t_i[1], \ldots, t_i[l_i]$ from left to right with element $t_i[j]$ assumed allocated to the cell numbered $b_{i,j} = b_i + j - 1$.

The interval of cells $G^1[i, j]$ with $i \leq j$ contains $|G^1[i, j]| = j - i + 1$ contiguous cells. The intersection of cell intervals,

$$G^1[i, j] \cap G^1[k, l] = \begin{cases} G^1[\max(i, k), \min(j, l)], & \text{iff } i \leq l \text{ and } j \leq k \\ \emptyset, & \text{otherwise} \end{cases}$$

Task t_i is thus allocated to the interval of cells $G^1[b_{i,1}, b_{i,l_i}]$. The executing tasks are arranged such that $b_i < b_j$ for $i < j$, and $G^1[b_{i,1}, b_{i,l_i}] \cap G^1[b_{j,1}, b_{j,l_j}] = \emptyset$ for all $t_i, t_j \in T$ with $i \neq j$.

The possibly empty block of free cells between tasks t_{i-1} and t_i is labelled f_i. The leftmost and rightmost free blocks are labelled f_0 and f_{n+1} respectively. The symbol f_i is used to refer to the free block as well as to its size.

It is assumed that the waiting task t_{n+1} with $l_{n+1} > f_i$ for all i is to be allocated.

4.2.2 Selecting an Optimal Allocation Site

In order to minimize the time to allocate the waiting task, it is desirable to place it at a location that takes minimum time to free of executing tasks. Lemma 1 in Section 2.4.3 describes the criterion for computing the optimal time required to move a task. This fact is used to find an optimal allocation site for the waiting task and to compute the length of an optimal schedule for clearing it of executing tasks.

When tasks are to be orderly compacted in a single direction, identifying the optimal rearrangement so as to minimize the allocation delay to the waiting task is easier in one dimension than in two because the allocation site can only intersect tasks in a single dimension. In two dimensions, the cost to free the allocation site can be affected by vertical and horizontal displacements of the site. In one dimension there is only one degree of freedom. The following definitions prepare the way for the characterization of an optimal allocation site.

Definition 21 *A task t_i is said to be totally (partially) covered by another task t_j if all (respectively some, but not all) of the cells allocated to t_i need to be reallocated to t_j.*

From Lemma 1, the cost to uncover a task that is totally covered by the allocation site is proportional to its length because it must be moved further than its size. However, the cost to uncover a task that is partially covered at one end by the incoming task is dependent on the direction it is moved in. If the uncovered portion is moved away from the incoming task, then the cost to move it is at least proportional to the number of cells covered. Otherwise, the cost is proportional to its length since the uncovered end of the task must be moved past the covered end. Should the task be so long as to be covered in the middle by the allocation site but be uncovered at both ends, then the cost to uncover the task is given by the sum of the uncovered length that is moved towards the allocation site and the length of the site.

Definition 22 *The sequence of tasks $t_i, t_{i+1}, \ldots, t_j$ with $i \leq j$ is said to be a contiguously allocated sequence of tasks if the free blocks $f_{i+1}, f_{i+2}, \ldots, f_j$ between them all have zero length. The sequence is said to be a maximal contiguously allocated sequence if the free blocks f_i, to the left of t_i, and f_{j+1}, to the right of t_j, each have non–zero length.*

Theorem 4 *The time needed to free an allocation site for the waiting task using left ordered compaction is minimized when the rightmost task element of the waiting task is allocated to the rightmost cell of a free block of non–zero size.*

Proof: The proof considers the time needed to free the allocation site for all possible positions for the rightmost task element of the waiting task as it is shifted from the rightmost cell of a non–empty free block across the maximal contiguous sequence of allocated tasks to the left.

Assume there is sufficient free space to accommodate the tasks partially or totally covered by the waiting task wherever it is placed. Consider the cost to free an allocation site that allocates the rightmost task element of the waiting task to the rightmost cell of a non–empty free block. The incoming task will cover some cells allocated to other tasks because no free block is large enough to accommodate it. Since all partially or totally covered tasks are moved away from the allocation site to the left, the time to free it is proportional to the number of allocated cells

covered by it. As the position of the rightmost task element of the incoming task is shifted to the left, additional allocated cells potentially become covered, thereby increasing the time to free the site. However, as the allocation site is shifted to the left over a contiguously allocated sequence of tasks, the time needed to free the site of occupying tasks can only decrease when an occupied task is completely uncovered. Thus the theorem follows. ■

Theorem 5 *It is necessary to consider both left, and the symmetric right ordered compaction of tasks to minimize the time needed to free an allocation site for the waiting task when tasks are compacted in a single direction*

Proof: It is easy to show that the minimum times needed by left and right ordered compaction to free the incoming task location of occupying tasks are not necessarily equal. For example, the allocated task set may consist of a single task of length $n/2$ allocated to cells $C_2, \ldots, C_{n/2+1}$ and the request to be satisfied is for $n/2$ cells as well. The only feasible site for left ordered compaction is the rightmost $n/2$ cells, which covers 1 task element of the allocated task. On the other hand, the only feasible site for right ordered compaction is the left half of the array, which covers $n/2 - 1$ task elements of the allocated task. ■

An allocation site that takes minimal time to free of executing tasks by a one–way ordered compaction can be found in linear sequential time by scanning a list of task and free block records in base order. The procedure *SelectTasksToCompactLeft* of Figure 4.3 selects the optimal sequence of tasks to compact.

4.2.3 Scheduling One–Dimensional Compaction

In choosing an allocation site, a sequence of tasks that is to be compacted is selected. It is necessary to know the source, destination, and length of each task that is to be compacted in order to perform a left ordered compaction. A left ordered compaction schedule specifies for each task in which step of the schedule it should be suspended and resumed, for each task element in which step it should move from its source to its destination, and for the segmentable bus how it should be switched in each step. The program for switching the bus is assumed to be implied by the schedule for moving the task elements.

After describing the scheduling goals and their influence on the scheduling problem, the complexity of one–dimensional ordered compaction is examined.

Procedure **SelectTasksToCompactLeft**

Input The size of the request and a doubly linked list of task and free block records containing their sizes in base order.

Output Pointers to the left– and rightmost tasks in the sequence of tasks to compact.

begin

1. Scan forwards through the list with a pointer R until R points to a non–zero sized free block record and the total free space to the left of $R.next$ exceeds the request.

2. Repeat until the list has been scanned:

   (a) Scan forwards through the list with a pointer L until the free space between $R.next$ and $L.next$ is less than the size of the request.

   (b) If the allocation site that allocates the rightmost task element of the waiting task to the rightmost cell of R covers the least number of allocated cells of any site considered so far, then

      i. Save pointers to the rightmost ($R.prev$) and leftmost ($L.next$) tasks in the sequence to compact.

   (c) Scan forwards through the list with R until R points to a non–zero sized free block record or until the list has been scanned.

end

Figure 4.3: Procedure *SelectTasksToCompactLeft*.

It is thought likely that the corresponding decision problem is NP–complete. An algorithm for optimally compacting unit length tasks and a heuristic for scheduling arbitrarily long tasks are then described and analyzed.

4.2.3.1 Scheduling goals

The goals of left ordered compaction scheduling are to schedule the movement of tasks from their initial base positions to their destinations in such a way as to:

1. minimize the time needed to allocate the waiting task;
2. minimize the delay to executing tasks; and
3. minimize the length of the compaction schedule.

These goals are constrained by the model, which does not allow the use of cells to be shared for execution, and which requires the buses used to move task elements to be disjoint.

Minimizing Allocation Delay To meet the first objective, the tasks occupying the allocation site need to be moved in a minimum number of steps. To commence executing the waiting task, the tasks covered by the allocation site need to be switched out of context or moved. If these tasks are not moved as soon as they are switched out of context, they are delayed from executing more than the minimum number of steps needed to move them. Tasks covered by the allocation site should therefore be moved as soon as possible so as to be able to commence executing the waiting task and to be in a position to meet the lower bound on the schedule length given by Lemma 3 below.

Minimizing Execution Delay A task must be suspended before it can be moved and the linear order of its task elements prior to suspension must be reestablished over a contiguous set of destination cells before it can be resumed. If the execution delay to a task that must move is to be minimized, it should be suspended immediately prior to moving, its task elements should be moved in the least possible number of steps, and it should be resumed as soon as its task elements have been reassembled at the destination. It is assumed that a cell can store the configurations for two task elements at a time, but that only one of them can be switched in and

taking part in the execution of a task at any time. Thus if some task t_i is moved to cells initially allocated to some other task t_j, t_j must be suspended at the time that t_i is resumed. Furthermore, unless t_j is moved immediately after being suspended, it will be delayed more than the minimum amount of time needed to move it. This constraint appears to have a profound effect on the scheduling complexity.

Rather than minimize the total delay to a task from the time it is suspended until it is resumed, an attempt is made to minimize the component of the delay during which the task is suspended and not moving. This choice is motivated by the decision to place primary emphasis on the allocation delay to the waiting task. An executing task is therefore considered delayed if it is switched out of context for more steps than the minimum number needed to move it. The amount of the delay is the number of steps it is switched out and none of its elements are moving.

Minimizing Schedule Length The following definition assists in the derivation of a lower bound.

Definition 23 *Let the symbol N_0 denote the number of task elements occupying the allocation site that are to be compacted left, and let $S_{\min}$ denote the minimum number of steps in a left ordered compaction schedule.*

The minimum number of steps needed to be able to allocate the waiting task is given by the following lemma, which also places a lower bound on the length of any left ordered compaction schedule.

Lemma 3 *The minimum number of steps needed to complete a left ordered compaction schedule, $S_{\min}$, is N_0 if the cell immediately to the left of the allocation site is initially free, and is $N_0 + 1$ steps otherwise.*

Proof: N_0 steps are needed to move the N_0 task elements occupying the allocation site. If the cell abutting the allocation site is not initially free, then the task element occupying it cannot be moved while other task elements are moved to it or past it from the right because the buses used to move these are all required to be disjoint from the bus needed to move its occupant to the left. ∎

In general, more than N_0 task elements in total have to be moved since the covered tasks are moved to cells occupied by other executing tasks. Several task elements therefore need to share the bus each step in order for the compaction schedule to meet the lower bound.

4.2.3.2 Complexity of One–Dimensional Left Ordered Compaction

Without loss of generality, assume that the complete set of n executing tasks is to be compacted left as a consequence of having to satisfy a request for $l_{n+1} = \sum_i f_i$ cells. The compaction therefore moves task t_j to the l_j cells commencing with the cell numbered $1 + \sum_{i=1}^{j-1} l_i$ on the left, i.e., t_j is moved a distance $d_j = b_j - (1 + \sum_{i=1}^{j-1} l_i)$ to the left.

Let $m_i[j]$ denote the step in which task element $t_i[j], 1 \leq j \leq l_i$, moves. In step $m_i[j], t_i[j]$ moves from the cell numbered $b_{i,j} = b_i + j - 1$ to the cell numbered $b_{i,j} - d_i$ using a bus that is said to span the interval of cells $G^1[b_{i,j} - d_i, b_{i,j}]$.

A task is suspended prior to moving and resumes execution as soon as all its task elements have moved to their destination. Letting the symbol r_i denote the step in which t_i resumes execution following movement, it is given by

$$r_i = \max_j \{m_i[j] : 1 \leq j \leq l_i\} + 1.$$

A task is suspended in the step immediately before it commences movement, or when its cells are switched out in order to resume a reallocated task. Thus if s_i is the step in which t_i is suspended,

$$\begin{aligned} 0 \leq s_i \quad = \quad & \min\{\min_k \{m_i[k] : 1 \leq k \leq l_i\} - 1, \\ & \min_j \{r_j : G^1[b_{i,1}, b_{i,l_i}] \cap G^1[b_{j,1} - d_j, b_{j,l_j} - d_j] \neq \emptyset\}\}. \end{aligned}$$

It is desired to move the tasks covered by the allocation site (assumed to be based at $b_{n+1} = L - l_{n+1} + 1$) as early as possible in the schedule. In order to meet the lower bound of Lemma 3, a schedule with

$$\max_{i,j} \{m_i[j] : G^1[b_{ij} - d_i, b_{ij}] \cap G^1[b_{n+1}, L] \neq \emptyset\} = \sum_i |G^1[b_{i1}, b_{il_i}] \cap G^1[b_{n+1}, L]|$$

is sought.

A decision problem corresponding to the left ordered compaction scheduling problem follows.

1D ORDERED COMPACTION SCHEDULING

INSTANCE: A list of tasks $t_1, \ldots, t_n$ to be compacted left; for each task a base b_i and length l_i; and three positive integers A, D, and S.

QUESTION: If d_i is set to be equal to $b_i - 1 - \sum_{j=1}^{i-1} l_j$, and $b_{n+1} = L - l_{n+1} + 1$, does an assignment of task element moves $m : t_i[j] \rightarrow Z^+$ exist such that:

- whenever $m_i[j] = m_k[l]$ and $(i \neq k$ or $j \neq l)$, $G^1[b_{i,j} - d_i, b_{i,j}] \cap G^1[b_{k,l} - d_k, b_{k,l}] = \emptyset$ (task element movements are disjoint);
- $\max\limits_{i,j} \{m_i[j] : G^1[b_{i,j} - d_i, b_{i,j}] \cap G^1[b_{n+1}, L] \neq \emptyset\} \leq A$ (the allocation site is freed in at most A steps);
- for all i, $r_i - (s_i + 1) - \min\{d_i + 1, l_i\} \leq D$ (no task is delayed by more than D steps longer than needed to move it); and
- $\max\limits_{i} \{r_i : 1 \leq i \leq n\} - 1 \leq S$ (the schedule is completed in at most S steps)?

1D ORDERED COMPACTION SCHEDULING can be visualized as a two–dimensional packing problem as in Figure 4.4. Let each task element have associated with it a vector of unit width directed from its source cell to its destination. Assign each task and its associated vectors a unique colour. The problem is to pack these vectors layer by layer using as few layers as possible to pack all vectors of the same colour. A packing of height at most S is sought such that the vectors leaving task elements occupying the allocation site are packed within the lowest A layers. Moreover, no more than D positive layers should separate the lowest layer containing a vector leaving a task from the highest layer containing a vector coloured the same as one destined for a cell allocated to the task.

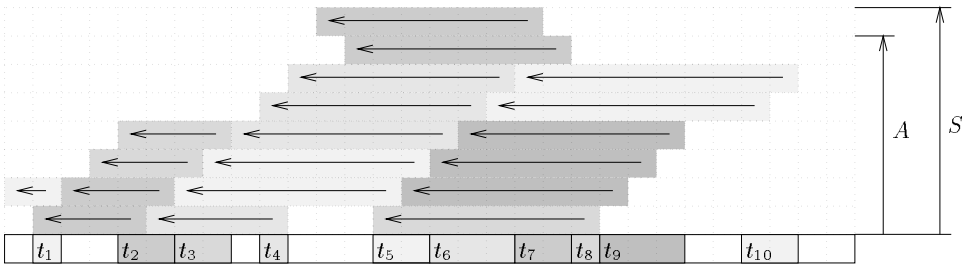


Figure 4.4: A clustered vector packing with $D = 0$.

Lack of success in finding a polynomial time solution to the problem leads to the following conclusion:

Conjecture 1 *1D ORDERED COMPACTION SCHEDULING is NP–complete.*

4.2.4 Compacting Unit Length Tasks

While ordered compaction scheduling of arbitrary length one–dimensional tasks appears to be NP–complete, unit length one–dimensional tasks can be optimally scheduled in linear time on a sequential machine. For this problem, each task

t_i is completely characterized by the cell it occupies, called its source $\mathrm{src}(t_i) = b_i$. Left ordered compaction moves each task as far left as possible without altering the left to right ordering of the tasks. Thus the destination of task t_i is $\mathrm{dst}(t_i) = C_i$.

Definition 24 *The region of the array the n executing tasks are moved to, $G^1[1, n]$, is known as the compaction zone. The array is further subdivided into regions. The region to the right of the compaction zone is denoted R_0. R_0 plays the role of the allocation site and is assumed to contain N_0 tasks to be moved, as in Definition 23. The region these tasks are moved to, the destination or target of the tasks in R_0, is denoted R_1. By definition, this region consists of the N_0 contiguous cells labelled $C_{n-N_0+1}, \ldots, C_n$. Let this region contain N_1 tasks initially. Regions $R_2, R_3, \ldots, R_r$ are defined respectively as the target regions of the $N_1, N_2, \ldots, N_{r-1}$ tasks in regions $R_1, R_2, \ldots, R_{r-1}$. That is, region R_i consists of N_{i-1} contiguous cells abutting region R_{i-1} on the left, and $N_r = 0$.*

The procedure *CompactUnitLengthTasks (CULT)* of Figure 4.5 schedules the optimal movement of unit length tasks. An example of the schedule produced by procedure *CULT* is depicted in Figure 4.6.

In the remainder of this section, the time complexity of procedure *CULT* is analyzed, and its correctness is proved. A lower bound on the length of a left ordered compaction schedule is derived, and it is shown that the procedure achieves this lower bound. Finally, it is shown that the delay to each task is a minimal constant.

It is assumed that the source of each task is presented as input in increasing order. As the input is read, a doubly linked list having a node for each task is built. For each node, an additional pointer is created to point to the node corresponding to the destination of a task, should it be occupied by a task, or to the rightmost task to the left of the destination, if it is initially free. If such a node does not exist, then it suffices to set the pointer to null. Such a list can be built in linear sequential time since the task destinations are also ordered.

The time complexity of procedure *CULT* is given by the following theorem.

Theorem 6 *Procedure CULT computes the schedule of task movements in $O(n)$ steps on a sequential machine. This is optimal.*

Proof: Step 1(a) of the procedure takes constant time if the pointers from t_n back to its destination and then forward to the next task in the list are followed. Step

Procedure **CompactUnitLengthTasks (CULT)**

Input A list of unit length task records containing the source of each task.

Output An assignment of tasks to be moved in each step of the schedule.

begin

1. Schedule the tasks to be moved in the first step:
   (a) Select the leftmost task in R_0 to be moved.
   (b) While there is a task whose source lies to the left of the destination of the previously selected task:
      i. Select the rightmost of these to be moved in the first step of the schedule as well.
2. While R_0 contains a task to be moved, in parallel:
   (a) Move the tasks out of R_0 from leftmost to rightmost in consecutive steps of the schedule.
   (b) Move tasks within the compaction zone in the step after they become the target of a move.
3. If the rightmost task in R_0 was moved to an allocated cell in the previous step, then
   (a) Move the rightmost task of R_1 and any other tasks moved to in the previous iteration of 2(b) in a final step of the schedule.

end

Figure 4.5: Procedure *CompactUnitLengthTasks (CULT)*.

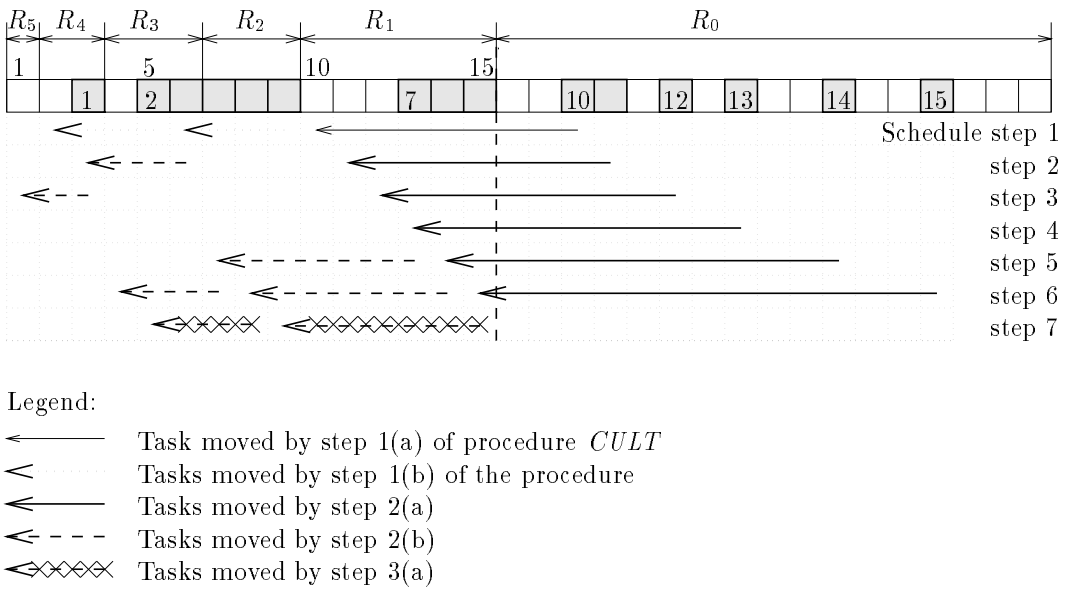


Figure 4.6: Left ordered compaction of unit length tasks by procedure *CULT*.

2(a) requires constant time by following the pointers to the next task each iteration. This step is executed $N_0 - 1$ times. Thus $O(N_0)$ time is needed to move the tasks occupying R_0. The remaining tasks are moved by steps 1(b), 2(b), and 3(a) of the procedure. Step 1(b) requires constant time each iteration when pointers to the previous task and to the destination nodes are followed. When tasks become the target of a move, which can be noted by following the pointers to the destinations of moves, pointers to the targeted tasks can be stored so that they can be moved in the following schedule step. Any data structure requiring constant insertion and removal time such as a linked list allows steps 2(b) and 3(a) to be executed in constant time per task. The remaining $n - N_0$ tasks can therefore be scheduled in $O(n)$ time. Since the schedule needs to associate a possibly unique step with each task, $\Omega(n)$ time is needed to perform the scheduling sequentially. The procedure therefore schedules in optimal time. ∎

Theorem 7 *Procedure CULT correctly schedules the left ordered compaction of unit length tasks.*

Proof: To prove procedure *CULT* correct, it needs to be shown that all tasks are moved to their correct destination and that the buses formed to move tasks are disjoint.

The procedure does not make use of the destinations of task moves other than to select tasks to be moved in the first step of the schedule and to determine

that a task should be moved in step 2(b). It can therefore be assumed that when a task is moved, it is moved to its correct destination assuming that the bus needed to move it does not interfere with any others.

The following propositions, which are proved in Lemmas 4 and 5 below, establish the basis for showing that all tasks are moved.

1. The tasks occupying R_0, to the right of the compaction zone, are moved in the first N_0 steps of the schedule.

2. Apart from the tasks moved in the first step of the schedule and those moved out of R_0, a task is moved in the step after its source receives a task.

Since the sizes of the regions are defined so as to accommodate the tasks arriving from the neighbouring region to the right, the tasks allocated to a region R_i are moved at the latest in the step after they become the destination of a move from the region R_{i-1} to the right. Since all tasks in R_0 are explicitly moved by steps 1(a) and 2(a) of the procedure, it follows that all tasks are moved.

Tasks chosen in step 1(b) of the procedure are selected so as to avoid interference of the buses used to move them. The buses used to move the tasks occupying R_0 do not conflict with any others, as shown in Lemma 4. In Lemma 5 it is proved that tasks that are forced to move as a result of becoming the target of a move in the previous schedule step are free to do so. ∎

Lemma 4 *The N_0 tasks to the right of the compaction zone (occupying R_0) are moved in the first N_0 steps of the schedule.*

Proof: The N_0 tasks occupying R_0 are moved from leftmost to rightmost in N_0 schedule steps according to the actions of steps 1(a) and 2(a) of the procedure. These tasks are moved into region R_1 by definition. Any such task t_k is free to move since it cannot conflict with any forced moves out of R_1 in step 2(b). A task t_i that is forced to move can only have been the target of a task t_j, originating from the left of t_k, and thus the source of t_i must be to the left of the destination of t_k. ∎

Lemma 5 *If the destination in R_i of a move from R_{i-1} during step σ of the schedule is occupied by a task t_j then t_j is scheduled and is free to be moved during step $\sigma+1$.*

Proof: Step 2(b) of the procedure forces a task to be moved in the step immediately following the step in which it becomes the target of a move. Thus if task t_j becomes the target of a move in step σ, it is moved in step $\sigma + 1$.

Task t_j is free to move in step $\sigma + 1$ unless some cell spanned by the bus used to move t_j is the source or destination of a forced move in step $\sigma + 1$ as well. But if a task that is allocated to such a cell is forced to move in step $\sigma + 1$, then it must have been the target of a task coming from the right of t_j in step σ because the order of tasks is preserved. This is not possible since the segment of the bus immediately to the right of the source of t_j was used during step σ to move the task destined for t_j.

By a similar argument, the forced movement of t_j during step $\sigma + 1$ cannot conflict with the forced movement of any task to a cell spanned by the bus used to move t_j since any such task must occupy a cell that would have been spanned by the bus used to move the task destined for t_j during step σ. ∎

Lemma 6 *For all $0 \leq i < r$, the number of tasks initially allocated to R_{i+1} is less than or equal to the number of tasks initially allocated to R_i. Thus the sequence $N_0, N_1, \ldots, N_r$, is non–increasing.*

Proof: By definition, the N_i tasks allocated to region R_i are compacted left to region R_{i+1}, which consists of the N_i contiguous cells abutting the region R_i on the left. At most all of the cells in R_{i+1} are initially allocated, so the number of tasks initially allocated to R_{i+1} is $N_{i+1} \leq N_i$. ∎

In Lemma 3 a lower bound on the number of steps needed by any left ordered compaction scheduling algorithm was provided. In the following theorem it is proved that procedure *CULT* produces a schedule of optimal length.

Theorem 8 *The schedules produced by procedure CULT are at most $S_{\min}$ steps long.*

Proof: The statement is proved by induction on each region. The tasks initially allocated to R_0 are moved in N_0 steps by instructions 1(a) and 2(a) of the procedure. According to Lemma 5, all tasks initially allocated to R_1 are moved out within N_0 steps as well unless the rightmost cell is occupied by a task, in which case an additional step is used by R_1 to move this task. In either case, no more than $S_{\min}$ steps are used to move the tasks initially allocated to regions R_0 and R_1. Now, the rightmost task of region R_2 is moved in the first step since it is the first task chosen

by step 1(b) of the procedure. Irrespective of whether this task was allocated to the rightmost cell of R_2 or not, when the rightmost cell of R_2 becomes the target of the move of the rightmost task from R_1, it is already free, and thus its arrival does not force a move out of R_2 in the following step. Since the last move into R_2 occurs in step $S_{\min}$ at the latest, all tasks are moved out of R_2 into R_3 within $S_{\min}$ steps.

Assume that the tasks initially allocated to regions $R_0, \ldots, R_{i-1}$ are moved out in at most $S_{\min}$ steps, and consider when the last task is moved out of R_i. There are two cases. It may be that the last task to be moved into region R_i is moved in step $\phi < S_{\min}$. Since the number of tasks initially allocated to R_i is at most as many as arrive from R_{i-1}, as shown in Lemma 6, and the tasks initially allocated to R_i are moved at the latest in the step after they become the target of a move, as shown in Lemma 5, the last task to be moved out of region R_i must be moved in step $\phi + 1 \leq S_{\min}$ at the latest.

On the other hand, suppose that the last task is moved into R_i in step $S_{\min}$. In the following, it is proved that in this case its destination is either initially free or freed in the first step of the schedule. The arrival of the last task into R_i therefore does not force a move out of R_i in the following step, and from Lemmas 5 and 6, the last task to move out of R_i does so in step $S_{\min}$ at the latest.

In Lemma 7 below it is established that tasks are moved into a region in cyclic order from left to right. There are three subcases to consider (refer to Figure 4.7 below):

1. The leftmost cell in R_i is hit by the leftmost task τ from R_{i-1} in the first step. Due to Lemmas 5 and 7, all other tasks leave R_{i-1} before $\mathrm{src}(\tau)$ becomes the destination of a move from R_{i-2}. In particular, the last task to leave R_{i-1} is the rightmost task occupying the region, and it leaves before $\mathrm{src}(\tau)$ becomes the destination of a move. Since $\mathrm{src}(\tau)$ is hit in step $S_{\min}$ at the latest, the last task leaves R_{i-1} before step $S_{\min}$, contradicting the assumption that it leaves R_{i-1} in step $S_{\min}$.

2. Some cell C_i other than the leftmost one in R_i is the target of a move in the first step. In this case the rightmost task to the left of C_i is moved in the first step by instruction 1(b) of the procedure. If this task is allocated to the adjacent cell to the left, C_{i-1}, then this cell is freed in the first step of the schedule and thus becomes a free destination for the last move into region R_i. Otherwise, C_{i-1}, which is the last cell moved to in R_i, is already free.

3. No cell in R_i is hit in the first step because the rightmost task to the left of the cell hit in the first step in R_{i-1} is in R_i. In this case, the rightmost task in R_i is moved in the first step. The last task moved to R_i is moved to the rightmost cell in R_i due to Lemma 7, and this cell was either initially free, or allocated to the task moved in the first step.

■

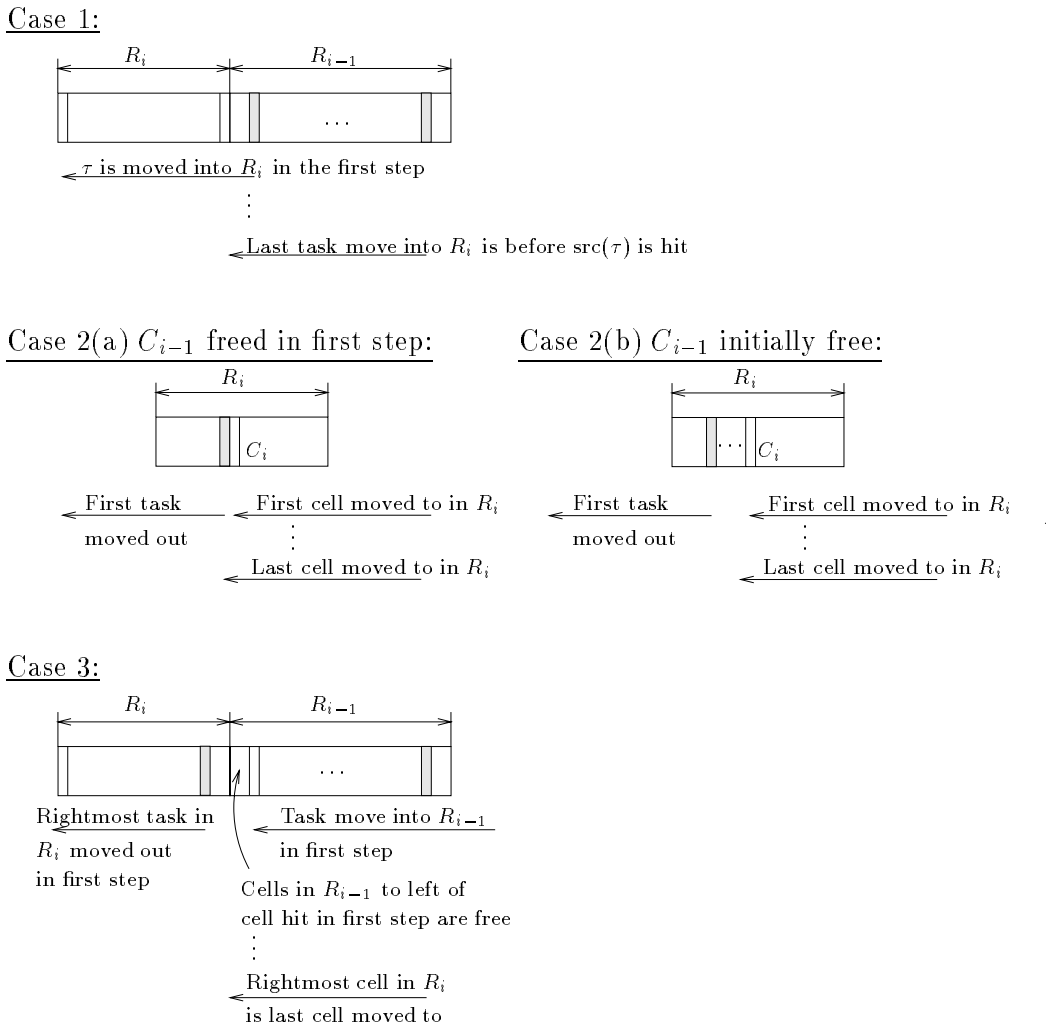


Figure 4.7: The destination of the last move into region R_i is either initially free, or it is freed in the first step.

Lemma 7 *Tasks are moved into each region R_i for $0 < i \leq r$ in cyclic order.*

Proof: Tasks are moved into region R_1, filling the cells in the region from leftmost to rightmost. When a task is hit it is moved in the following step into R_2, as shown

in Lemma 5. Region R_2 is therefore also filled from the leftmost cell to the rightmost by tasks arriving from R_1. Instruction 1(b) selects the rightmost task of region R_2 to be moved in the first step of the schedule. Its destination is the rightmost cell of R_3, following which R_3 is filled from the leftmost to the rightmost but one cell with tasks arriving from R_2.

Let the cells in region R_{i-1} with $i > 1$ be labelled $C_j, C_{j+1}, \ldots, C_{j+N_{i-2}-1}$. Let C_{j+k} with $0 \leq k \leq N_{i-2} - 1$ be the cell moved to in the first step of the schedule, and suppose that tasks move into R_{i-1} in cyclic order, i.e., the cells $C_{j+k+1}, C_{j+k+2}, \ldots, C_{j+N_{i-2}-1}$ are filled in that order in subsequent steps, followed by the cells $C_j, C_{j+1}, \ldots, C_{j+k-1}$. According to instruction 1(b) of the procedure, the rightmost task to the left of cell C_{j+k} is chosen to be moved in the first step of the schedule. If there is no task in R_{i-1} initially allocated to the left of C_{j+k}, then the rightmost task in R_i is selected, and R_i is filled from the leftmost to the rightmost cell in the region as tasks in R_{i-1} are hit.

On the other hand, suppose that the rightmost task to the left of C_{j+k} is the lth task from the left in region R_{i-1} and that it is allocated to cell C_{j+m} with $m < k$. Then the lth task in R_{i-1} is chosen to be moved in the first step of the schedule to the lth cell from the left in region R_i, the cell labelled $C_t = C_{j-N_{i-1}+l-1}$. The $N_{i-1} - l$ cells in R_i to the right of C_t are filled in subsequent steps as the tasks allocated to the right of C_{j+k-1} in R_{i-1} are hit. Similarly, those cells in R_i to the left of C_t are filled as the tasks to the left of C_{j+m} in R_{i-1} are hit. In either case, tasks are moved into R_i in cyclic order as well. ∎

Finally, it can be shown that the delay to each task is bounded by a constant number of cycles.

Theorem 9 *No task is suspended by procedure CULT for more than 3 steps. Given the assumptions of the model, this is optimal*

Proof: All tasks are switched out of context in the cycle before they are moved, get moved in the schedule step allocated them by the procedure, and are switched into context again in the following cycle. Since the destination of a move is either free when the task arrives or is occupied by a task that is switching out of context prior to being moved, each task is free to be switched into context in the step following its arrival at its destination. The total number of cycles in which a task could not execute therefore is 3. ∎

4.2.5 Compacting Arbitrary Length Tasks

Unfortunately, straightforward applications of the unit task length solution to the arbitrary task length problem are not likely to be satisfactory. While procedure *CULT* readily compacts tasks of arbitrary length in a linear array in the least possible number of steps, it may delay some tasks for $O(S_{\min})$ steps. For instance, see Figure 4.8. With the regions defined by counting the number of allocated cells rather than the number of tasks occupying a region, a task of length 2 whose left cell is the rightmost cell of region R_1 and whose right cell is the leftmost cell of R_0 would be suspended for $S_{\min} + 2$ steps were procedure *CULT* used to compact the tasks although as few as 4 steps are needed to move the task and switch contexts out and in again.

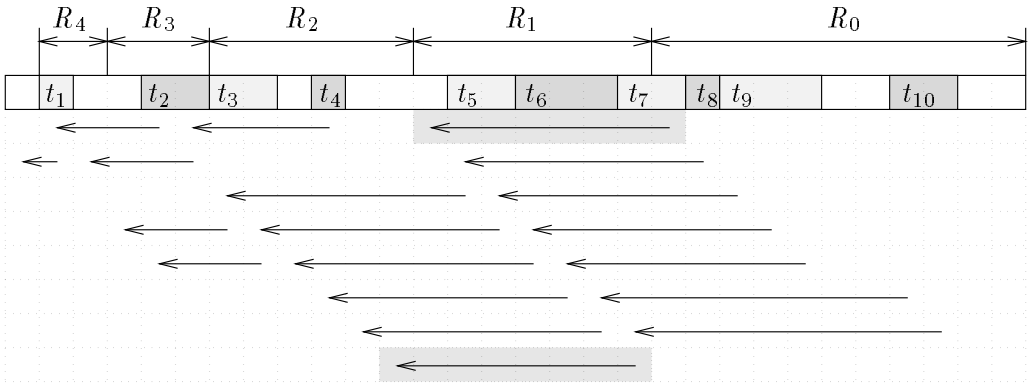


Figure 4.8: The compaction of arbitrary length tasks $t_1, \ldots, t_{10}$ using procedure *CULT* suspends t_7 for $S_{\min} + 2$ steps.

This section describes a heuristic algorithm for the left ordered compaction of arbitrary length tasks. The algorithm produces a schedule of length less than $2S_{\min}$ steps, which frees the allocation site in $S_{\min}$ steps, and delays tasks for less than $S_{\min}$ steps. The advantage of this algorithm over procedure *CULT* is that it attempts to minimize the maximum delay to the moving tasks by its choice of task to move first.

The left ordered compaction schedule is obtained by identifying sequences of task elements that can be moved in parallel. The sequences are defined so that the task elements within each sequence can be moved without affecting the moves of task elements in neighbouring sequences. Task elements within a sequence commence moving with the rightmost element in the first step of the schedule and proceed to move in decreasing address order until the leftmost element of the sequence has

been moved. The boundaries of the sequences are selected so as to minimize the maximum delay to tasks. The rightmost sequence may therefore not cover all of the task elements covered by the incoming task. If it does not, then the task elements contained in it do not commence moving until its rightmost element is free to do so without interfering with task element movements in the adjacent sequence to the left. The schedule is complete when the leftmost task elements of all sequences have been moved.

Figure 4.9 illustrates a schedule for a left ordered compaction instance. Sequence 1 covers tasks t_5, t_6, and t_7. These can be moved in parallel with those of sequence 2 (tasks t_2, t_3, and t_4) and sequence 3 (task t_1). The rightmost sequence, comprising tasks t_8, t_9, and t_{10}, does not include all the allocated cells assumed to be covered by the allocation site. The rightmost task element of the rightmost sequence is thus restrained from moving until the third step of the schedule.

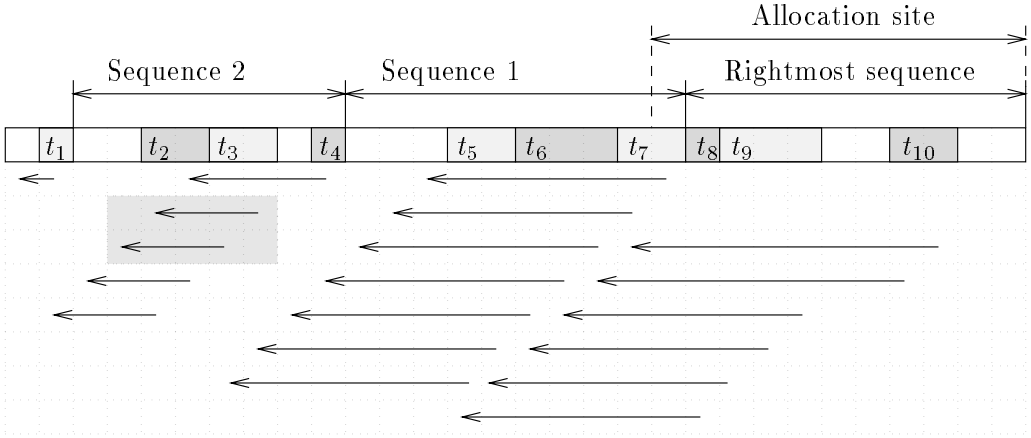


Figure 4.9: The left ordered compaction schedule found by procedure *CALT* delays t_2 for 2 steps.

Scheduling begins by computing a set of *candidate schedules*, each of which commences with the rightmost task element of one of the tasks that is covered or partially covered by the allocation site. From among these candidates, the schedule that minimizes the delay to moving tasks is chosen. If there are two or more such schedules, then one that minimizes the schedule length is selected. The procedure *CALT* of Figure 4.10 finds a left ordered compaction schedule that minimizes the delay to the incoming task and that attempts to minimize the delay to executing tasks while moving them in a regular fashion.

For each task t_k whose rightmost task element is covered by the allocation site, the sequences of a candidate schedule are identified by finding the set of task

Procedure **CompactArbitraryLengthTasks (CALT)**

Input The list of tasks to be compacted left, and for each task, its base, its size, and the distance it is to be moved.

Output A left ordered compaction schedule specifying for each time step those task elements that are to be moved.

begin

1. Initialize the minimum delay $\delta_{\min}$ and the minimum schedule length $S_{l_{\min}}$ to large values.
2. For each task t_k whose rightmost cell is covered by the incoming task:
   (a) Call procedure *GetCandidateScheduleSequences* to get the list of task elements to be moved in the first step of a schedule commencing with $t_k[l_k]$, the maximum delay to tasks $\delta_{\max}$, and the candidate schedule length S_l.
   (b) If ($\delta_{\max} < \delta_{\min}$ or ($\delta_{\max} = \delta_{\min}$ and $S_l < S_{l_{\min}}$)), then save the list of task elements, $\delta_{\min}$, and $S_{l_{\min}}$.
3. Derive the schedule for the sequences defined by the saved list of task elements.

end

Figure 4.10: Procedure *CompactArbitraryLengthTasks (CALT)*.

elements $\mathcal{E}$ that can be safely moved in parallel with $t_k[l_k]$. Each task element $e \in \mathcal{E}$ marks the right boundary of a sequence of task elements to its left which extends to but does not include the next element to the left in $\mathcal{E}$. As described below, the candidate schedule length and the maximum delay to a task can be computed while the sequences are being found. The schedule that minimizes the maximum delay to executing tasks and the total schedule length can thus be chosen once all candidate sequences have been computed.

The task elements contained in $\mathcal{E}$ are selected in the following way. Suppose that some task element $t_j[v]$ with $1 \leq j \leq n$ and $1 \leq v \leq l_j$ has been chosen to be moved in the first step. Consider where $t_j[v]$ is to be moved to so as to find the next task element to the left that can safely be moved in parallel with it on the bus. The possibilities are illustrated in Figure 4.11. There are three cases:

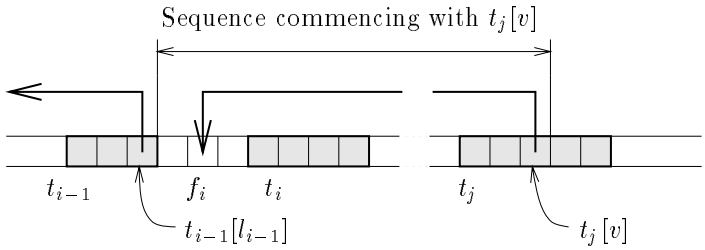
1. If $t_j[v]$ is moved to a free cell of some free block f_i, then $t_{i-1}[l_{i-1}]$, if it exists, can be moved to the left together with $t_j[v]$.

2. Should $t_j[v]$ be moved to a cell occupied by $t_i[u]$ with $d_i \leq l_i$, then no task element from tasks to the right of t_i are moved to the left of t_i. This fact makes it safe for the next task element to the left of $t_i[u]$ to be moved at the same time as $t_j[v]$ is moved. If $u > 1$, then $C_{b_{i,(u-1)}}$ writes to the left at the same time as $C_{b_{i,u}}$ reads from the right. If $u = 1$, then $t_{i-1}[l_{i-1}]$, if it exists, can be moved to the left together with $t_j[v]$.

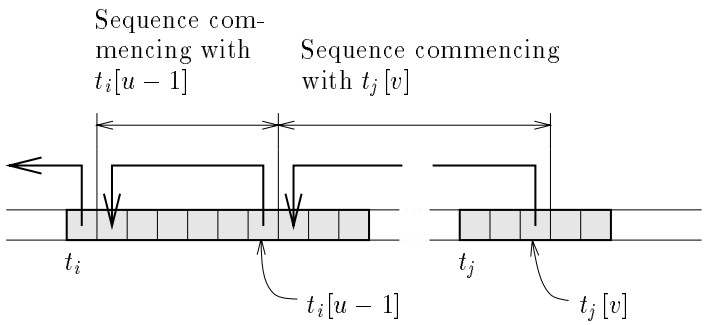
3. When $t_j[v]$ is moved to a cell occupied by $t_i[u]$ with $d_i > l_i$, then some task element from tasks to the right of t_i is moved to the left of t_i. Since the task elements of t_i to the right of and including $t_i[u]$ cannot move while the tasks $t_{i+1}, \ldots, t_j$ move to the left of $t_i[u]$, t_i must wait until after they have moved for it to be able to move in the least possible number of consecutive steps. In this case, $t_{i-1}[l_{i-1}]$, if it exists and needs to be moved, can be moved in parallel with $t_j[v]$. To compute the delay to task t_i, note that there are $v - 1$ task elements of t_j remaining to be moved before t_i must switch out of context. There are also $u - 1$ task elements of t_i plus a gap of $d_i - l_i$ cells to the left of t_i to be moved to by the tasks $t_{i+1}, \ldots, t_j$ before t_i can begin to move. The delay to task t_i therefore is $(u-1) + (d_i - l_i) - (v-1) = u - v + d_i - l_i$, which is the sum of the lengths $l_{i+1} + l_{i+2} + \ldots + l_{j-1}$.

The above sequence selection rule is repeated for the task element just found until it is not possible to choose another task element to the left that can be

Case 1. If $t_j[v]$ is moved to f_i, then $t_{i-1}[l_{i-1}]$ can move.



Case 2. If $t_j[v]$ is moved to $t_i[u]$ with $d_i \leq l_i$, then $t_i[u-1]$ can move.



Case 3. If $t_j[v]$ is moved to t_i with $d_i > l_i$, then $t_{i-1}[l_{i-1}]$ can move.

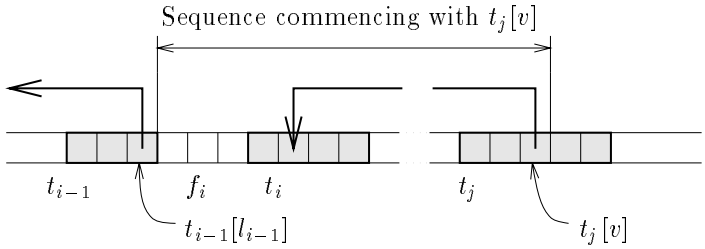


Figure 4.11: Identifying the task element to move in parallel with $t_j[v]$.

moved in parallel with $t_j[v]$. The task elements thus found form the rightmost task elements of the sequences for a candidate schedule commencing with $t_k[l_k]$ for some task t_k with $1 \leq k \leq n$ whose rightmost task element is covered by the incoming task.

The task elements to the right of $t_k[l_k]$, i.e., $t_{k+1}[1], \ldots, t_n[l_n]$ form a rightmost sequence that commences moving with $t_n[l_n]$ after $S_{\min} - \sum_{i=k+1}^{n} l_i$ steps so as not to conflict with the sequence commencing with $t_k[l_k]$.

The sequence selection rule is at the heart of step 2(b) in the candidate scheduling procedure of Figure 4.12.

Finally, expressions for the maximum schedule length and maximum delay to tasks can be derived. The results are a consequence of the sequence selection rule.

Lemma 8 *A task t_i whose leftmost task element is not covered by the incoming task is moved a distance of at most N_0 cells.*

Proof: Let t_j with $1 \leq j \leq n$ be the rightmost task whose leftmost task element is not covered by the incoming task. Then t_j moves $d_j \leq N_0$ cells to the left to make way for the N_0 task elements covered by the incoming task. Clearly t_{j-1} will need to move $d_{j-1} = d_j - f_j$ cells to the left to make room for t_j, and in general t_i with $1 \leq i < j$ moves $0 < d_j - \sum_{k=i+1}^{j} f_k \leq N_0$ cells to the left. ∎

Lemma 9 *At most $N_0 + 1$ allocated cells are spanned by a bus used to move a task element.*

Proof: Task elements not covered by the incoming task move at most N_0 cells to the left by Lemma 8. At most $N_0 + 1$ allocated cells are therefore spanned by the buses used to move them. Buses used to move task elements covered by the incoming task span at most $N_0 + 1$ allocated cells since, in the worst case, the N_0 cells immediately to the left of and abutting the allocation site are allocated before any task is moved. ∎

The number of steps S_l needed to complete a candidate schedule is given by the maximum number of task elements spanned by any sequence in the schedule.

Lemma 10 *The length S_l of a candidate schedule is $2N_0 - 1 < 2S_{\min}$ at most.*

Procedure **GetCandidateScheduleSequences**

Input The list of tasks to be compacted left. For each task, its base, its size, and the distance it is to be moved. A pointer to the task t_k whose rightmost task element $t_k[l_k]$, which is covered by the allocation site, is to be moved in the first step.

Output A list of task elements to be moved in parallel with $t_k[l_k]$, the maximum delay $\delta_{\max}$ to any task, and the candidate schedule length S_l.

begin

1. Initialize $\delta_{\max}$ and S_l to zero, and set $t_k[l_k]$ to be the task element $t_j[v]$ that is to be moved in the first step.

2. While a task element $t_j[v]$ remains to be moved:

   (a) Append the task element to the list of task elements to be moved in the first step.

   (b) Find the next task element $t_i[u]$ to the left of $t_j[v]$ that can be safely moved in parallel with it:

      i. If $t_j[v]$ is moved to t_i with $l_i < d_i$, then compute and save the maximum delay $u - v + d_i - l_i$ if it exceeds $\delta_{\max}$.

   (c) Compute and save the length of the sequence commencing with $t_j[v]$ if it exceeds S_l.

   (d) Set $t_j[v]$ to $t_i[u]$ if it exists.

end

Figure 4.12: Procedure *GetCandidateScheduleSequences*.

Proof: The rightmost sequence completes after $S_{\min}$ steps.

By Lemma 9, at most $N_0 + 1$ task elements are spanned by the sequences commencing with a task element $t_j[v]$ not covered by the incoming task in cases 1 and 2. Therefore these sequences are moved in at most $N_0 + 1$ steps. In case 3, at most $N_0 + 1$ task elements are spanned by the bus used to move $t_j[v]$, and by Lemma 8, since $l_i < d_i$, the length of t_i is at most $N_0 - 1$. Thus the length of the sequence commencing with $t_j[v]$ is at most $2N_0 - 1$ in this case. ∎

Lemma 11 *The maximum delay to an executing task is at most* $N_0 - 1 < S_{\min}$ *steps.*

Proof: In case 1, tasks $t_j, t_{j-1}, \ldots, t_i$ are moved one after another from rightmost task element to leftmost in the least possible number of steps since each moves further than its length. They can be switched out in the step before moving, therefore none of these are delayed.

The same is true for tasks $t_j, t_{j-1}, \ldots, t_{i+1}$ in case 2, and t_i commences moving as soon as it is moved to. All task elements spaced $d_i + 1$ apart in t_i move in the same step. In the first step $t_i[u - 1]$ moves. Indeed, all task elements in t_i whose index mod $(d_i + 1)$ is congruent with $(u - 1)$ mod $(d_i + 1)$ move in the first step, followed by all those with index one less and so on, until t_{i+1} finishes moving by writing to the cell allocated to $t_i[l_i - d_i + 1]$, when all task elements of t_i with an index mod $(d_i + 1)$ congruent with $(l_i - d_i)$ mod $(d_i + 1)$ move. Immediately thereafter, $t_i[l_i]$ and all elements congruent with l_i mod $(d_i + 1)$ move, followed by the next task elements to the left, until $t_i[u]$ and all elements congruent with u mod $(d_i + 1)$ are moved. Thus all task elements of t_i move in $d_i + 1$ consecutive steps and t_i is also not delayed.

It is possible that in case 3 the task t_i is delayed from moving since it must switch out of context to allow t_j to execute after $t_j[1]$ has been moved. In this case, t_i cannot begin to move until t_{i+1} finishes moving. By Lemma 9, at most $N_0 + 1$ task elements are spanned by the bus used to move $t_j[v]$. Since t_i may need to switch out of context in the step following the arrival of $t_j[v]$ to allow t_j to execute, t_i may need to wait $N_0 - 1$ steps before it can begin to move. ∎

It should be noted that the maximum delay could be halved by resuming the task t_j in case 3 some time after it arrives, that is, by resuming t_j at the midpoint

in the time interval between the last task element of t_j arriving and the first task element of t_i leaving.

The following theorem summarizes the above results.

Theorem 10 *There is a left ordered compaction schedule with less than* $2S_{\min}$ *steps that delays tasks by less than* $S_{\min}$ *steps. Moreover, such a schedule can be found in quadratic sequential time.*

Proof: The length of the schedule is less than $2S_{\min}$ steps as shown in Lemma 10, and it delays tasks less than $S_{\min}$ steps by Lemma 11. Generating the sequences for each candidate schedule necessitates scanning the list of tasks to be moved, and since the allocation site may cover all of the tasks, the time to find the shortest schedule that minimizes the delay to executing tasks is quadratic in the number of tasks to move. ∎

4.2.6 Final Remarks

This section considered moving tasks over a segmentable bus with a single port to each logic cell. This model is equivalent to a reconfigurable mesh [47] of height 1 with the constraint that a cell may either send or receive a task element in each step but not both. This constraint simplifies the hardware implementation at the cost of some scheduling complexity. Lifting the constraint simplifies the compaction of unit length tasks because a task that is initially allocated to a cell that is the destination for a task t can be moved in the step t arrives. However, the complexity of compacting arbitrarily long tasks appears to be unaffected by this hardware enhancement.

Allowing tasks to be orderly compacted towards both ends of the array can at best halve the time needed for the compaction without simplifying the scheduling complexity. Identifying optimal allocation sites is slightly more complicated than for left ordered compaction, but can still be done in polynomial time.

Arbitrary compactions, i.e., ones that allow the order of the tasks to be permuted, offer the possibility of reducing the number of tasks that need to be moved and may thus reduce the mean execution delay to tasks. However, they cannot allocate the waiting tasks more quickly than ordered compactions, and finding effective heuristics is challenging. Identifying an optimal allocation site is likely to be NP–hard, and scheduling the movements is more complicated when tasks are allowed to be moved past one another in opposite directions.

A fundamental difference between one– and two–dimensional compaction is that in two dimensions tasks to the right of the allocation site can be moved further than the maximum number of allocation site cells, in any single row, that are initially occupied. The scheduling results for one dimension do not therefore apply. In Section 4.1.1 the time to perform an ordered compaction of two–dimensional tasks over nearest neighbour links was found to be equal to the distance the leftmost task occupying the allocation site had to be moved to free the site. Figure 4.13 illustrates the point that this is also an upper bound for the length of a schedule to move the tasks were segmentable buses used instead. In Figure 4.13, tasks t_1 through t_{2k} must move to the right of the allocation site, whose boundary coincides with task t_{2k}. The ordered compaction protocol forces task t_{2k+1} to be moved $2k-1$ cells to the right. According to Lemma 1, $\min\{w(\mathrm{or}(t_{2k+1})), 2k\}$ steps are therefore needed to move task t_{2k+1} although only 3 steps are needed to free the allocation site with segmentable buses.

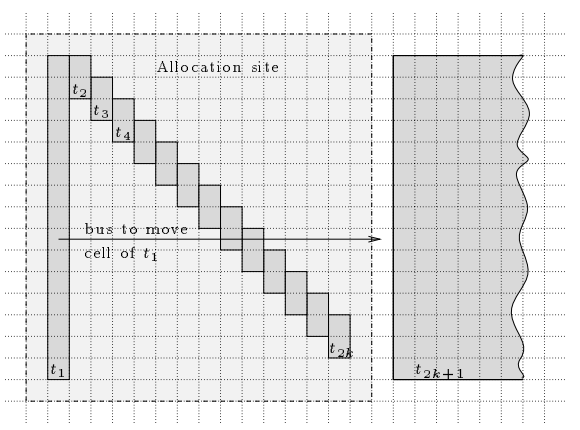


Figure 4.13: Example illustrating upper bound for schedule to compact two–dimensional tasks using segmentable buses.

The use of segmentable buses to compact two–dimensional tasks offers the possibility of moving individual tasks more quickly than they can be moved over nearest neighbour links. Nevertheless, finding schedules to do so appears to be difficult — certainly at least as difficult as minimizing the delays to one–dimensional tasks of arbitrary length, which is conjectured to be NP–hard.

The hardware and control required to support the use of segmentable buses on an FPGA is an as yet unresearched problem.

4.3 Conclusions

Moving tasks on–chip avoids the I/O bottleneck at the border of the chip, thereby lifting the constraint that forces tasks to be reloaded sequentially. The large available bisection width allows several tasks to be moved in parallel when task elements are moved over inter–cell connections. In addition, moving tasks on–chip allows I/O to tasks not being moved to continue without interruption. There are many benefits of moving tasks on–chip. The time needed to move a task is reduced, and the overall schedule length is shortened. The execution delays to the moving tasks are thus reduced, and shorter schedules result in higher allocation rates.

To gain insights for scheduling two–dimensional ordered compactions over segmentable buses, the one–dimensional problem was considered. One–dimensional ordered compaction scheduling over a segmentable bus appears to be NP–hard. However, an optimal algorithm for compacting unit length tasks, and a heuristic which bounds the delay to tasks of arbitrary length were described. Nevertheless, there is scope for simplifying and improving upon the results.

While the use of segmentable buses offers the potential for reducing the interruption to individual tasks and for increasing the rate at which arbitrary rearrangements can be performed, little progress has been made towards describing these methods in two dimensions. There is, therefore, considerable scope for further research, although the effort may not be rewarded with significant improvements beyond those that can be gained from the use of nearest neighbour links.

Chapter 5

Experimental Results

This chapter presents and analyzes the results of simulation experiments to assess the performance of the techniques discussed in Chapters 3 and 4. First, the performance of the approximate FPGA rearrangement scheduling method is examined to determine the appropriate amount of lookahead needed for path cost estimation. Partial rearrangements are then evaluated for a range of input parameters. Moving tasks by reloading them is assessed first — the effects of task load, configuration delay, and task size are examined. In the third section, the benefits of moving tasks on–chip are evaluated. The fourth section compares the results with those obtained by Youn et al. for the mesh architecture. The chapter finishes with a brief summary of the main conclusions and suggestions for algorithmic refinements that could lead to better performance.

5.1 Performance of FPGA Rearrangement Scheduling

In this section, the scheduling performance of the approximate FPGA rearrangement scheduling algorithm *AFRS* is compared with that of the exact *EFRS* algorithm. The algorithms are assessed by comparing the maximum schedule delay and the number of states expanded on identical problem instances. The quality of a schedule produced by *AFRS* is gauged by calculating a relative performance figure which is given by the ratio of the maximum schedule delay (amount of time a task was suspended and not moving) of the solution found by *AFRS* to the maximum schedule delay of the optimal solution found by *EFRS*.

A set of randomly generated problem instances were presented to each of the algorithms under investigation to compare the scheduling outcomes. The pa-

rameters used to specify a randomly generated problem instance were the number of tasks in the test, the maximum side length of a task, and the likelihood of intersection between the tasks. Task side lengths were uniformly distributed independent random variables, and the likelihood of task t having more than l intersections was given by an exponentially decreasing probability density function, $P[|I(t)| \geq l] = x^l$ with $0 < x < 1$.

The algorithms were implemented with two optimizations to reduce the number of expanded states: all executing tasks with empty intersection sets were relocated when the suspended task list was empty; and the suspended tasks were relocated according to Lemma 2 of Chapter 3 when no executing tasks remained to be moved. *EFRS* was aborted when the size of the open state list exceeded 4MB to prevent it from using exponential space and time to find a solution. Note that the number of states examined by the cost estimator were not counted in the assessment.

Tests were carried out with solution path cost estimators using one and two states of lookahead. The results for one– and two–state lookahead are presented next, followed by a brief discussion of the findings.

5.1.1 One–State Lookahead

The performance of the exact and approximate FPGA rearrangement scheduling algorithms was compared using one–state lookahead on 23,040 randomly generated problem instances. The number of tasks in a problem instance and the maximum task side lengths ranged from 5 up to 20, and the base x for extending the intersection set ranged from 0.1 to 0.9. For each parameter combination, 10 tests were performed.

The significant findings were:

- The relative performance of the approximate algorithm is summarized in Table 5.1. The table lists the number of instances achieving a specified relative performance figure or better. In 95% of instances a relative performance of 1.5 or better was achieved. However, a relative performance as high as 10.4 was observed.

- The exact method expanded less states than the approximate method in 74.0% of instances.

- The exact algorithm failed to find a solution within 4MB of working storage in 7.9% of instances.

Relative Performance Achieved	Number of Instances	Percentage of Total
1.0	17,200	81.0
1.1	18,225	85.6
1.2	19,021	89.6
1.3	19,566	92.2
1.4	19,976	94.1
1.5	20,286	95.6
2.0	21,000	98.9
3.0	21,185	99.8
4.0	21,207	99.9
5.0	21,219	99.9
10.0	21,230	100.0
10.4	21,231	100.0

Table 5.1: Relative performance of algorithm AFRS with a lookahead of one state.

5.1.2 Two–State Lookahead

The performance of the exact and approximate algorithms using one–state lookahead (*EFRS–1* and *AFRS–1* respectively) was compared with algorithms using a lookahead of two states (*EFRS–2* and *AFRS–2* respectively).

Each algorithm was tested on the same set of 2,560 randomly generated FPGA compaction instances. The parameters used to generate the problem set were selected to provide a mix of trivial and non–trivial intersection patterns. Between 11 and 14 tasks were generated, the maximum task side length ranged from 5 up to 20, and the base for extending the intersection set ranged from 0.5 to 0.8.

The findings were:

- Table 5.2 summarizes the relative performance of *AFRS–1* and *AFRS–2*. For each relative performance figure, the table lists the number of instances which achieved the figure or better. In 95% of instances, *AFRS–1* achieved a relative performance of 2.0 or better and *AFRS–2* achieved a relative performance of 1.5 or better.

- Of 2,458 tests solved by both *EFRS–1* and *EFRS–2*, in 28.2% of cases the relative performance of *AFRS–2* was less (better) than that of *AFRS–1*. The mean reduction in relative performance from *AFRS–1* to *AFRS–2* was 17.2% with a standard deviation of 0.5%. In 5.1% of cases the relative performance of *AFRS–2* was greater (worse) than that of *AFRS–1*. The mean increase in

relative performance from *AFRS–1* to *AFRS–2* was 15.5% with a standard deviation of 1.8%.

- Of 2,560 tests, *EFRS–2* expanded less states than *AFRS–2* in 54.8% of cases, whereas *EFRS–1* expanded less states than *AFRS–1* in 51.0% of cases.
- Of 2,560 tests, 2.5% were unsolved by *EFRS–2* within 4MB of open state information. *EFRS–1* was unable to solve an additional 1.5%, or 4.0% of the total.

Relative Performance Achieved	Number of *AFRS–1* Instances	Percentage of *AFRS–1* Instances	Number of *AFRS–2* Instances	Percentage of *AFRS–2* Instances
1.0	1,307	53.2	1,551	62.2
1.1	1,573	64.0	1,881	75.4
1.2	1,797	73.1	2,099	84.1
1.3	2,017	82.1	2,262	90.7
1.4	2,150	87.5	2,359	94.6
1.5	2,245	91.3	2,420	97.0
2.0	2,408	97.8	2,486	99.6
2.5	2,445	99.5	2,495	100.0
3.0	2,455	99.9		
4.0	2,457	100.0		
5.0	2,458	100.0		

Table 5.2: Relative performance of algorithm AFRS with lookahead of one and two states.

5.1.3 Discussion

The approximate algorithm appeared to perform well on these small instances since it achieved less than twice the maximum schedule delay of an optimal solution in more than 95% of trials. Considering the exact method failed to obtain a solution within 4MB of working storage in approximately 5% of instances, this result appears good indeed. Two–state lookahead appears to offer only a slight performance improvement over one–state lookahead at the cost of adding a linear factor to the scheduling complexity. The speed of the exact method suggests a practical approach to frequently obtaining a good solution quickly: if a solution is not found after expanding up to $n(n-1)$ states with the exact algorithm, then use the approximate method.

5.2 Performance of Off–Chip Rearrangements

This section reports on simulation experiments conducted to gauge the performance of partial rearrangement strategies that reload tasks from off–chip to perform task movements. First, the operation of the simulator and its components are outlined. The experiments conducted and the main findings are then briefly summarized before the results are reported on and discussed in detail.

5.2.1 Overview of Simulator

Simulation is an effective means by which various approaches to allocating and rearranging tasks may be compared. In outline, the simulator's operation was as follows.

The simulator generated a random set of tasks within specified parameters and placed them in arrival order into a first in, first out queue. The task at the head of the queue was loaded onto the FPGA when a site for it was found using the allocation method under test. The task remained allocated until its service period had finished whereupon it was removed from the FPGA. Time–stamping the significant events in a task's life cycle allowed various performance metrics for the task set as a whole to be calculated. A more detailed description of the simulator follows.

Each simulation run commenced with the generation of 10,000 independent tasks that were characterized by 4 uniformly distributed independent random variables. A task was represented by a rotatable rectangular request for a subarray of cells with independently chosen side lengths $x \times y = \mathrm{U}_1(1, L) \times \mathrm{U}_2(1, L)$, with maximum side length L. The period of time between task arrivals, the inter–task arrival period, was $\mathrm{U}_3(1, P)$ time units with P a specified maximum. The service period of a task was chosen from $\mathrm{U}_4(1, 1000)$ time units. The size of the task set was chosen to provide a reasonably accurate measure of steady state behaviour. Task sets were generated using uniformly distributed random variables because expected workload characteristics were not known; it is also more straightforward to draw conclusions from the results based on uniformly distributed data than from data based on more complex distributions. Since task sets were synthetic, all measurements of time were scaled to a common time unit (tu). The results are independent of the magnitude of a time unit.

The simulated FPGA had its size fixed at 64×64 cells. The configuration

delay per cell CD was varied. For the results in this section, it was assumed that tasks were loaded or reconfigured via a single I/O port. The time to load or reload a task was therefore $x \times y \times CD$ time units, and the mean configuration delay was given by the product of the mean task size and the configuration delay per cell.

Three allocation methods were compared. They were:

First fit [71] allocated the waiting task to the bottom–leftmost block of free cells large enough to satisfy the request. Both orientations of a waiting task were tried as soon as the previous task finished loading and then following each deallocation until allocation was successful. All subsequent pending tasks were prevented from advancing in the pending task queue until allocation succeeded. First fit is a good standard for comparison because it has complete recognition capability — it always finds an allocation site when one exists.

Local repacking attempted allocating the waiting task by locally repacking a subset of the executing tasks whenever first fit failed. Local repacking was tried with both orientations of the waiting task and both orientations of the subarray being repacked. Sleator's algorithm [62] was used to attempt a repacking of all the tasks partially or completely intersected by the subarray. A two–state lookahead cost estimator was used to perform the scheduling.

Ordered compaction attempted allocating the waiting task by orderly compacting a subset of the executing tasks whenever first fit failed. Ordered compaction was attempted in each compass direction with both orientations of the waiting task. Scheduling tasks as they were discovered in a depth–first traversal of the visibility graph of executing tasks delayed them for the minimum amount of time needed to move them and loaded the waiting task last of all.

Note that instead of searching for minimum cost allocation sites, simulations of local repacking and ordered compaction allocated at the first feasible site found. Since service periods are assumed not to be known, partial rearrangements were not aborted if the waiting task could have been allocated earlier, following task completions that would have occurred during the rearrangement process. The simulations did not account for the time required to find a bottom–left allocation site or to identify or schedule a feasible task rearrangement.

The simulator recorded the time a task arrived, the time the allocation for a task commenced, the time the task commenced loading, and the time the

task finished processing (accounting for delays due to task movements). From these fundamental quantities the following performance measures were computed:

Mean allocation delay: the mean over all tasks of the time between the allocation for a task commencing and the loading of the task commencing.

Mean queue delay: the mean over all tasks of the time between a task arriving and the allocation of the task commencing.

Mean response time: the mean over all tasks of the time between a task arriving and the task finishing processing.

Utilization: the mean amount of time an FPGA cell spent executing tasks as a percentage of the time needed to finish processing all tasks.

A further performance indicator was derived from these aggregate measures:

Mean execution delay: the mean over all tasks of the time a task was delayed from executing as a consequence of being moved.

5.2.2 Overview of Experiments

Three experiments were conducted to compare the performance of the different allocation methods. An experiment consisted of a specified number of runs for a fixed set of parameters: maximum task side length L; maximum inter–task arrival period P; and configuration delay CD. The results of 10 runs were averaged to reduce the uncertainty in the result. The experiments were designed to investigate the following effects.

1. The effect on performance of varying load with nominal configuration delay.

   Performance was measured for a configuration delay of 1/1000 time unit per cell as the maximum inter–task arrival period P was varied from 20 to 1,000 time units. The maximum side length L was fixed at 32 cells.

   When task loads were heavy, rearranging executing tasks significantly reduced allocation delays without causing significant execution delays.

   When task loads were light, partial rearrangements were not required because arriving tasks were easily accommodated.

2. The effect on performance of varying the configuration delay at different system loads.

Performance was measured at maximum inter–task arrival periods of 40 and 120 time units, which corresponded respectively to a heavy load, for which the FPGA was saturated with work, and a medium load, for which it was just coming out of saturation. The configuration delay was varied from the nominal level of 1/250 time unit per cell to the extreme level of 2.2 time units per cell. The maximum side length L was fixed at 32 cells.

Local repacking began performing worse than first fit at very low configuration delays because of long delays to executing tasks. By contrast, ordered compaction was able to sustain longer configuration delays because of shorter execution delays.

3. The dependency of performance upon task size at saturation with nominal configuration delay.

   The maximum task side length L was varied from 8 to 64 cells, while the maximum inter–task arrival period was fixed at 1 time unit, and the configuration delay was set to 1/1000 time unit per cell.

   Performance benefits due to rearrangements increased as maximally sized tasks grew to cover 1/4 of the array. Performance benefits decreased with further increases in size. Superior performance of local repacking when tasks are small reflect the benefit of collecting free space in two dimensions. The superiority of ordered compaction when tasks are large highlights the need for better two–dimensional packing heuristics.

Detailed results of the experiments and their discussion follow.

5.2.3 Effect of System Load on Allocation Performance

Figures 5.1(a) through 5.1(d) plot the allocation performance as the system load was varied by altering the inter–task arrival period. Three distinct regions are evident in all curves. From the left, these correspond to operating regions where the FPGA is saturated with work, where the FPGA is coming out of saturation, and finally the unsaturated region.

At small inter–task arrival periods, tasks arrive more quickly than they can be processed by the FPGA. The FPGA consequently saturates with work, meaning tasks need to wait before they can be allocated. While the next request is frequently blocked, it is sometimes possible to combine the free resources to satisfy the next

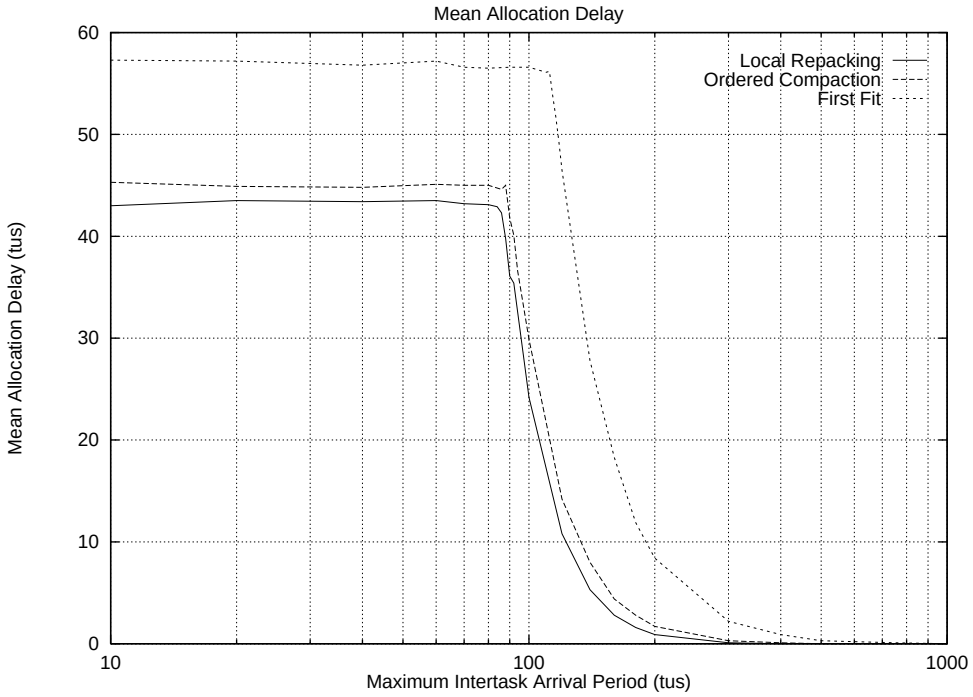


Figure 5.1: (a) Mean allocation delay for local repacking, ordered compaction, and first fit as the maximum inter–task arrival period was varied.

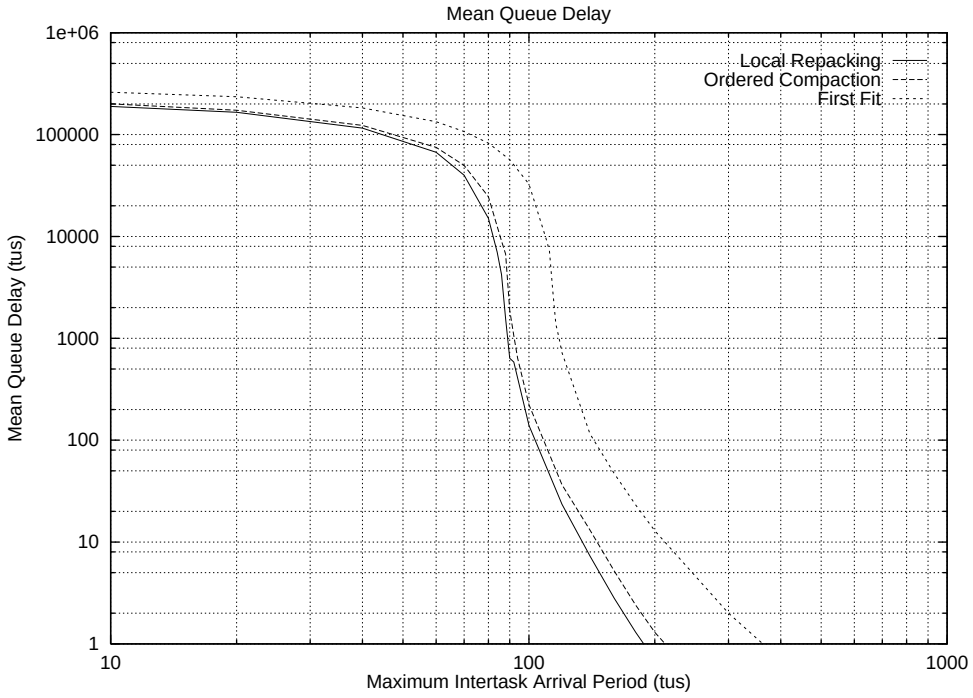


Figure 5.1: (b) Mean queue delay for local repacking, ordered compaction, and first fit as the maximum inter–task arrival period was varied.

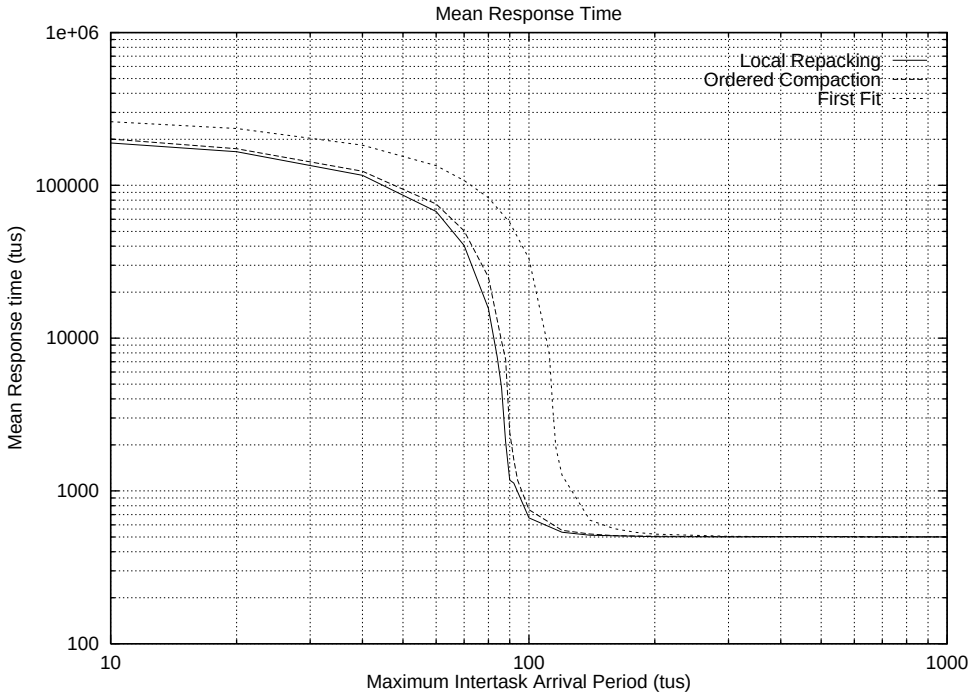


Figure 5.1: (c) Mean response time for local repacking, ordered compaction, and first fit as the maximum inter–task arrival period was varied.

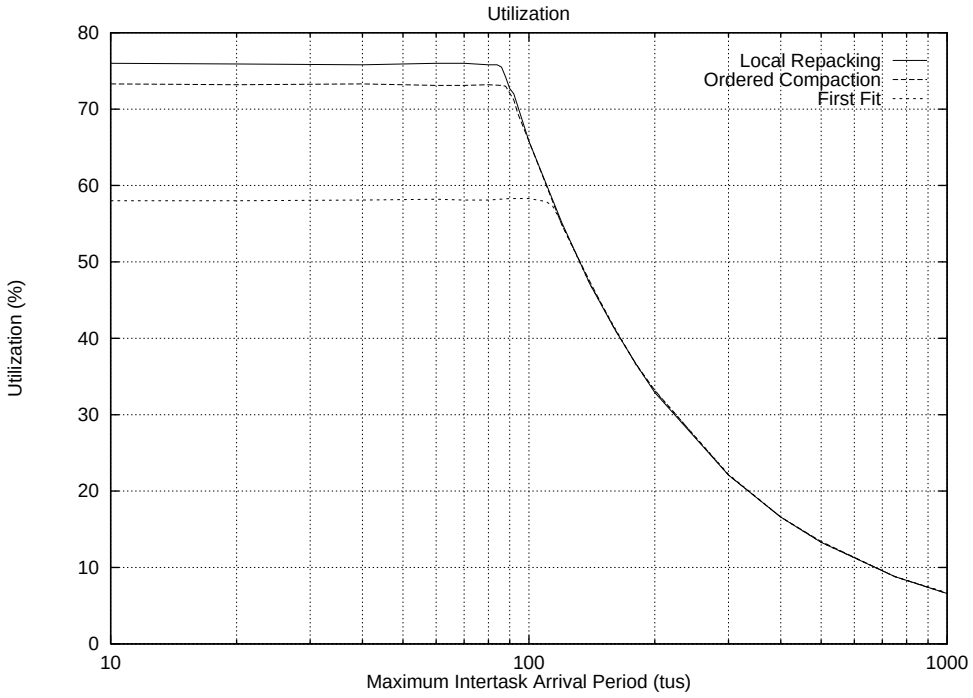


Figure 5.1: (d) Utilization for local repacking, ordered compaction, and first fit as the maximum inter–task arrival period was varied.

request by rearranging the executing tasks. Local repacking and ordered compaction therefore lead to a reduction in the mean allocation delay at saturation.

The mean allocation delay at saturation depends upon the size of the chip, the task size and service period distributions, the ability of the allocation method to find allocation sites, and the time needed to load tasks. Since work is always queued up, the rate at which tasks can be allocated is constant irrespective of the inter–task arrival period. The mean allocation delays at saturation for the first fit, ordered compaction, and local repacking allocation methods were 57.2, 44.9 and 43.5 time units respectively. Local repacking was therefore almost 24% quicker at allocating a task than first fit and over 3% quicker than ordered compaction

As the mean inter–task arrival period (half the maximum inter–task arrival period) increases beyond the mean allocation delay at saturation, tasks begin to arrive less frequently than they can be accommodated on the FPGA, and the chip quickly comes out of saturation. By lowering the mean allocation delay at saturation, partial rearrangements therefore lower the inter–task arrival periods at which FPGAs saturate. Said another way, partial rearrangements increase the load–bearing capacity of the system.

The mean allocation delay dropped below 1 time unit at maximum inter–task arrival periods of 200, 300 and 400 time units respectively for local repacking, ordered compaction, and first fit. When the FPGA is no longer saturated with work, it is likely that suitable free blocks can be found for tasks as they arrive without needing to rearrange executing tasks. At low loads there is therefore no benefit from partially rearranging the tasks.

In the saturated region, the queue delay to the kth task is approximately k times the difference between the mean allocation delay at saturation and the mean inter–task arrival period. Thus the mean queue delay in saturation is approximately $(\bar{a} - \bar{p}) \times (n + 1)/2$ where $\bar{a}$ is the mean allocation delay, $\bar{p}$ is the mean inter–task arrival period, and n is the number of tasks. Since the allocation delay drops to zero as the FPGA comes out of saturation, so too does the mean queue delay.

The mean response time is given by the sum of the mean queue delay, allocation delay, configuration delay, service period, and the time needed for partial rearrangements. At low configuration delays per cell, the time to load a task and the time to rearrange tasks is negligible. At saturation, the mean queue delay therefore dominates, and out of saturation, the queue delay and allocation delay become negligible, thus the mean service period dominates.

Utilization is the ratio of cell usage to cell capacity. It is given by the formula

$$\text{utilization} = 100 \times \frac{\sum_{i=1}^{n} e_i \times s_i}{\text{FPGA size} \times \max\{f_i : 1 \leq i \leq n\}}$$

where n is the number of tasks processed, e_i is the execution time (service period), s_i is the size, and f_i is the completion time of the ith task. For a given task set, the numerator is constant. In the saturated region, an estimate for the completion time is given by the mean allocation delay multiplied by the number of tasks, and thus the utilization is constant. At saturation, the utilization can be approximated by multiplying the mean task size by the mean service period, and dividing the result by the FPGA size and the mean allocation delay. This model predicts values approximately 5% lower than the utilization of 58.0%, 73.2%, and 75.9% observed for first fit, ordered compaction, and local repacking respectively. Further investigation indicated that the gaps between the predicted and the observed values are mainly due to deviations in mean task size from the expected value.

When tasks arrive less quickly on average than they can be accommodated, the completion time of the last task to finish depends upon its arrival time. As the FPGA comes out of saturation, a drop in utilization proportional to the rise in the inter–task arrival period is therefore observed.

From the above discussion it is apparent that the performance metrics are interrelated. Given one, the others can be derived knowing the parameters used in the experiment. In subsequent discussions, the focus will therefore concentrate on the effect of varying a parameter on the mean allocation delay alone because it is the factor most immediately influenced by the performance of the allocation method when rearrangements are carried out.

5.2.4 Effect of Configuration Delay on Allocation Performance

The effect of varying the configuration delay was examined at saturation and coming out of saturation. Figures 5.2(a) and (b) illustrate the performance at a maximum inter–task arrival period of 40 time units as the configuration delay per cell was increased from 1/250 time unit to 2.2 time units. This range corresponds approximately to a mean configuration delay per task of between 1 and 600 time units. Recall that the mean service period is only 500 time units.

At mean configuration delays below 1% of the mean service period, the performance benefits of local repacking and ordered compaction are similar to those observed at saturation with negligible configuration delay. However, local repacking

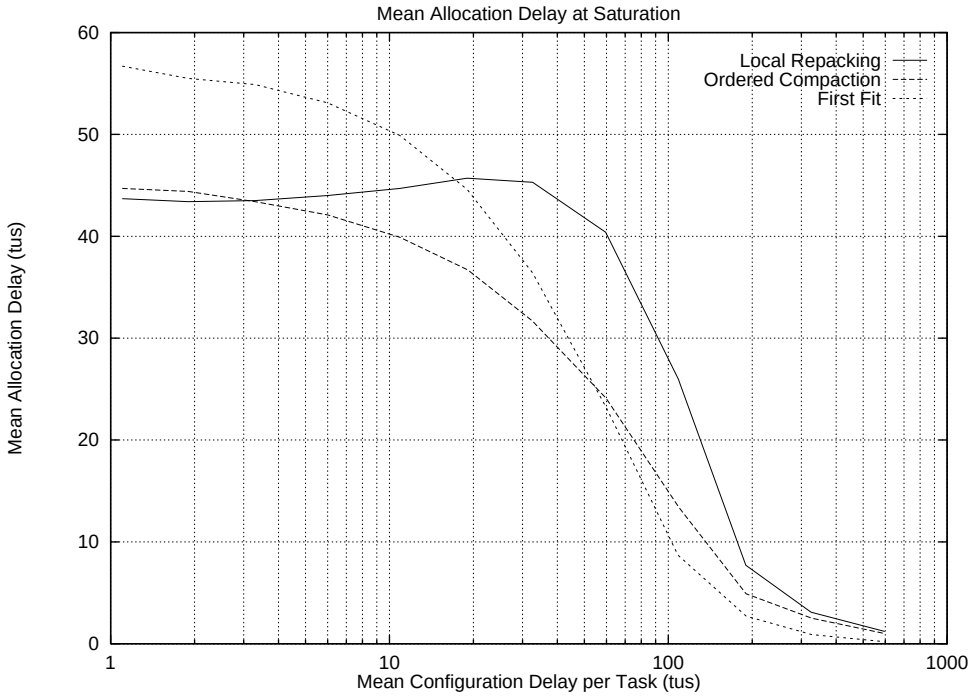


Figure 5.2: (a) Mean allocation delay at saturation for local repacking, ordered compaction, and first fit as the mean configuration delay per task was varied.

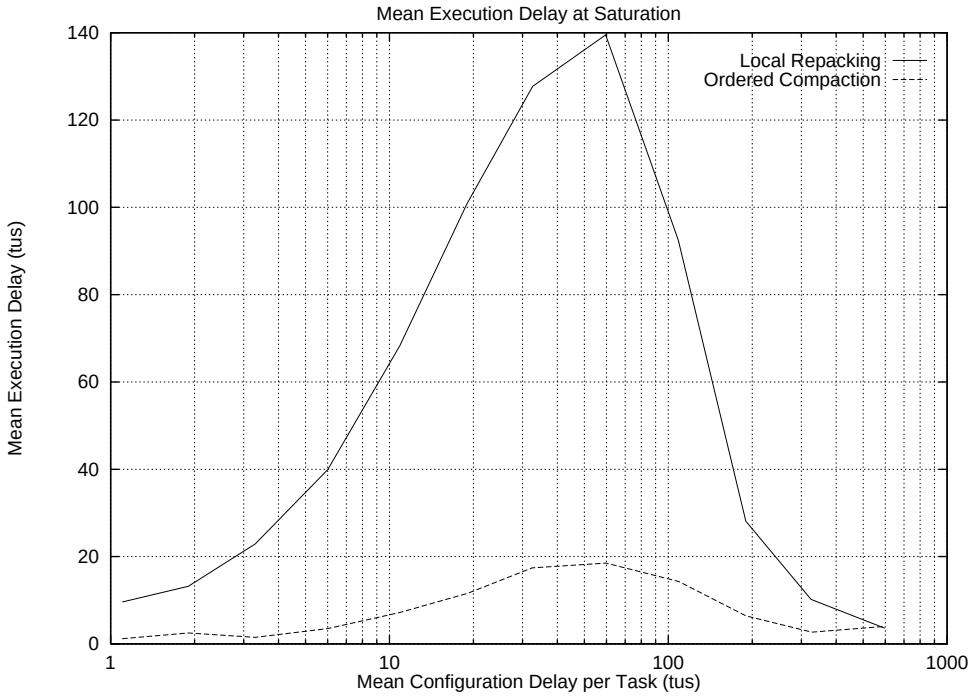


Figure 5.2: (b) Mean execution delay at saturation for local repacking and ordered compaction as the mean configuration delay per task was varied.

begins to perform worse than ordered compaction at relatively low configuration delays. Two factors probably contribute to this reversal. First, the delays to moving tasks are larger for local repacking than for ordered compaction because they are scheduled less optimally. Second, local repacking readily rearranges more tasks than ordered compaction because the feasible rearrangements considered are less constrained.

As the configuration delay rises, the time to load a task increases and the allocation delay for first fit falls because more free blocks become available due to tasks finishing before allocation of the next task begins. Unfortunately, by committing to rearranging, local repacking and ordered compaction retard allocation rates relative to first fit because the I/O port is tied up reloading moving tasks. This effect limits the usefulness of partial rearrangement by off–chip task movements to applications where the configuration delays are low relative to the service periods. At long configuration delays, the increased likelihood of finding allocation sites without the need to rearrange executing tasks reduces delays to waiting and executing tasks.

While the shape of the mean execution delay curve is explicable, the magnitude of the maximum, and the reason why it should occur at a mean configuration delay of 60 time units are yet to be explained. At mean configuration delays below 100 tus, the plot suggests that local repacking relocates each task multiple times.

Allocation performance was also examined at a maximum inter–task arrival period of 120 time units, which corresponds to a load level not quite high enough to saturate the FPGA. The results appear plotted in Figures 5.3(a) and (b).

Despite the much larger performance gap between local repacking and first fit, the points where the curves cross over occur at similar configuration delays.

The similarity between the execution delay curves of Figures 5.2(b) and 5.3(b) is yet to be explained.

5.2.5 Effect of Task Size on Allocation Performance

Figure 5.4 plots the mean allocation delay at saturation as the maximum task size is increased to fill the FPGA.

The plot shows that tasks were better allocated using local repacking when the longest task side was less than half the array side length. However, they were more easily allocated by ordered compaction when larger tasks occurred.

Free fragments trapped between small tasks are presumably easily gathered in two dimensions by repacking, but cannot necessarily be gathered along one

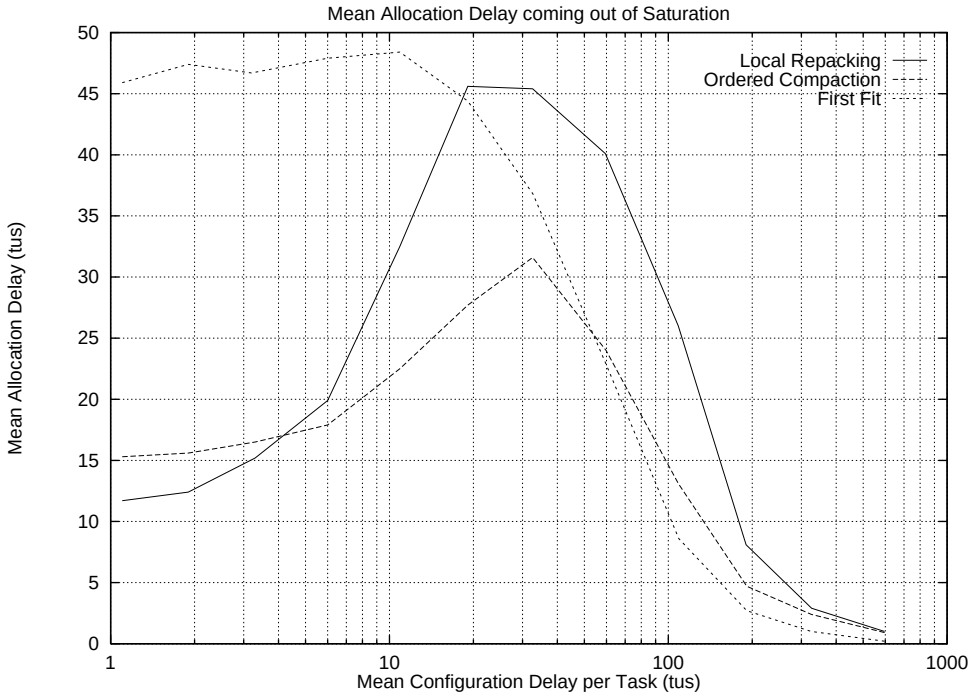


Figure 5.3: (a) Mean allocation delay coming out of saturation for local repacking, ordered compaction, and first fit as the mean configuration delay per task was varied.

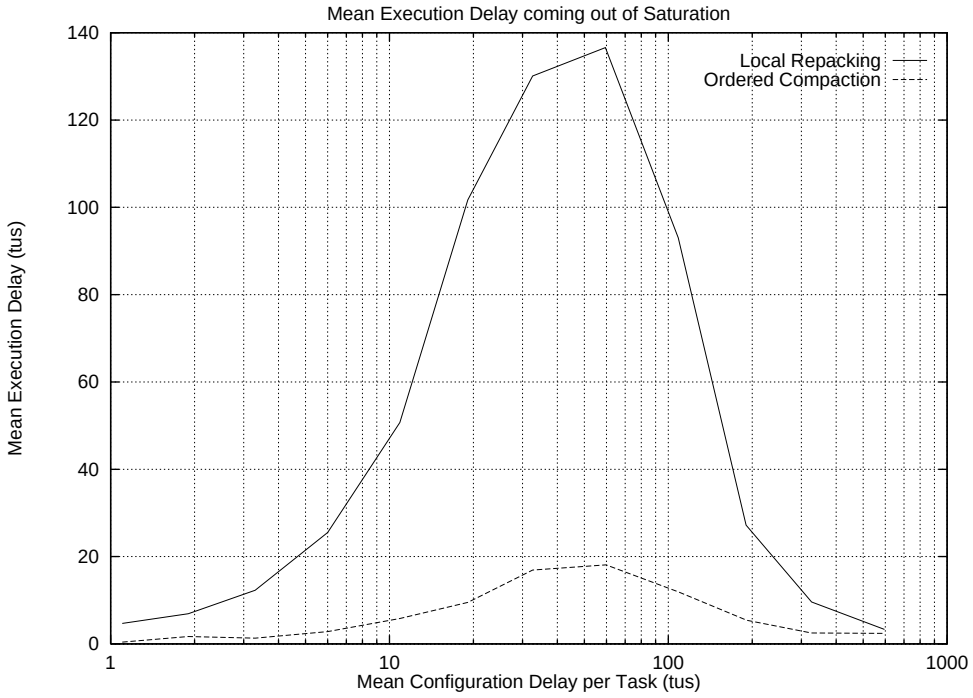


Figure 5.3: (b) Mean execution delay coming out of saturation for local repacking and ordered compaction as the mean configuration delay per task was varied.

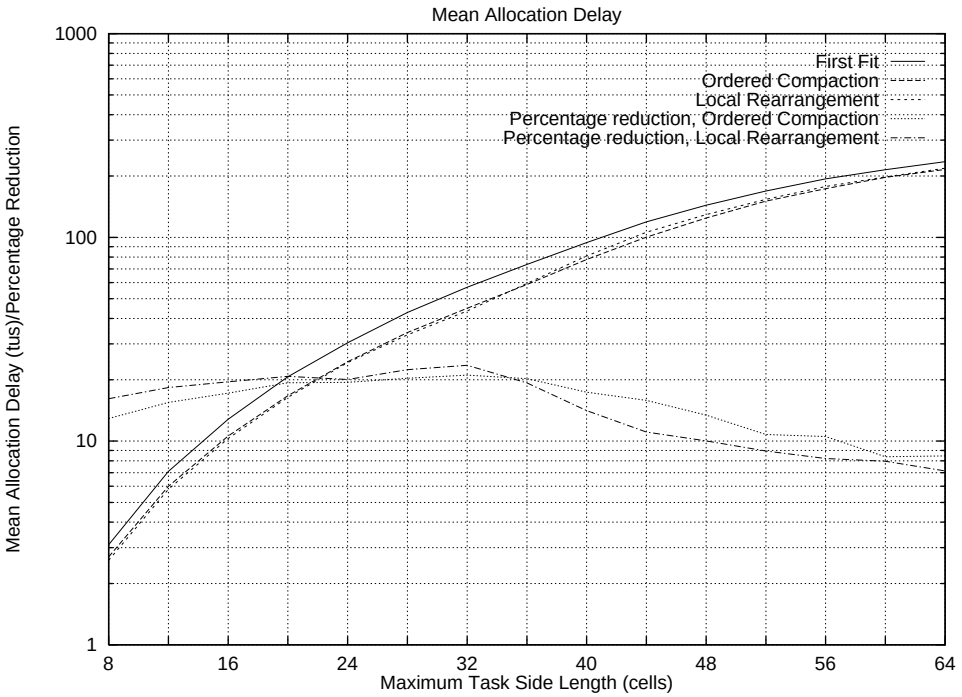


Figure 5.4: Mean allocation delay for local repacking, ordered compaction, and first fit as the maximum task size was varied.

dimension by ordered compaction. When task sizes are small therefore, arbitrary rearrangements ought to be used.

Although the results for ordered compaction indicate space was available, Sleator's method frequently failed to repack the tasks when large tasks were present. A possible explanation for this observation is that large regions of the array were left unused when the tasks wider than half the array were stacked on top of one another and those tasks that occupied these regions before could not be squeezed into the free space remaining between the top of the stack and the top of the array. Better packing heuristics ought therefore be used when large tasks need to be rearranged.

5.3 Performance of On–Chip Rearrangements

In this section, the performance of ordered compaction using the nearest neighbour links between FPGA cells to move tasks is compared with the performance of orderly compacting by reloading the moving tasks. The use of segmented buses is not reported upon because current methods do not provide significant additional performance improvements due to the infrequency with which tasks are moved beyond their boundaries. For a report on an experimental assessment of a

one–dimensional ordered compaction algorithm, see [23]. Two–dimensional ordered compaction using segmented buses was investigated and is reported upon in [19].

The simulator described in the previous section was used with two new allocation methods:

Cost free ordered compaction attempted allocating the waiting task by orderly compacting a subset of the executing tasks whenever first fit failed. Ordered compaction was attempted in each compass direction with both orientations of the waiting task. Tasks were moved instantaneously without accounting for compaction costs.

Nearest neighbour ordered compaction attempted allocating the waiting task by orderly compacting a subset of the executing tasks whenever first fit failed. When a feasible rearrangement of the executing tasks was discovered, moving tasks were simultaneously suspended and moved over nearest neighbour links to their destinations where they were resumed upon their arrival.

The time needed to move a task element from one cell to a neighbour over a link is referred to as the link delay LD. The link delay used in these experiments was assumed to be equivalent in length to the configuration delay per cell CD. This assumption is made although slightly more data needs to be communicated because the link is faster than I/O from off chip. Thus to move a task t with $\text{or}(t) = (x, y)$ a distance of d cells to the right requires $dLD = dCD$ time units whereas to move it by reloading requires $x \times y \times CD$ time units.

At low configuration or link delays, there would be little difference in performance between an ordered compaction allocation method that moves tasks over nearest neighbour links and one that moves tasks by reloading them. Thus it is of interest to determine the effect on performance as the configuration delay rises.

5.3.1 Effect of Configuration Delay on Allocation Performance

The effect of varying the configuration delay on ordered compaction when tasks are moved on–chip was compared with the effect on first fit and compaction when tasks are reloaded. The results of Section 5.2.4 for first fit and ordered compaction by reloading were used as a basis for the comparison. Performance was examined at saturation, at a maximum inter–task arrival period of 40 time units (Figures 5.5(a) and (b)), and as the FPGA was coming out of saturation, at a maximum inter–task arrival period of 120 time units (Figures 5.6(a) and (b)).

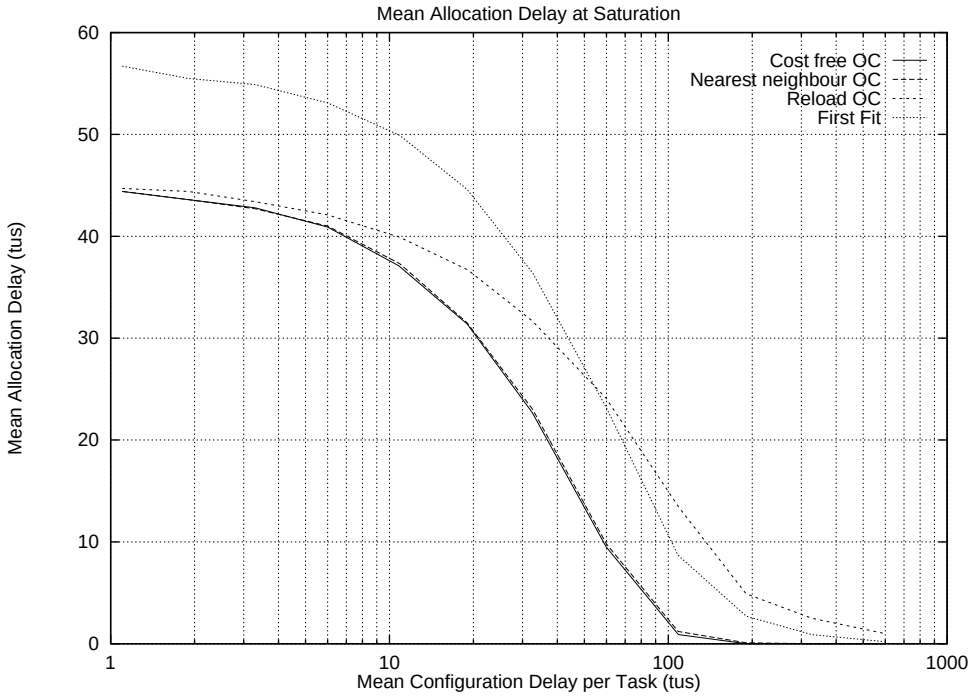


Figure 5.5: (a) Mean allocation delay at saturation for ordered compaction (OC) and first fit as the mean configuration delay per task was varied.

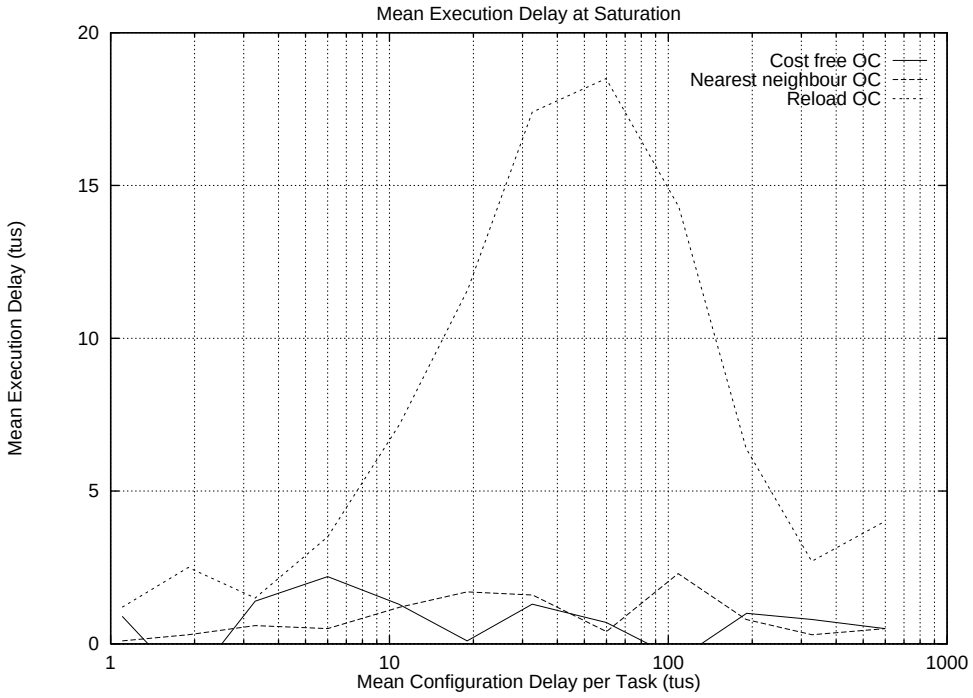


Figure 5.5: (b) Mean execution delay at saturation for ordered compaction and first fit as the mean configuration delay per task was varied.

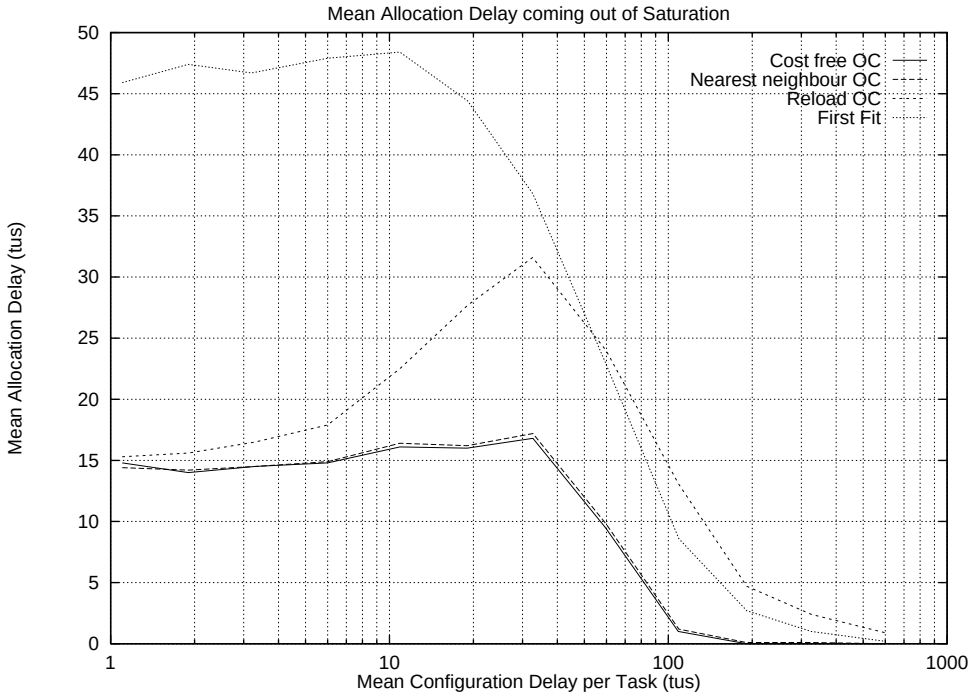


Figure 5.6: (a) Mean allocation delay coming out of saturation for ordered compaction (OC) and first fit as the mean configuration delay per task was varied.

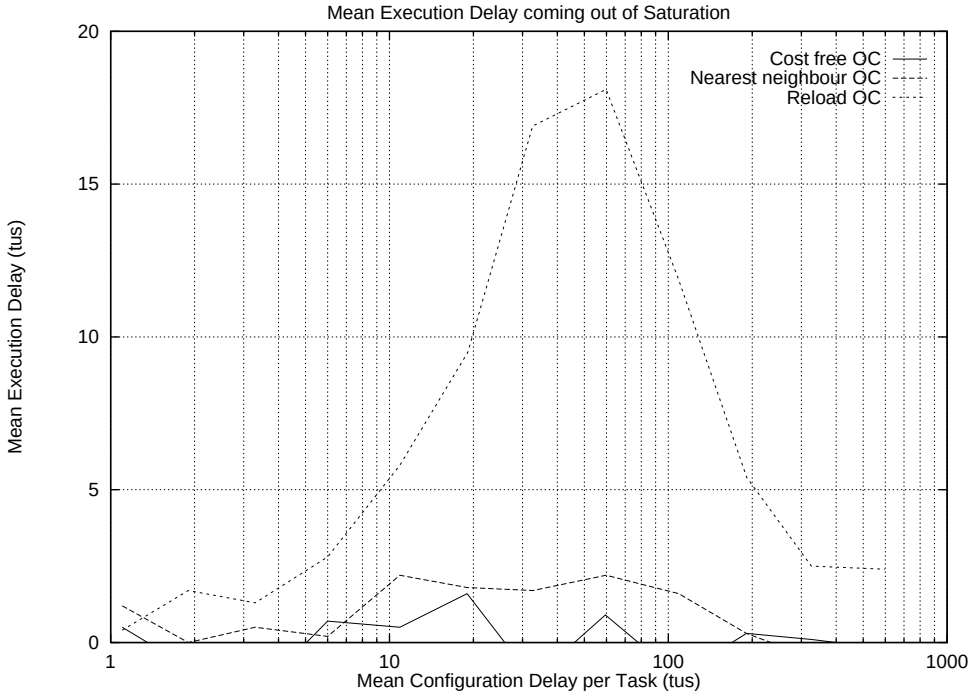


Figure 5.6: (b) Mean execution delay coming out of saturation for ordered compaction and first fit as the mean configuration delay per task was varied.

The mean configuration delay per task refers to the time needed to load an average sized task rather than the time needed to move it over nearest neighbour links. In these plots, performance is compared when the configuration delay per cell and the link delay coincide.

By avoiding the I/O bottleneck, moving tasks over nearest neighbour links eliminates the deterioration in performance that occurs when moving tasks are reloaded as configuration delays rise. The allocation performance is therefore improved even when link delays are extremely high. Given the similarity between cost free and nearest neighbour results for ordered compaction, there appears to be little incentive to use segmented buses to move tasks except when tasks need to be moved substantially further.

5.4 Comparison with Previous Methods

Youn et al. [70] proposed and simulated complete and partial task rearrangement schemes for the mesh of processors. The complete scheme repeatedly shifts the set of executing tasks left then down as far as possible until no further movements are possible or the waiting task can be allocated. The partial task rearrangement scheme places the waiting task at a location that forces a minimal number of allocated processors to be reallocated provided new locations for the displaced tasks can be found that do not cause additional tasks to be relocated. Both schemes relocate processor contexts by moving them over nearest neighbour links. Results were reported for sequential task movements, however, parallel task movements were also examined for the complete scheme.

In contrast to the results reported in this thesis, their results suggest that only modest performance improvements (less than 7%) are possible with complete rearrangements, and that partial rearrangements are not particularly beneficial, especially as communication overheads rise.

Several reasons probably contribute to these findings. The partial rearrangement scheme may have missed some feasible rearrangements because it attempted to reallocate tasks using the adaptive scan method, which is an incomplete recognition scheme [14]. However, it is remarkable that a complete method, which moves all the tasks, should outperform one that only moves a subset of them: the overheads of partial rearrangement were greater than those for complete rearrangement when tasks were moved sequentially. This result suggests that complete rear-

rangements were either not performed very often, or their cost was offset by greater benefits. On the one hand, the complete method appears to have been less effective at liberating trapped free space, and on the other, the costs of a complete rearrangement were amortized over a larger number of allocations as subsequently arriving tasks were also accommodated in the recently cleared mesh area. At 1,000 tasks, the simulation sample size may also have been too small to obtain precise results.

Local repacking and ordered compaction overcome the problems of Youn's methods by increasing recognition capabilities and reducing costs. Whereas local repacking attempts to optimize the use of space in two dimensions, the partial method of Youn et al. misses opportunities for rearrangement, and the complete method appears to be ineffective at collecting free space in both dimensions. The ordered compaction method reduces fragmentation by creating larger free spaces like Youn's complete method, but for less cost. The number of tasks rearranged by the methods described in this thesis adapts to the space requirements of the waiting task. The cost of moving tasks is therefore less than for complete methods.

5.5 Conclusions

Partial rearrangements appear to be effective at reducing the mean allocation delay when the task load is high, and if tasks are reloaded, when configuration delays are short in relation to service periods. As a result, the load–bearing capacity of the FPGA before it saturates is increased, the queue delays and response times in and coming out of saturation are reduced, and the utilization of the FPGA is boosted. These benefits decline as the load falls below saturation levels.

Unfortunately, as configuration delays increase, the I/O bottleneck causes increasing delays to reloading tasks, thereby eroding the benefits of rearrangement. Ordered compaction appears to be slightly more resistant to this effect because scheduling delays are minimized and, since less tasks are usually moved, schedule lengths are shorter. Nevertheless, ordered compaction also fails at relatively low configuration delays when tasks are reloaded. However, the I/O bottleneck can be avoided by moving tasks on–chip. The cost of moving individual tasks are then lower, and tasks can be moved simultaneously. Moreover, the execution delays to individual tasks and the performance benefits of partial rearrangement do not appear to be affected by large configuration and link delays.

The results could be improved upon by enhancing the algorithms' abilities

to identify feasible rearrangements. For example, arbitrary rearrangements such as local repacking are good at gathering free space when tasks are small, however, the results indicate that better packing heuristics are needed to implement such rearrangements when tasks are large. It may be possible to improve the rearrangement recognition capabilities of local repacking further by considering different ways of handling partially intersected tasks. For example, one could try sliding them out of the way into neighbouring subarrays to reduce the number and total area of tasks that need to be repacked. The recognition capabilities of ordered compaction could be improved by considering two– and four–way compactions. Two– or four–way ordered compactions would also reduce schedule lengths if sets of tasks with less total area could thereby be found.

Further performance improvements would flow from modified scheduling strategies. If tasks are to be reloaded, then the delays to tasks being locally repacked could be reduced by switching the order in which scheduling goals are applied. The mean execution delay to tasks could be reduced by minimizing the delays to executing tasks first and inserting the waiting task into the schedule as early as possible without increasing the maximum schedule delay. With tasks spending less time on the array as a result of reduced execution delays, tasks would be allocated sooner.

Refinements that would benefit both ordered compaction and local repacking include: searching for the best rearrangement; aborting the rearrangement and waiting for deallocations that would allow the waiting task to be allocated sooner if task service periods were known; and reducing the number of times a task is moved. If service times and deadlines for tasks are given, then it is straightforward to avoid punishing tasks by moving them too often. However, when these are not known, some other mechanism must be found. Perhaps tasks should not be interrupted in total for more than some specified maximum deviation from the mean allocation delay.

The accuracy of the simulation results could be improved if the delays associated with the host finding and scheduling rearrangements were accounted for. It should be noted that it is possibly unrealistic to assume the time to halt and resume a task is negligible and that it should thus be ignored when tasks are moved on–chip. The assumption is that these times are nevertheless small relative to the configuration delay, and are thus accounted for when the task is loaded. More detailed information on delays to individual tasks would be useful. The interested

reader is referred to [23] for the results of a study on individual delays arising from a one–dimensional ordered compaction algorithm.

Chapter 6

Conclusion

This chapter reviews the results, conclusions, and areas for further study discussed in this thesis.

6.1 Review of the Results and Conclusions

This thesis proposed partially rearranging the executing tasks as a means of alleviating the resource fragmentation problem that occurs when tasks are allocated on–line to a dynamically reconfigurable FPGA.

Two heuristics were developed to overcome the intractability of determining whether a waiting task can be allocated by partial rearrangement. The first method, local repacking, uses a quadtree decomposition of the free space to identify subarrays that may accommodate the waiting task if the tasks executing within them are repacked. The second method, ordered compaction, searches a visibility graph representation of the executing tasks to determine whether the waiting task can be inserted by sliding a subset of the executing tasks off to one side.

Current dynamically reconfigurable FPGAs allow tasks to be relocated by reloading them at their destinations. Scheduling arbitrary rearrangements to minimize the maximum amount of time any task is suspended while waiting to be moved is shown to be NP–hard. However, an approximate solution to an instance of the arbitrary scheduling problem can be found in polynomial time by a depth–first ordered search of a state–space tree, and an optimal ordered compaction schedule can be found in linear time.

When tasks are moved by reloading, partial rearrangements were found to be effective at alleviating fragmentation when tasks arrive more quickly than they

can be processed and the time to configure tasks is small relative to their service needs. However, it was also found that the the benefit of rearranging tasks was overwhelmed by the cost of sequentially reloading the moving tasks when configuration delays were relatively small.

To avoid the I/O bottleneck, it was therefore proposed to move the executing tasks on the chip. The resulting increase in bandwidth would also allow tasks to move in parallel, thereby reducing the interruption to moving tasks and the rearrangement schedule length. Task movements over nearest neighbour links and segmentable buses were investigated. Ordered compactions over nearest neighbour links were found to provide performance benefits at link delays ranging up to the mean task service period in length. The use of buses further reduces the time to move tasks when they are moved beyond their bounding box and offers hope for performing arbitrary rearrangements quickly.

The complexity of scheduling FPGA task movements over a one–dimensional segmentable bus is unknown and is conjectured to be NP–hard. An algorithm for optimally compacting unit length tasks was described, and a heuristic for compacting arbitrarily long tasks was developed. The worst case schedule length for orderly compacting 2D tasks over segmentable row buses was shown to be no better than that using nearest neighbour links.

The methods described in this thesis could also be used to alleviate fragmentation in mesh computers that statically allocate tasks to contiguous processor partitions. On these machines the time to load a task is typically small relative to its service period [35, 45, 64]. Preemptive reallocation could therefore reduce the long queue delays that have been reported. To overcome the fragmentation problem in MIMD arrays, tasks are often allocated non–contiguously. However, when tasks are allocated non–contiguously, messages between task fragments contend for use of links and response times consequently degrade [46, 48]. It may be that the benefits of allocating contiguously and partially rearranging tasks over mesh links outweigh the benefits of non–contiguous allocation. Since it is unlikely that moving the local memory and context for a node over a mesh link would take more time than the average service period, mesh compaction could also be used to move contiguously allocated tasks to unify time–slots in gang–scheduled systems, as suggested by Feitelson [27], who considered reallocation to be too costly.

6.2 Directions for Further Study

Several lines of further investigation can be identified. These include improving the time complexity of the algorithms described in the thesis, improving the allocation performance of the algorithms, making the best use of segmentable buses, and making the methods more applicable. These issues are described in more detail below.

The time complexity of the algorithms described in this thesis could be improved by maintaining the data structures dynamically, through the use of more effective search strategies, and by parallelizing the algorithms, as suggested in Chapter 3.

The allocation performance of the algorithms could be improved by using service periods effectively, not relocating any task too often, using more effective search, packing, and scheduling strategies, and by coping with arbitrarily shaped tasks. All but the last of these points were discussed in Chapter 5. In order to expand upon the last point, consider the fact that FPGA tasks are rarely rectangular, but that rectangles are a convenient abstraction to perform geometric operations. A consequence of bounding tasks with rectangles is the introduction of fragmentation internal to each rectangle, which has been ignored in this thesis, but has also not been available for use. Were the actual task profiles used, better packings may be possible. Given the NP–hardness of packing regular shapes, finding effective heuristics for arbitrary shapes is very challenging.

As discussed in Chapter 4, segmentable buses might be better used if the time complexity of one–dimensional compaction were known, if they could be fully exploited for arbitrary rearrangements, and if the architectural requirements for on–chip task movements were known. Techniques for scheduling rearrangements of two–dimensional tasks are yet to be described.

Finally, partial rearrangements for FPGAs would be more applicable if multi–chip protocols were known, if practical means of rerouting I/O to migrated tasks were determined, and if a decentralized control mechanism were used. Since these issues have not previously been mentioned, a description of them follows.

Field–programmable custom computing machines typically consist of several FPGAs. While individual tasks do not usually span chips because of the slowdown in communicating off–chip, subtasks may be spread over several chips. To minimize the amount of global routing resource used to communicate between task

components, it is desirable to allocate them close to one another. There is therefore a role for rearrangements at the multi–chip (system) level. How best to carry them out and the performance tradeoffs are yet to be investigated.

When I/O to tasks is performed by direct addressing, as is possible with the Xilinx XC6200 series, tasks can be relocated without needing to worry about routing I/O. However, many FPGA task designs depend upon hard–wired I/O from pins at the chip periphery. How to reroute the I/O to migrated tasks so as to minimize the path length, routing area, routing time, and fragmentation of routing resources are open questions.

This thesis has presented a centralized view of partial rearrangements. The solution is not scalable because the overhead for determining the feasibility of rearranging and scheduling rearrangements increases as array sizes and task numbers grow. Effective, autonomous, decentralized defragmentation strategies are therefore sought. Such mechanisms will need to be implemented on the chip to avoid the I/O bottleneck at the chip periphery. If found, and implemented, self configuring FPGAs will be a step closer.

Bibliography

[1] H H Alnuweiri, M Alimuddin, and H Aljunaidi. Switch models and reconfigurable networks: Tutorial and partial survey. In *Proceedings of the Workshop on Reconfigurable Architectures,* 8th International Parallel Processing Symposium, Los Alamitos, CA, April 1994. IEEE Computer Society.

[2] Atmel. AT6000 FPGA configuration guide. Document 0436B, Atmel, August 1997.

[3] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual wires: overcoming pin limitations in FPGA–based logic emulators. In Duncan A Buell and Kenneth L Pocek, editors, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142 – 151, Los Alamitos, CA, April 1993. IEEE Computer Society.

[4] Brenda S Baker, Donna J Brown, and Howard P Katseff. A 5/4 algorithm for two–dimensional packing. *Journal of Algorithms*, 2(4):348 – 368, December 1981.

[5] Brenda S Baker, E G Coffman Jr, and Ronald L Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 4(9):846 – 855, November 1980.

[6] George H Barnes, Richard M Brown, Maso Kato, David J Kuck, Daniel L Slotnick, and Richard A Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C–17(8):746 – 757, August 1968.

[7] Avron Barr and Edward A Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume I. William Kaufmann, Inc., Los Altos, CA, 1981.

[8] Kenneth E Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, C–29(9):836 – 840, September 1980.

[9] Bryan Beresford-Smith, Oliver Diessel, and Hossam ElGindy. Optimal algorithms for constrained reconfigurable meshes. *Journal of Parallel and Distributed Computing*, 39(1):74 – 78, November 1996.

[10] Michael Bolotski, André DeHon, and Thomas F Knight Jr. Unifying FPGAs and SIMD arrays. Transit Note 95, MIT AI Lab, February 1994.

[11] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In Reiner W Hartenstein and Manfred Glesner, editors, *Field–Programmable Logic: Smart Applications, New Paradigms and Compilers, 6th International Workshop, FPL'96 Proceedings*, pages 327 – 336, Berlin, Germany, September 1996. Springer–Verlag.

[12] Timothy Bridges. The GPA machine: A generally partitionable MSIMD architecture. In Joseph JaJa, editor, *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation (Frontiers'90)*, pages 196 – 203, Los Alamitos, CA, October 1990. IEEE Computer Society.

[13] Ming-Syan Chen and Kang G Shin. Subcube allocation and task migration in hypercube multiprocessors. *IEEE Transactions on Computers*, 39(9):1146 – 1155, September 1990.

[14] Po-Jen Chuang and Nian-Feng Tzeng. Allocating precise submeshes in mesh connected systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):211 – 217, February 1994.

[15] E G Coffman Jr., M R Garey, and D S Johnson. Approximation algorithms for bin-packing – an updated survey. In G Ausiello, M Lucertini, and P Serafini, editors, *Algorithm Design for Computer System Design*, pages 49 – 106. Springer-Verlag, Vienna, Austria, 1984.

[16] E G Coffman Jr, M R Garey, D S Johnson, and R E Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808 – 826, November 1980.

[17] E G Coffman Jr and P W Shor. Packings in two dimensions: Asymptotic average–case analysis of algorithms. *Algorithmica*, 9(3):253 – 277, March 1993.

[18] Debendra DasSharma and Dhiraj K Pradhan. Submesh allocation in mesh multicomputers using busy–list: a best–fit approach with complete recognition

capability. *Journal of Parallel and Distributed Computing*, 36(2):106 – 118, August 1996.

[19] Oliver Diessel and Hossam ElGindy. Ordered partial task compaction on mesh connected computers. Technical report 96-11, Department of Computer Science and Software Engineering, The University of Newcastle, September 1996.

[20] Oliver Diessel and Hossam ElGindy. Partial FPGA rearrangement by local repacking. Technical report 97–08, Department of Computer Science and Software Engineering, The University of Newcastle, September 1997. Accepted as a poster paper for FPGA'98.

[21] Oliver Diessel and Hossam ElGindy. Run–time compaction of FPGA designs. In Wayne Luk, Peter Y K Cheung, and Manfred Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, pages 131 – 140, Berlin, Germany, 1997. Springer–Verlag.

[22] Oliver Diessel and Hossam ElGindy. Partial rearrangements of space–shared FPGAs (Extended abstract). Technical report 98–01, Department of Computer Science and Software Engineering, The University of Newcastle, January 1998. Accepted for presentation at EHPC'98.

[23] Oliver Diessel, Hossam ElGindy, and Bryan Beresford-Smith. Partial task compaction reduces queuing delays in partitionable–array machines. In *Proceedings of the Third Australasian Conference on Parallel and Real–Time Systems*, pages 186 – 194, Brisbane, Australie, September 1996. Griffith University.

[24] Oliver Diessel, Hossam ElGindy, and Lachlan Wetherall. Efficient broadcasting procedures for constrained reconfigurable meshes. In *Proceedings of the Third Australasian Conference on Parallel and Real–Time Systems*, pages 85 – 88, Brisbane, Australia, September 1996. Griffith University.

[25] P Dillien and I Phillips. ASIC design flexibility with ERAs. *Electronic Product Design*, 10(10):29 – 34, October 1989.

[26] Charles R Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335 – 348, August 1982.

[27] Dror G Feitelson. Packing schemes for gang scheduling. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Los Alamitos, CA, April 1996. IEEE Computer Society.

[28] L Ferguson. Image processing using reconfigurable FPGAs. *Proceedings of the SPIE - The International Society for Optical Engineering. High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, 2914:110 – 121, November 1996.

[29] M R Garey and D S Johnson. *Computers and Intractability — A Guide to the Theory of NP–Completeness*. W. H. Freeman, San Francisco, CA, 1979.

[30] David F George, Bernard K Gunther, and George J Milne. Modelling concurrent systems for simulation on reconfigurable computers. In Nalin Sharda and Audrey Tam, editors, *Proceedings of PART'97 The 4th Australasian Conference on Parallel and Real–Time Systems*, pages 298 – 309, Singapore, Singapore, September 1997. Springer–Verlag.

[31] A Graf. A field programmable gate array. In *Proceedings, 6th International Conference on Custom and Semicustom ICs*, pages 1 – 7, Saffron Walden, UK, 1986. Prodex.

[32] B K Gunther, G J Milne, and V L Narasimhan. Assessing document relevance with run-time reconfigurable machines. In Kenneth L Pocek and Jeffrey M Arnold, editors, *4th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96)*, pages 10 – 17, Los Alamitos, CA, 1996. IEEE Computer Society.

[33] J D Hadley and B L Hutchings. Design methodologies for partially reconfigured systems. In Peter Athanas, editor, *1995 3rd Annual Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 78 – 84, Los Alamitos, CA, April 1995. IEEE Computer Society.

[34] John R Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L Pocek and Jeffrey M Arnold, editors, *5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 24 – 33, Los Alamitos, CA, April 1997. IEEE Computer Society.

[35] Steven Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Los Alamitos, CA, April 1996. IEEE Computer Society.

[36] Chih-Hao Huang and Jie-Yong Juang. A partial compaction scheme for processor allocation in hypercube multiprocessors. In Benjamin W Wah, editor, *Pro-*

ceedings of the 1990 International Conference on Parallel Processing, volume 1, pages 211 – 217, University Park, PA, 1990. Pennsylvania State University Press.

[37] Brad L Hutchings and Michael J Wirthlin. Implementation approaches for reconfigurable logic applications. In W Moore and W Luk, editors, *Field–Programmable Logic and Applications, 5th International Workshop, FPL'95 Proceedings*, pages 419 – 428, Berlin, Germany, August 1995. Springer-Verlag.

[38] Ju-wook Jang, H Park, and Viktor K Prasanna-kumar. A bit model of reconfigurable mesh. In *Proceedings of the Workshop on Reconfigurable Architectures,* 8th International Parallel Processing Symposium, Los Alamitos, CA, April 1994. IEEE Computer Society.

[39] Muhammad Khellah, Stephen Brown, and Zvonko Vranesic. Modelling routing delays in SRAM–based FPGAs. In *Proceedings of the Canadian Conference on Very Large Scale Integration (CCVLSI '93)*, pages 6B.13 – 6B.18, November 1993.

[40] Shahram Latifi. Migration of tasks in interconnection networks based on the star graph. *Journal of Parallel and Distributed Computing*, 31(2):166 – 173, December 1995.

[41] Wong-Hua Lee and Miroslaw Malek. MOPAC: A partitionable and reconfigurable multicomputer array. In Howard Jay Siegel and Leah Siegel, editors, *Proceedings of the 1983 International Conference on Parallel Processing*, pages 506 – 510, Silver Spring, MD, August 1983. IEEE Computer Society.

[42] Keqin Li and Kam Hoi Cheng. Complexity of resource allocation and job scheduling problems on partitionable mesh connected systems. Technical report UH-CS-88-11, Department of Computer Science, University of Houston, Houston, TX, 1988. Appeared in Proceedings 1st IEEE Symposium on Parallel and Distributed Processing, 1989, pp. 358 – 365.

[43] Keqin Li and Kam-Hoi Cheng. Job scheduling in a partitionable mesh using a two–dimensional buddy system partitioning scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):413 – 422, October 1991.

[44] D A Lifka. The ANL/IBM SP scheduling system. In D G Feitelson and L Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295 – 303, Berlin, Germany, 1995. Springer–Verlag.

[45] Virginia Lo, Jayne Miller, Kurt Windisch, Yuke Zhuge, Dror Freitelson, Reagan Moore, and Bill Nitzberg. A comparison of workload traces from two production parallel machines. Supercomputing 95 Poster Paper. Available by ftp from `ftp.cs.uoregon.edu/pub/lo/sc95poster.ps.gz`.

[46] Virginia Lo, Kurt Windisch, Wanqian Liu, and Bill Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. Technical report, Department of Computer and Information Science, University of Oregon, Eugene, OR, 1994. Appeared as "Non-contiguous Processor Allocation Algorithms for Distributed Memory Multicomputers" in *Proceedings of Supercomputing '94*, Nov. 1994 pp 227 – 236.

[47] Russ Miller, Viktor K Prasanna-Kumar, Dionis I Reisis, and Quentin F Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678 – 692, June 1993. (A preliminary version of this paper was presented at *5th MIT Conference on Advanced Research in VLSI*, 1988).

[48] Sherry Q Moore and Lionel M Ni. The effects of network contention on processor allocation strategies. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 268 – 273, Los Alamitos, Ca., April 1996. IEEE Computer Society, IEEE Computer Society Press.

[49] J R Nickolls. The design of the MasPar MP–1: A cost effective massively parallel computer. In *Digest of papers COMPCON Spring 90. Proceedings of the 35th IEEE Computer Society International Conference*, pages 25 – 28, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[50] Gary J Nutt. A parallel processor operating system comparison. *IEEE Transactions on Software Engineering*, SE–3(6):467 – 475, November 1977.

[51] John K Ousterhout, Gordon T Hamachi, Robert N Mayo, Walter S Scott, and George S Taylor. The MAGIC VLSI layout system. *IEEE Design and Test of Computers*, 2(1):19 – 30, February 1985.

[52] Panel Session: How to promote reconfigurable computing R&D in Australia?, PART'97 The 4th Australasian Conference on Parallel and Real–Time Systems, Newcastle, Australia, September 1997.

[53] J Rose, A ElGamal, and A Sangiovanni-Vincentelli. Architecture of field–programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013 – 1028, July 1993.

[54] J Rosenberg. DSP acceleration using Cache Logic FPGAs. *Proceedings of the SPIE - The International Society for Optical Engineering. Field Programmable Gate Arrays (FPGAs) for Fast Board and Reconfigurable Computing*, 2607:54 – 59, October 1995.

[55] Michael Saleeba. A self-contained dynamically reconfigurable processor architecture. *Australian Computer Science Communications. Sixteenth Australian Computer Science Conference. ACSC-16.*, 15(1):59 – 70, February 1993.

[56] H Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187 – 260, June 1984.

[57] T Schwederski, H J Siegel, and T L Casavant. A model of task migration in partitionable parallel processing systems. In *Frontiers 88: 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 211 – 214, Washington, DC, October 1988. IEEE Computer Society.

[58] T Schwederski, H J Siegel, and T L Casavant. Task migration transfers in multistage cube based parallel systems. In K P McAuliffe and P M Kogge, editors, *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 296 – 305, University Park, PA, August 1989. Pennsylvania State University Press.

[59] Howard Jay Siegel. The theory underlying the partitioning of permutation networks. *IEEE Transactions on Computers*, C–29(9):791 – 800, November 1980.

[60] Howard Jay Siegel, Leah J Siegel, Frederick C Kemmerer, Philip T Mueller Jr., Harold E Smalley Jr., and S Diane Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C–30(12):934 – 947, December 1981.

[61] Satnam Singh, John Patterson, Jim Burns, and Michael Dales. PostScript rendering with virtual hardware. In Wayne Luk, Peter Y K Cheung, and Manfred Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, pages 428 – 437, Berlin, Germany, 1997. Springer–Verlag.

[62] Daniel D K D B Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37 – 40, February 1980.

[63] Alan P Sprague. A parallel algorithm to construct a dominance graph on nonoverlapping rectangles. *International Journal of Parallel Programming*, 21(4):303 – 312, August 1992.

[64] Takashi Suzuoka, Jaspal Subhlok, and Thomas Gross. Evaluating job scheduling techniques for highly parallel computers. Technical report CMU–CS–95–149, School of Computer Science, Carnegie Mellon University, August 1995.

[65] Jerry L Trahan, Ramachandran Vaidyanathan, and Ratnapuri K Thirachelvan. On the power of segmenting and fusing buses. *Journal of Parallel and Distributed Computing*, 34(1):82 – 94, April 1996.

[66] Lewis W Tucker and George G Robertson. Architecture and applications of the Connection Machine. *Computer*, 21(8):26 – 38, August 1988.

[67] John Villasenor, Chris Jones, and Brian Schoner. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):565 – 567, December 1995.

[68] Michael J Wirthlin and Brad L Hutchings. Sequencing run–time reconfigured hardware with software. In *FPGA'96 1996 ACM Fourth International Symposium on Field Programmable Gate Arrays*, pages 122 – 128, New York, NY, February 1996. ACM.

[69] Xilinx. XC6200 Field Programmable Gate Arrays. Technical report, Xilinx, Inc., October 1996.

[70] Hee-yong Youn, Seong-Moo Yoo, and B Shirazi. Task relocation for two–dimensional meshes. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 230 – 235, Raleigh, NC, October 1994. ISCA.

[71] Yahui Zhu. Efficient processor allocation strategies for mesh–connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328 – 337, December 1992.

[72] Yahui Zhu. Fast processor allocation and dynamic scheduling for mesh multiprocessors. *Computer Systems Science and Engineering*, 11(2):99 – 107, March 1996.