2016

Runtime performance enhancement of nondeterministic applications on partailly [i.e. partially] reconfigurable architectures using task agglomeration

David Austin

RUNTIME PERFORMANCE ENHANCEMENT OF NONDETERMINISTIC APPLICATIONS ON PARTAILLY RECONFIGURABLE ARCHITECTURES USING TASK AGGLOMERATION

by

DAVID AUSTIN

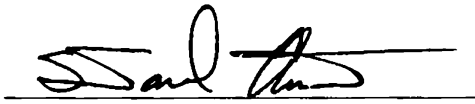
A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
The Shared Computer Engineering Program of
The University of Alabama in Huntsville
The University of Alabama at Birmingham
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2016

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this dissertation.



David Austin

8 Aug 2016

(date)

DISSERTATION APPROVAL FORM

Submitted by David Austin in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering.

B. Earl Wells 5/5/16
Dr. B. Earl Wells (Date) Committee Chair

Sampson Gholston 8-5-16
Dr. Sampson Gholston (Date)

8-5-16
Dr. Jeff Kulick (Date)

8/5/16
Dr. Aleksandar Milenkovic (Date)

8/5/16
Dr. Murat Tanik (Date)

8/24/16
Dr. Ravi Gorur (Date) Department Chair

08/24/16
Dr. Shankar Mahalingam (Date) College Dean

11/7/16
Dr. David Berkowitz (Date) Graduate Dean

ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Doctor of Philosophy College/Dept. Engineering/Electrical and Computer Engineering

Name of Candidate David Austin

Title Runtime Performance Enhancement of Nondeterministic Applications on Partially Reconfigurable Architectures Using Task Agglomeration

Reconfigurable hardware architectures have the potential to dramatically improve run-time performance by altering their low-level hardware to match the inherent structure of a given application. Partially-reconfigurable architectures extend this capability by allowing their reconfigurable resources to be partitioned into multiple subregions which are independently configurable. This permits the run-time reconfiguration of a single hardware partition while other partitions continue their operation unabated. The major limiting factor of these architectures is the relatively long time required to perform this fine-grained hardware modification.

Previous research on the Single-Chip Multiprocessor and Dynamically Reconfigurable Hybrid Architecture have developed an application framework and a partially reconfigurable hardware architecture that can be applied to both deterministic and bounded nondeterministic applications. In this work, both models are extended to account for the limited resources available within each hardware partition. This resource constraint model was developed consistent with implementation in modern commercially available programmable logic devices.

To balance widely varying task resource requirements, two new compile-time task agglomeration methods have been developed. These methods are designed to identify tasks and combine them into common partial bitstreams that do not exceed the resource limitations of the reconfigurable partition. By more efficiently using the partition resources the total number of required reconfigurations can be reduced.

Both methodologies have as their core element a static scheduler that is used to estimate the resultant execution time of a task grouping. The first method applies this static scheduler once and then performs greedy local optimization to minimize the number of such groupings. The second method utilizes simulated annealing to actively optimize the estimated execution time. The effectiveness of both of these techniques was evaluated using the same synthetically generated task systems and discrete event simulator from previous research. In 86% of the cases, the Simulated Annealing algorithm produced significantly shorter run times while the greedy method produced significantly shorter run times in 70% of the cases when compared to the case where task agglomeration is not applied.

Abstract Approval: Committee Chair B. Earl Wells 8/5/2016
Dr. B. Earl Wells

Department Chair 8/24/16
Dr. Ravi Gorur

Graduate Dean 11/7/16
Dr. David Berkowitz

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

CHAPTER 1

INTRODUCTION TO DYNAMIC RECONFIGURABLE SYSTEMS

Partially reconfigurable systems possess the potential to revolutionize high performance computing. These systems contain fine grained digital logic that can be continuously modified while the system is operating. Such systems have the potential to combine the performance of application specific hardware with the flexibility of a software implementation. In order to understand the potential of reconfigurable systems, their capabilities and limitations must be understood, as well as the specialized terminology associated with them. This foundation will be essential as the reconfigurable computational model is extended to include nondeterministic application characteristics with finite hardware resource availability.

1.1 Overview of Reconfigurable Computing

A reconfigurable computer is a system where the underlying computational hardware can be modified at runtime. The concept of reconfigurable computing has been proposed for a number of years [1], but the technology necessary to implement a partially reconfigurable system has only recently become commercially available. Since the demise of the Application Specific Integrated Circuit (ASIC) [2], programm-

able devices such as Field Programmable Gate Arrays (FPGAs) are the most widely used components to implement reconfigurable systems [3]. Programmable devices are designed to implement arbitrary circuits, and recent technological advances have led to a wide assortment of reconfigurable systems that have been built using programmable logic components.

Traditional instruction-set microprocessors sacrifice efficiency for the sake of generality. They are able to execute a wide variety of applications, but they are required to conform to a load-execute-store architecture model. The stored program approach incurs significant overhead by the repeated accesses to memory [4]. On the other hand, application-specific hardware is capable of faster performance because the hardware can be optimized to a specific application. However, under most circumstances, the use of such custom hardware is not acceptable due to the high cost and design effort for custom designed, single-use, application specific hardware. System designers, therefore, look to implement solutions in software rather than hardware, since software can be applied to a wide range of problems, which allows it to adapt to changing requirements [5].

Recently, General Purpose Graphical Processors (GPGPUs or GPUs) have been the focus of much high-performance computing research, primarily because they are readily available and cost effective. While GPUs excel at solving many highly parallel problems, their Single Instruction Multiple Data (SIMD) [6] model is not efficient when it is applied to applications with irregularly structured task dependencies. Hardware implementations are not constrained by SIMD limitations and are inherently parallel. When applied to these irregularly structured problems, hardware

solutions outperform GPUs [7]. Furthermore, for specialized problems, additional improvements can be realized by including application specific logic.

Although hardware solutions outperform most currently available high performance computing alternatives, they require significant design costs and are inflexible once implemented. Reconfigurable systems offer a means to apply application specific hardware in a general manner, and represent a possible compromise between the two extremes of general purpose and application specific hardware [8]. Since the hardware can be modified after fabrication, highly efficient application specific designs can be created using commodity hardware. Given that reconfigurable computers use existing hardware, the design time is shortened significantly compared to fully custom designs. The physical components are not modified, only the internal device configuration, leading to the potential to reuse existing technology. This potential does not exist only across different applications, but may also be applied to different components within a single application.

There are several types of reconfigurable computers. One of the most useful types of reconfigurable system is the *run-time reconfigurable computer*, which does not require the processor to be completely powered off to be reconfigured. Run-time reconfiguration allows a single device to implement several different functions independently, much as a microprocessor can context switch between many executing processes. Since the system does not require a full restart, it can perform several discrete operations that are part of a larger processing task.

Another benefit of run-time reconfiguration is that a design can be mapped into a reconfigurable system that is physically smaller than what would normally be

necessary to implement the full design. Just as software tasks consume system memory, reconfigurable hardware consumes logic, routing, and memory resources of the reconfigurable device. Using run-time reconfiguration, a large design can fit into a physically smaller device much as virtual memory systems [9] allow large software applications to be mapped into systems with smaller physical memory. Portions of the application problem that are not active can be removed from the reconfigurable computer and replaced, so that other parts of the design can make use of the computer's resources, allowing time-sharing of the resources within the device.

In order to map the design logic associated with an application onto a physically smaller device, it must be divided into independent discrete functional units, or tasks. Often, division along solely functional boundaries creates a suboptimal condition since the device's hardware must be partitioned so that enough resources are available to implement the largest possible functional unit it will ever contain. If the functional partitioning results in one large task and many smaller tasks, the spatial efficiency is poor; much of the area available in the hardware goes unused by the majority of the tasks. Since there are unused resources not performing useful work, this also affects the computational efficiency of the system as a whole.

The concepts involved in reconfigurable computing are similar to many of the foundational concepts that have been recognized in traditional computing for some time. The architecture of the reconfigurable system and the structure of the application problem that the system has been employed to solve are two of these concepts. Although there is some overlap between the two areas, most reconfigurable

systems can be resolved into a separate reconfigurable application and reconfigurable platform.

The reconfigurable system executes an application that is intended to solve a particular problem. This application consists of a number of discrete activities, or tasks, that effect a specific result. Although the application can be partitioned arbitrarily, it is usually assumed that the partitioning reflects functional boundaries such that each task represents a functional set of operations executed in order that transforms inputs into outputs. The overall application then accomplishes its intended result due to the temporal arrangement of the tasks in the application.

In a nonpreemptive model, the successful operation of the application is the arrangement of tasks in the execution order dictated by a set of precedence constraints. These constraints can either be classified as *data dependencies* or *control dependencies* depending on if they transfer data or control to their successor task. If a task is data dependent on another task, the dependent task is said to be a data dependency *sink task*, whereas the predecessor task is said to be a data dependency *source task*. Likewise, the subjects of a control dependency are known as control dependency sink tasks and control dependency source tasks. An application may be viewed as an arrangement of tasks such that the data and control dependencies are met in a meaningful way to accomplish a specific purpose.

Each task is distinguished from the other tasks by a unique task ID, which serves to identify it in the application. In addition to its ID, each task is described by several additional parameters, which help define the application's execution. A task consumes resources when it is implemented in the reconfigurable system. This metric

is often referred to as the area of the task because resource usage is often closely related to the area it consumes in the device. The time required for the task to complete execution is its *execution time*, while the time required to reconfigure (context switch) the device for the task is it's *reconfiguration time*. For most runtime reconfigurable systems, the context switching time is comparatively long relative to the execution time, which imposes a significant penalty on performing a context switch. Given the long context switching time, a preemptive model, where a task is interrupted before it completes execution, is not considered to be practical.

An application may require a given function to be performed multiple times during its execution. To accommodate this, each task is assigned a *task type*. A task's type represents in an abstract way the specific set of actions the task performs. Tasks with the same type perform the same operations; however, the data that the task operates on is expected to differ between instantiations of the same type with different task IDs. The application may require the same task type to be executed several times during the application lifetime. In this case, each instance of a task is assigned a separate ID, but the task type for each task instance is identical.

The reconfigurable system is composed of a *hardware platform*, which describes the execution environment. The platform consists of multiple execution units, referred to as Processing Elements (PEs), which are responsible for performing the computational work of the reconfigurable system. These PEs are either designated as hardware or software, depending on the selection of reconfigurable hardware or a traditional microprocessor for the execution unit.

Hardware PEs are composed of reconfigurable logic and are the execution units for hardware tasks. Each hardware PE can be configured separately and independently multiple times during the scope of the application. Depending on the reconfigurable architecture, a single PE may equate to a single Integrated Circuit (IC), or to a portion of a partially reconfigurable device. In devices that implement partial dynamic reconfiguration, the device may be partitioned to include several hardware PEs. In older devices that do not have this capability, the PE is equivalent to the complete device. This affects the platform architecture. In partial dynamic devices, the number of PEs can be increased or decreased by repartitioning the device. Architectures that do not support this capability can only add PEs by adding additional IC chips.

Software PEs are traditional instruction set processors and are used to execute software tasks. A software PE may be a physical processor either as a separate IC or as part of the reconfigurable hardware system. With recent advances in programmable logic density, soft-core microprocessors can be implemented using programmable resources. Modern soft-cores are sufficiently powerful to support the processing of many applications. Regardless of the implementation of the software PE, it is assumed to include sufficient resources, such as memory, to execute the tasks it has been assigned.

1.2 Taxonomy of Reconfigurable Computing Architectures

The most general form of reconfigurable computer relies on a one or more programmable hardware resources that may or may not be dynamically reconfigurable.

Such systems generally rely on a hardware accelerator to improve the performance of a few actions that are executed frequently enough to justify special purpose hardware. While the reconfigurable component may be modified to change the hardware accelerator, doing so causes all of the devices reconfigurable resources to be unavailable. While the dedicated hardware accelerates certain functions, the delay to reconfigure the hardware represents significant overhead. Practical reconfigurable systems require greater responsiveness than this type of reconfigurable computer can support.

A more advanced reconfigurable system can be constructed from several programmable chips, which are individually reconfigurable. In this case, each chip represents a single PE. When it is necessary to context switch one of the reconfigurable PEs, that PE is loaded with an implementation for another task. A master control resource, such as a dedicated microprocessor is responsible for scheduling execution and reconfiguration of the reconfigurable computer. Additionally, since the hardware can be modified, data flow between two tasks must be scheduled so that both tasks are currently implemented in PEs, or arrange for data storage until they are. Although this computer does not suffer the same resource bottleneck, it suffers from additional runtime and design complexity through the use of multiple IC components, and the need for a dedicated interconnect among them.

In devices supporting partial dynamic reconfiguration, a reconfigurable computer can be realized with a single programmable device. In such a *Reconfigurable System on Programmable Chip* (RSoPC), the individual PEs can be implemented by partitioning the internal logic resources into a number of partailly reconfigurable regions. As there is no need for the partitions to be of equal size, it is also possi-

ble to optimize the number, location and size of the reconfigurable partitions to a specific application. Since the chip is programmable, a different application can be targeted without modifying the underlying hardware. Given the reconfigurable nature of the hardware, the programmable interconnect can also be modified to optimize the solution to the application. Functionally, this architecture is equivalent to the one previously discussed. The only difference is that the reconfigurable interconnect allows greater flexibility. Both architectures support a discrete number of hardware PEs with an interconnection scheme allowing individual tasks to communicate as necessary.

To extend the architectural discussion further, the most general purpose reconfigurable systems can be realized by deploying several partially dynamically reconfigurable devices. Adding a single device results in adding several PEs. Although the internal connections can be reprogrammed as necessary, the physical routes between chips remain constant. While point-to-point connections are the most efficient in terms of data throughput, fully connecting each PE in each chip would be highly inefficient in terms of resource utilization. To combat this problem, many reconfigurable computers have transitioned to network topologies, which use high-speed switches to maximize throughput between PEs [10], [11]. While this architecture is highly generic, it is also extremely complex, with significant overhead to manage scheduling execution, reconfiguration, and communication.

1.3 Practical Reconfigurable Systems

Most reconfigurable hardware PEs are realized by using programmable logic, typically, SRAM-based FPGAs. FPGAs are used to implement reconfigurable systems since they are commercially available, easy to reprogram in circuit, and relatively inexpensive. Modern FPGAs also contain many logic resources, which allow them to implement complex systems. This makes it possible to implement a reconfigurable computer using application platforms built around commodity FPGAs [12].

Standard FPGAs make use of the concept that an 2^n x m memory can be used to implement m arbitrary boolean functions of n inputs [13] by implementing a Lookup Table (LUT). Rather than implementing combinatorial logic, the LUT approach computes the outputs for each input combination before circuit realization. The results are then stored in a memory element, and the memory's address signals are treated as circuit inputs. Applying a given input value to the address lines causes the memory to produce the output corresponding to the stored value at the given memory address. To enhance performance, LUTs are often combined with additional logic to implement a Logic Cell (LC). Typical LC implementations also include registers along with the LUT so that inputs may be retained. Figure 1.1 depicts such a LC. The primitive LC depicted in the figure contains only a one-bit LUT and a D-type flip-flop that can be used to register the LUT output. A multiplexor is also included to select either the LUT output or the flip-flop output. In advanced devices, LC's may also include block memory or application specific hardware accelerators, such as adders and multipliers. Vendor terminology differs, but these are often called Digital Signal

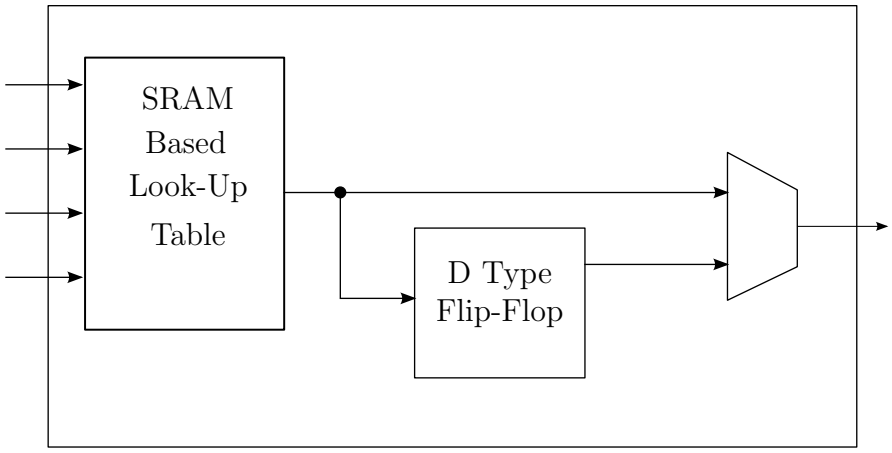


Figure 1.1: Example Logic Cell

Processor (DSP) blocks. Collectively, the non-LC logic is referred to as heterogeneous resources.

Within a standard FPGA, LCs are arranged in a matrix layout with regularly placed programmable switches. The LC output signals interface to an interconnect system that is routed to a nearby programmable switch. Dedicated routes between the switches allow arbitrary connections between the inputs and outputs of the various LCs, as well as the input/output (I/O) pins on the exterior of the FPGA IC package. The I/O pins allow connections to external components, such as memory elements, transceiver chips, ASIC chips, or other FPGAs. Given a FPGA with sufficient resources, arbitrary logic can be constructed by appropriately programming LCs and their associated interconnect resources.

Figure 1.2 shows a simplified representation of a FPGA's internal arrangement. In this figure, the rounded blocks represent the LCs, and the square blocks represent the programmable switches. The figure also depicts an example arrangement of mem-

ory and DSP resources. Memory blocks are represented by the shaded blocks, while DSP blocks are represented by the hatched blocks. Such an arrangement is typical of most modern FPGAs where the LCs and heterogeneous resources are arranged in a columnar structure.

A variety of technologies may be used to implement the LUT and other logic in the device. The most popular devices are implemented using SRAM or DRAM components, which lose their configuration when power is lost. When power is applied, an external device is required to program the FPGA to the desired state. This device must send the FPGA the initial content of all of the LCs as well as the settings for the programmable switches. The device requires an external non-volatile memory to store the FPGA configuration (FPGA bitstream). The most recent devices are capable of reading the configuration memory directly and configuring themselves on power-up. In advanced FPGAs, this self-configuration capability can also be used to perform self-dynamic partial configuration [14].

The most widely used types of FPGAs are programmed in a column-oriented manner [15]. Each word of configuration data is written sequentially to the device until it receives enough data for a complete column. The entire column is then programmed at once. Columns of heterogeneous resources are interspersed in the FPGA fabric. These columns may be DSP LCs, block memory LCs, or other accelerator resources. These heterogeneous resource columns are programmed in the same manner, although the amount of configuration data per column depends on the type of heterogeneous resource it consists of.

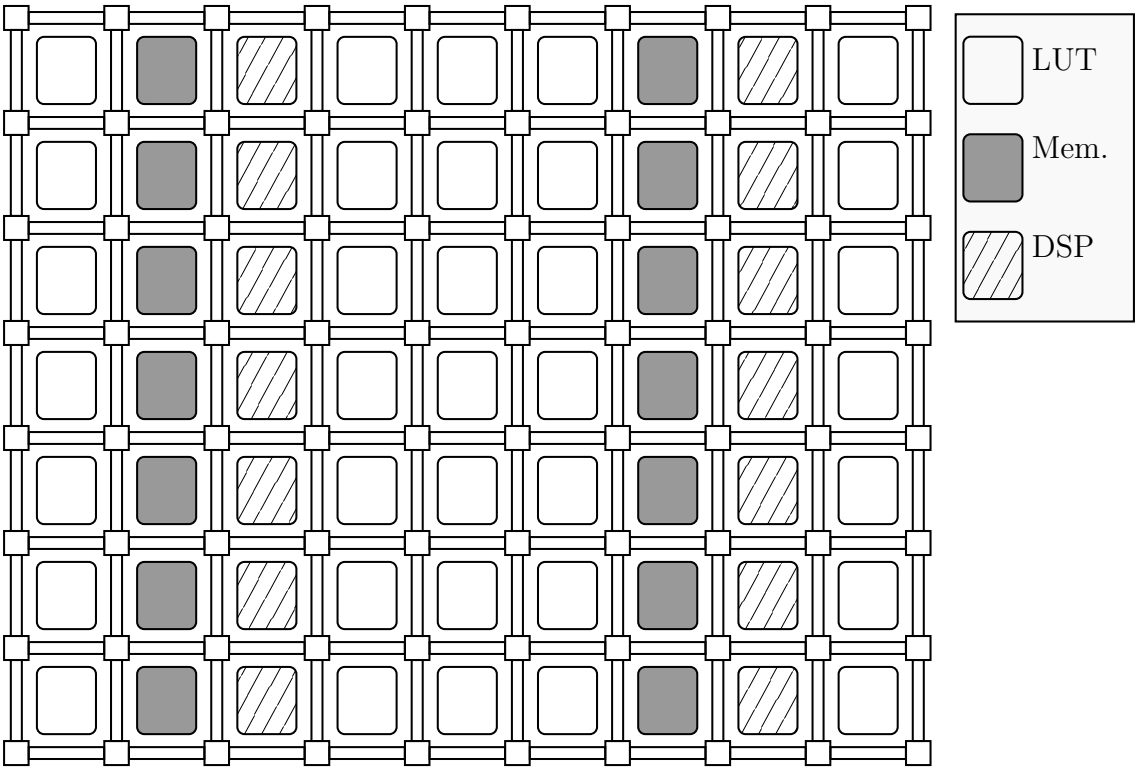


Figure 1.2: Example FPGA Arrangement

Although FPGAs are capable of implementing reconfigurable systems, the technical literature on reconfigurable computing often makes many simplifying assumptions about the capabilities of these systems. Principally these are related to a few often used simplifications. Practical reconfigurable computers are: moderate to coarse-grained reconfigurable systems, have a non-zero reconfiguration time, have a finite number of reconfiguration controllers, and require a one-to-one mapping between PEs and their programming image.

Theoretical models of reconfigurable systems often consider the reconfigurable resources as an array of fine-grained logic that can be modified at will. This was true in older reconfigurable devices, that were relatively small and composed of a regular

structure [16]. The nonuniform pattern of heterogeneous resources in modern FPGAs makes this impractical for realistic systems [17].

As a consequence of the fine grained logic model, early research often viewed the reconfigurable resources as a contiguous region where application logic could be placed at arbitrary locations in the FPGA fabric [18]. Resource allocation was performed dynamically by reserving sufficient resources in a variable one-dimensional region occupying the entire height of the device for a specified length [19], [20]. This model was later extended to allow two-dimensional placement at arbitrary locations based on a specified height and width [21]. Since early devices had a homogeneous structure, this was a reasonable approach. However, in modern FPGAs, heterogeneous resources are dispersed throughout the device. This results in the configuration bitstream being a function of both location and size in the reconfigurable device, which makes both the 1D and 2D models impractical in modern devices [22].

To overcome the problems with the theoretical model, a modular approach [23] is required to successfully implement a partially reconfigurable architecture. In this approach, a moderate grained function is targeted to a specific region in the reconfigurable device. The device is partitioned into one or more execution regions, each of which can be independently reconfigured at runtime.

The implementation of moderate grained, modular architectures requires additional planning. When using partial reconfiguration, the device must be divided into a static region and one or more dynamically reconfigurable regions. The static region is used to implement functions that are not altered during the life of the application, such as reconfiguration control logic and I/O between the various reconfigurable re-

gion(s) as well as any I/O that connects external components to the reconfigurable device. Generally, the regions are viewed as a set of reconfigurable tiles which in a reconfigurable computing system would equate to individual hardware PEs.

In addition to the requirements for partitioning the design, partial reconfiguration introduces some additional complications to the manner logic buffering is implemented. The input and output of the dynamic region(s) must be buffered to prevent unstable signal levels on these interfaces while the region is being reconfigured. This is done by sensing an imminent reconfiguration, and driving the I/O to a known state. Any buffering must be implemented in the static logic area to ensure that the buffer logic is itself not reconfigured.

If a partial reconfiguration region shares columns with other logic in the static region, it can cause undesired operation. The static logic may be overwritten by the partial bitstream of the reconfigurable region. To prevent this possible catastrophic error, a process known as floorplanning is used in practical FPGAs to fix the static and dynamic configuration areas to specific locations within the device. The process of floorplanning also serves to reserve space for the reconfigurable tiles. These floorplanned tiles are used to implement the hardware PE partitions in partially reconfigurable systems.

When a tile is floorplanned, it should be sized large enough to contain the largest piece of logic that it will contain. Modifying the floorplan after the static logic has been placed would invalidate the static bitstream, resulting in the need to recompile it. Since recompiling is a computationally expensive task, it should not be done during runtime. To prevent the need to recompile the static region, therefore,

the PEs should be floorplanned to reserve enough resources to implement all tasks that will be targeted to that specific PE.

Another practical restriction on the capabilities of existing hardware is the number of simultaneous reconfigurations a device may support [24]. Typical hardware limits the number of such reconfigurations because the device relies on dedicated circuitry to perform reconfiguration. A practical reconfigurable system cannot perform simultaneous reconfiguration if the dedicated circuitry is being used to reconfigure another dynamic region. A partial reconfiguration request will be required to wait until the hardware becomes free before the process is completed. The idle time imposed by a finite number of reconfiguration controllers can be significant when a system requires many simultaneous reconfigurations.

Reconfiguration time is a function of the size and shape of the reconfigurable region. The configuration data must be sent to the on-board reconfiguration logic and processed. There is latency associated with reading each unit of reconfiguration data from memory, writing to the reconfiguration logic, and processing it. As will be seen in Chapter 4, the reconfigurable region contains a fixed amount of logic based on its size and shape. Therefore, the time to complete a reconfiguration event is tied to the location of the region within the reconfigurable device. Due to the latency of fetching reconfiguration data from external memory, pre-fetching and caching approaches have been widely considered [25], [26].

Another drawback of FPGA-based reconfigurable systems is that due to the way reprogramming files are generated, each functional unit must be pre-compiled so that it is specifically tied to a given location within the device [27]. To be able

to implement a functional unit in an arbitrary location, each region must have an available task implementation that is specifically mapped to that location in the device. This can cause a large number of FPGA bitstreams to be generated for the entire application. Storing extra configuration files represent a significant impact on embedded systems where memory resources are likely to be scarce. There has been some work [28], [29] to allow bitstreams to be placed at generic locations in a FPGA, but even then tasks cannot be arbitrarily placed arbitrarily. This further restricts the ability to dynamically reprogram the device.

These characteristics of realistic reconfigurable systems impose significant constraints on reconfigurable systems. The process of reconfiguration, and it's associated blocking idle time imposes a significant penalty on performance. Also, the data used for reconfiguration is used inefficiently due to the need to replicate it for each reconfigurable region. Further, the reconfigurable images also contain unused logic, which is not used for productive computational work. Improvements in system performance could be realized if the system were to make better use of the time spent reconfiguring. This could be done by reducing the number of tasks, the number of partially configurable regions, or both.

1.4 Problem Statement and Contribution

Reconfigurable computing remains underutilized due to the partial reconfiguration inefficiencies outlined in the preceding sections. This research proposes a compile-time enhancement to improve the makespan of a nondeterministic computing application that could be executed on a reconfigurable computer.

In order to improve the execution of these nondeterministic systems, this dissertation explores the utility of combining reconfigurable tasks into meta-tasks capable of performing the aggregate computation of the constituent tasks. This approach takes advantage of the fact that most tasks fail to use all of the configurable resources available in a given PE. Combining tasks in such a way effectively increases the number of PEs, without increasing the number of reconfigurations required.

The contributions of this work are:

1. Development of grain packing algorithms that can be used to improve the performance of a reconfigurable computer by reducing the number of reconfigurations. This will in turn reduce the idle time caused by reconfiguration events. Further, this approach will minimize the number of FPGA partial bitstreams necessary to implement the system.

2. Quantifying the cost of reconfiguration on a real-world, resource constrianed partially reconfigurable system.

3. Verification of the effectiveness of the clustering algorithms using runtime results from sophisticated discrete event type simulation on dynamically executing, control dependent applications.

1.5 Dissertation Overview

This dissertation is arranged in nine chapters. The discussion is divided among the remaining chapters as follows.

Chapter 2 presents previous work that is relevant to this dissertation. This includes a discussion of reconfigurable systems, application scheduling, and graph clustering techniques that will be adapted later in this dissertation. These concepts have been widely studied in parallel and distributed processing literature. There is much research that is relevant to the task agglomeration approach for reconfigurable systems.

Chapter 3 describes the finite reconfigurable system architecture that will be used as a basis for the remainder of the dissertation. This includes a discussion of the task system model, as well as the practical and simplifying assumptions that govern its operation. Additionally, a reconfigurable execution platform is presented that is capable of executing the application under consideration. This is an extension of a reconfigurable architecture that has been used in previous research.

In Chapter 4, an implementation of a reconfigurable system is developed using a simplified application. The implementation uses real world hardware and a commercially available development board to implement a partially reconfigurable system and run the application. The implemented system is based on a previously used partially reconfigurable architecture with some simplifications to aid the implementation.

Chapter 5 introduces the grain packing technique that will be further developed in chapters 6 and 7. This includes an introduction to the grain packing concept. This approach has been successfully applied to task systems in traditional computing environments with good results. Applying the technique to reconfigurable computers improves several issues relative to the reconfiguration time in the system.

Chapter 6 discusses the simulation framework used to test the grain packing algorithms. Each algorithm is implemented as part of a sophisticated scheduler. Results are generated by comparing each implementation in a static model as well as a dynamic approach that simulates nondeterministic execution.

Chapter 7 discusses approaches for the grain packing technique. Several computational algorithms are developed. After discussion of the computational methods employed, a comparative analysis is performed to evaluate their complexity.

Chapter 8 summarizes the results obtained from running the simulation framework on the grain packed applications and presents the conclusions that can be determined from the results. In Chapter 9, ideas for future work are presented, along with concluding remarks.

CHAPTER 2

BACKGROUND AND PREVIOUS RESEARCH

Reconfigurable systems have been widely studied in previous research. Given their general application to real-time embedded applications, much of this research deals with targeting real-time computing problems. When developing real-time systems, specialized design techniques are generally employed to identify the best system architecture for a particular problem. Consequently, a wide variety of architectures have been explored in reconfigurable computers. These many different architectures also result in an assortment of reconfigurable operating systems and applications. The common theme, however, is that reconfiguration time plays a significant role in reducing the performance of these systems.

2.1 Hardware Software Co-design

Closely related to reconfigurable computing is the area of hardware software co-design [30]. Co-design is a formal method to decompose a system into hardware and software components, and then assign system functions to these components in a way that optimizes the total system performance [31]. Originally, the co-design process was applied to microprocessor based systems that were coupled with static

ASIC accelerators [32], [33]. However, since reconfigurable computers can provide both software execution units and reconfigurable application specific accelerators, the co-design process is well suited to designing reconfigurable systems [34]. Given this relationship between reconfigurable computing and co-design, much reconfigurable computing research has been written from a co-design perspective [35], [36].

When applying co-design to reconfigurable computing, the partitioning and scheduling [37] phases are of primary interest. Some authors also include the initial step of specification in order to highlight the work that is needed to define the application. Another step, temporal partitioning, or clustering, is often required for architectures without partial reconfiguration support. Although specific models for the co-design process may vary, the basic concept relies on: determining what functions the system is required to perform; deciding which portions of the system perform these functions; and deciding the order that these functions will be performed in. When using a co-design methodology for reconfigurable system design, each of these is a key step.

The first of these steps is referred to as either specification or modeling [38]. This is often performed using a defined specification model that may be parsed by automated tools in later stages of the process [39], [40], [41]. The specification step identifies the goals of the system in terms of required functions, execution time, energy efficiency, or other system level parameters [42]. Ideally, these functions should be quantifiable so that alternative designs can be considered and compared to each other.

The process of determining which system elements perform what function is referred to as the partitioning, mapping, or allocation phase [43]. In this phase, tasks

are assigned to specific hardware or software components in the system being designed. Methods for performing mapping are generally straightforward, often relying on heuristic methods to make task to processor assignments [44], [45], [46], [47], [48]. Mapping may also be performed concurrently with the scheduling phase [49], [50] in order to achieve better results and reduce the amount of time needed. Since both problems are NP Hard [51], concurrent processing is more effective than sequential processing. Many varied allocation methodologies have been proposed [52], [53], resulting in a variety of partitioning techniques.

During scheduling, the execution order of the tasks is arranged for each hardware and software processing element. Scheduling takes into account the allocation made in the previous step, as well as the precedence constraints of the application. A variety of mechanisms are usually employed to create a schedule, depending on the characteristics of the application. For non-complex systems, simple list-based methods are generally appropriate [54]. Heuristic approaches are also not uncommon [55]. In more complex systems, scheduling may be performed dynamically at run-time by part of the operating system [56].

Once a system has been allocated and scheduled, cosimulation and cosynthesis may be performed to evaluate the performance of a given design. An objective function is employed to quantitatively evaluate a given system design [57]. Optimization techniques are then employed to find a partitioning and schedule that results in the best value of the objective function. Multi-objective optimization techniques can also be employed when the system is being designed to meet competing objectives.

Cosimulation is the process of evaluating the objective function for the entire system without implementing it directly. Usually, a model will be constructed from the allocated and scheduled design using a designated system modeling language. The complete co-design process is usually performed iteratively to construct different models in order to find the optimum system performance [58]. This model can then be executed on a simulator to evaluate the relative performance of the given design. After finding the optimal system design, cosynthesis [32] is the process of generating the system components. Once a design has been selected, the reconfigurable system hardware and software can be synthesized directly from the model [59].

By applying the co-design process from specification through co-synthesis, reconfigurable systems can be developed using a formal development process. This process ensures that the designed system meets its performance goals. Applying the co-design process iteratively allows the design space to be explored, which gives confidence in finding the best solution.

2.2 Reconfigurable System Architectures

Reconfigurable systems have been hailed as revolutionizing computing [60], but these revolutionary systems have largely failed to materialize [61]. Despite this, there has been a large amount of research performed on reconfigurable computing platforms. Although they have been widely studied, a standard programming paradigm has failed to emerge [62]. However, a wide variety of reconfigurable architectures that vary in scope and capabilities have been developed as a result of the research that has been conducted.

Many different reconfigurable architectures have been proposed in previous research. This ranges from traditional microprocessors which are supplemented by reconfigurable accelerators [63], [64] to complex multiple FPGA systems [65], [66]. With the availability of partial dynamic reconfiguration, entire reconfigurable systems can now be implemented in a single FPGA [67]. These single FPGA systems can be supplemented with a soft microprocessor [68], or an on-chip hard microprocessor [69].

An on-chip microprocessor can greatly improve the performance of a reconfigurable system [70]. Primarily, it increases locality between the processor and reconfigurable hardware, improving the throughput on the communications link. It also provides resources to execute infrequent tasks without the overhead of partial reconfiguration. The microprocessor is also available to execute a software operating system, which can be used to manage the execution of both the software and the hardware tasks [71]. When outfitted with either hard or soft microprocessor and suitable software, a single dynamically reconfigurable modern FPGA system can replace a dedicated and costly purpose-built multi-FPGA architecture.

In addition to the processing architecture, interconnection schemes are also widely studied. Some researchers have even applied the partial reconfiguration technique to the interconnect [72], [73]. However, this approach increases the reconfiguration time, because the programmable interconnect must be setup, and the interconnect to the previous reconfigurable PE must be torn down [74]. This penalty generally leads to the implementation of static interconnect between reconfigurable PEs.

At one extreme are fully connected architectures where each PE has a point-to-point link with every other PE. A drawback of this approach is poor scalability with increasing PEs because the number of links increases exponentially with the number of PEs [75]. Due to the poor scalability, a bus-based interconnect is a common alternative approach. However, a large number of PEs lead to poor performance due to bus contention [76].

To overcome the drawbacks of fully connected and bus architectures, some researchers have focused on using a interconnect approach such as a Network on Chip (NoC) solution [77], [78]. The NoC architecture may also be tiered, in which case the design is partitioned into locally communicating PEs and globally communicating PEs [79]. The local PEs are connected via a fully connected architecture. The groups of local PEs are then globally connected via a switched interconnect network for more infrequent communication outside the local group [80]. The challenge with such an architecture is identifying tasks which frequently communicate, and mapping them to a generic architecture. Finding communities of frequently communicating tasks has been studied to improve the performance of these tiered interconnect architectures [81].

One common problem with all of the proposed architectures is the comparatively long reconfiguration time. Reconfiguration latency has been recognized as a significant impediment to reconfigurable systems. Several techniques have been proposed to minimize this latency [82]. Generally, these methods can be categorized as caching [83], [84] or prefetching [85]. The caching technique relies on storing the bitstream in fast memory with good locality to the reconfiguration logic in order to

minimize transfer time. Prefetching overlaps execution time with the time to access the memory element containing the partial bitstream. Both of these schemes are effective at reducing reconfiguration latency, although prefetching can be complex to apply to nondeterministic systems due to the complexity of predicting stochastic task execution.

Another novel technique to reduce reconfiguration time is known as configuration reuse or configuration locking [86]. Configuration locking is primarily applicable to architectures with multiple PEs. This method works by preventing commonly used tasks from being removed from the device, preferring instead to reconfigure a less commonly used task, even if the incoming task would have to wait for another PE. Some systems have critical tasks that are performed repeatedly, and the application may exhibit poor performance if these tasks are continuously swapped in and out of the device. Performance is improved since the incoming task only waits a short time, but locking the critical task may prevent multiple reconfigurations throughout the application's runtime. Obviously, this is subject to the constraint that there are sufficient number of PEs that can execute the other tasks to prevent deadlock conditions.

2.3 Operating Systems for Reconfigurable Systems

Except in the most trivial cases, reconfigurable systems require some level of operating system (OS) support. Generally, reconfigurable computer OSes are designed to allow a reconfigurable computer the capability to handle one or more dynamic functions at run-time. This includes task scheduling, interprocess communi-

cation [87] processor relocation, or task preemption. A reconfigurable system OS may be implemented fully in software on the microprocessor, or split between software and hardware.

Much of the research in operating systems for reconfigurable systems is focused on run-time task scheduling, which has been mentioned many times [88], [89], [90]. The authors of [91] and [92] provide a very thorough overview of previous work on the subject. Scheduling of reconfigurable systems can broadly be divided into two domains, static schedulers, and dynamic schedulers. Either type of scheduler may include optimizations for a variety of features. Commonly, reconfigurable OSes are designed to meet hard real-time schedules, although optimizing power efficiency is another frequent consideration [93], as well as preventing physical damage to the PEs [44].

Static schedulers are probably the most widely studied type of scheduling method. Much of this work is based on previous work in heterogeneous multiprocessor computing environments [54]. Such schedulers are generally applied to deterministic systems where the precedence amongst tasks is well understood and does not change. Since the precedence constraints do not change, static scheduling can be performed once and then optimized when the system is designed. Since the static schedule is generated beforehand, computationally complex scheduling techniques can be used to optimize the system schedule. List based scheduling methods are widely used static scheduling methods [94], but more sophisticated scheduling methods can also be employed [95].

Dynamic schedulers for reconfigurable computers are less well studied, but are generally required in dynamic systems where task precedence relationships and execution can change during run-time [96], [97]. These types of schedulers are often based on heuristic methods, such as Earliest Deadline First (EDF). Real-time software systems have similar constraints when scheduling execution of software tasks, and many methods exist to schedule such systems. Many scheduling techniques have been proposed by extending relevant concepts from real-time software scheduling. Due to the time required to context switch in reconfigurable computing, the preemptive scheduler is almost never considered.

A real-time operating system (RTOS) framework for reconfigurable systems is proposed in [98]. This RTOS is capable of online scheduling and placement of tasks in the reconfigurable PEs. It also supports multiple implementations of each task, and is capable of selecting different task implementations to optimize different objectives (runtime or power consumption). Further, it is targeted toward a realistic multi-island modular architecture, which is very similar to the architecture this research targets. Although this architecture is very capable, it is still subject to significant reconfiguration time penalties, and does not consider task clustering to alleviate it. The architecture proposed in this dissertation does not make use of a separate placement step, assuming the homogeneous PEs can accept tasks based on the scheduler's decisions. It is believed that focusing on maximizing available PE resources at compile time will yield better results compared to optimizing task placement at runtime.

In [99], the authors propose a novel architecture for a reconfigurable system operating system scheduler. This scheduler is capable of online placement of tasks in a 2-dimensional array of logic resources. However, the model is not suited towards multi-island modular types of reconfigurable architectures. Further, the online placement of tasks generally requires online synthesis of task implementations, which is not realistic under real-time conditions.

The authors of [100] look at the problem of run-time scheduling of nondeterministic real-time applications on dynamically reconfigurable systems. This scheduler is energy aware, and may be used to minimize power consumption while still meeting real-time schedules. Their technique achieves good results and is capable of configuration reuse and configuration pre-fetch. It is also realistic since it is constrained by a limited number of simultaneous reconfigurations.

Another approach is shown in [101]. Instead of simply optimizing runtime, the dynamic scheduler also attempts to take into account possibly faulty hardware. While meeting runtime requirements, the scheduler will attempt to detect damaged hardware and place tasks to avoid those areas. This method is also aware of heterogeneous device resources, and several consequences that arise as a result of them.

A unique capability of reconfigurable computers allows tasks to relocate between a traditional processor and reconfigurable processor array implemented in a single partially reconfigurable FPGA [102]. This concept extends the domain of traditional OS requirements into new areas. In this method, the OS relocation overhead is small, but the act of partial reconfiguration imposes significant delay on the pro-

cess. Although this limits the utility in hard real-time systems, it would be useful in embedded memory constrained systems without real-time requirements.

2.4 Previous Grain Packing Research

Grain Packing has been previously suggested as a means to improve the performance of various scheduling techniques. Clustering of software tasks on multiprocessor systems has been considered for some years [103], [104], [105]. Clustering for reconfigurable systems has been proposed in a few papers, mostly in terms of temporal partitioning. The next section highlights some implementations of grain packing to solve reconfigurable system problems.

The term task agglomeration has been proposed for the process of grouping tasks after partitioning a larger application problem [106]. A key distinction of this type of grain packing approach is that the application tasks do not lose their individual identities (non-immersive grain packing). This is in opposition to immersive grain packing methods where the constituent tasks are combined such that the individuality of each task is lost.

Graph clustering is closely related to the area of graph partitioning [107]. Both of these topics have applications in diverse fields such as computer science, VLSI design [108], biology [109], and music [110]. Computer science applications are generally related to "Big Data" problems of identifying community structure from social networks. Such problems have received a great deal of research in recent years based on the popularity of social networking and availability of data.

The basic concept of both partitioning and clustering relies on identifying community structure in a graph in order to locate closely related nodes together. In the case of graph partitioning, the intent is usually to partition the graph such that the resultant clusters are approximately the same size. However, in the case of cluster identification, the intent is to infer characteristics from a group; therefore the equal size constraint is not generally applicable. Although the cluster identification problem is often applied to undirected graphs.

Modularity [111] is a metric that is often used to determine the "goodness" of a given cluster. This metric measures the prevalence of graph edges in a cluster and compares them to the expected value of connecting edges in a random graph [112]. There are some limitations to this approach, as the algorithm identifies clusters based on the graph edges and does not consider the number of nodes in the cluster. This leads to some special cases where the metric does not correctly identify small clusters [113]. Several algorithms have been proposed to generate clusters based on optimizing the modularity metric. This has been shown to be an NP-hard problem [114].

2.5 Grain Packing in Reconfigurable Systems

Applying the grain packing concept to applications targeting a reconfigurable system has been previously discussed by a number of authors. Most of these discuss grouping tasks to support the temporal partitioning or clustering phase in a co-design methodology. While the benefits of grain packing have been recognized as favorable, each approach is geared toward different outcomes. This results in a wide variety of applications and approaches for the grain packing technique. Since grain packing

at runtime would require the synthesis of new partial bitstreams, these approaches are almost always considered at design time. Most approaches are viewed from a co-design perspective. Several variations exist as to when the clustering step is performed. Proposals have been made to perform clustering as a separate step prior to mapping and scheduling. Others have suggested clustering as part of the mapping and scheduling process.

Temporal partitioning is generally a divisive partitioning method often used with reconfigurable computers that do not support partial reconfiguration [115]. In such architectures, the entire reconfigurable device must be reconfigured, so combining tasks can make better use of the available resources. In this approach, the goal is to divide the overall application into the minimum number of equal-size partitions containing multiple tasks. When a partition is loaded into the reconfigurable device, all of its constituent tasks are available for execution. If a needed task is not in the currently loaded partition, a reconfiguration is performed to load the correct task set. Although this saves some reconfigurations, significant reconfiguration time is required unless the reconfigurable device is sufficiently large to contain all of the application's tasks in a single partition.

In [116], a partitioning method is employed on a static graph with a predetermined number of temporal partitions. The authors of [117] propose a clustering approach that is similar to the technique that was developed for this dissertation. However, their approach applies the clustering step one time only before the task graph is scheduled. The technique that will be discussed iteratively engages the scheduler to determine the effectiveness of the packed schedule.

The authors of [74] point out that significant space may be wasted in the unused area of a tiled reconfigurable system. To combat this, they propose combining multiple tasks into a single partial bitstream. They also note that significant bitstream storage would be wasted from generating and storing all possible task combinations. This approach is mainly targeted to design time optimizations and does not examine the effects of their approach on system execution.

In [118], the authors describe a methodology to map an application onto a NoC to improve communication performance. This is accomplished by combining high communication cost tasks into small System-on-Chip (SoC) like clusters that then connect to a larger NoC. This technique is primarily concerned with optimizing the inter-task communication by minimizing the communication distance between regularly communicating tasks. The authors of [119] develop a very capable algorithm for clustering tasks as part of a co-design process. However, their approach relies on being able to effectively profile the system's operation. While they do apply this approach to a heterogeneous reconfigurable architecture, they do not apply it to any dynamically reconfigurable systems.

The authors of [120], [121] describe an algorithm to group tasks into clusters in order to improve system performance. This algorithm targets an architecture composed of a grid based reconfigurable array connected via an inter-region NoC. The algorithm is divided into a static component and a run-time component. The static component performs better but is not capable of assigning new tasks at runtime. The dynamic component does not perform as well, but can assign tasks that are part of a new application. One of the goals of this approach is to promote reuse of tasks across

multiple applications. Results are based on a reduction in reconfiguration latency and amount of task reuse. Again, this approach is not concerned with nondeterministic task graphs.

In [122], the authors propose two methods to combine tasks during mapping and scheduling. This approach does not attempt to deal with nondeterministic execution, assuming that the graph precedence remains constant. The authors of [49] also look at a combined mapping scheduling approach that pre-partitions the graph and iteratively attempts to merge the partitions. While the approach produces significant application speedup, it does not consider nondeterministic execution.

Task clustering is shown in [123] as a way to reduce reconfiguration time. Interestingly, this paper also looks at splitting tasks to be able to fit within the resource constraints of PEs. However, the primary focus of this paper is on optimizing the interconnects and the resultant effects of communication on performance.

A dynamically reconfigurable clustering technique is proposed in [121]. This approach is similar to the technique that will be presented. However, the authors assume that the reconfigurable architecture is large enough to contain the entire application. In their approach, they use a tiered NoC Approach. Tasks are grouped into clusters interconnected with a network switch and then assigned to a reconfigurable slot, which is connected to the other slots via inter-slot network switches. This system allows dynamic bitstream generation and is primarily concerned with improving area utilization, and adding capabilities at run time. None of the previous research examined considers the limited resource case where the reconfigurable system needs to be shared to implement the entire application.

Although many clustering techniques have been proposed and implemented for reconfigurable systems, the effects of clustering on graphs with nondeterministic execution profiles have not been examined. Additionally, the resource constrained case is often considered in co-design contexts, but the combination of a nondeterministic application executing on a resource constrained platform appears to be unique.

CHAPTER 3

SYSTEM AND APPLICATION ARCHITECTURE

When dealing with reconfigurable systems it is useful to differentiate between the reconfigurable platform and the application executing on this platform. Modeling the platform specifies the number and type of execution units, their interconnection scheme, and any ancillary resources, such as local and global memory structure. The application model defines the individual tasks and their precedence constraints along with the scheduling constraints, such as hard real-time or periodic scheduling. Collectively, these two models can be used to estimate an application's execution.

3.1 Introduction

Key to the discussion of the platform architectural model are the capabilities of the PEs. As noted previously, reconfigurable computers typically include one or more FPGAs. Recent product offerings include hard microprocessors; examples of such chips include the Xilinx Virtex 6, the Xilinx Zynq, or the Altera Cyclone V. The first may be obtained with up to 2 PowerPC processors, while the latter two provide ARM microprocessors. This heterogeneous architecture allows development

of complex reconfigurable systems that contain both software and hardware PEs on a single chip.

Most of these FPGAs are partially reconfigurable devices, allowing multiple hardware PEs to be realized in a single device. With this approach, the device is partitioned into a number of partially reconfigurable regions that perform the computational work of the system. Since these partitions are partially reconfigurable, they may be modified at run-time without affecting the computation being performed in the other regions. This is a very powerful technique which allows a single FPGA to replace a system that formerly would have required several discrete chips to implement.

Partially reconfigurable partitions are distributed throughout the device with an interconnection scheme to allow them to communicate with each other and any microprocessors that may be available. Collectively the microprocessor(s) and reconfigurable regions represent the PEs of the system. All computational work is performed in a PE of the appropriate type. Microprocessors execute software tasks, while the reconfigurable tiles execute hardware tasks. The partitions are the minimum reconfigurable unit of the reconfigurable device. If the device is not partially reconfigurable, it is functionally equivalent to a single hardware PE with the resources of the entire device.

There is no requirement for the reconfigurable tiles to be heterogeneous in terms of size. In fact, the system may contain many different PEs with different capabilities; every PE may be a unique partition of the device. Each PE partition has a limited number of reconfigurable resources available for processing. These are

classified into two principal categories: routing resources and logic resources. It is assumed that a reconfigurable task implementation consumes a larger share of available logic resources than routing resources. Logic resources are therefore considered to be the limiting constraint when associating a task with a reconfigurable PE.

Reconfigurable architectures may be classified according to the arrangement of the reconfigurable areas [124]. In island-type architectures, there is a single reconfigurable region that may contain exactly one task. If the region contains more resources than the task consumes, the extra resources go unused. Multi-island architectures may be built by including more reconfigurable regions. In order to reduce the problem of unused resources in island architectures, slot and grid architectures have been proposed. In these architectures, a single large reconfigurable region is implemented in the device. This large region is then divided into individual reconfigurable regions, similar to the way memory may be dynamically allocated from a statically allocated array. If the region is divided along one dimension only, it is referred to as a slot architecture, whereas a region divided into small two-dimensional regions is referred to as a grid architecture. Grid architectures hold the potential for the best resource usage, but slot architectures are particularly suited to systolic and pipeline based data processing. However, given the heterogeneous nature of modern FPGAs, slot and grid architectures are difficult to implement without detailed knowledge of the underlying hardware resources.

3.2 Platform Architectural Model

This research targets a multi-island architecture, discussed previously as the Dynamic Hybrid Reconfigurable Architecture (DHRA) [125], [126]. This architecture was selected because it represents a realistic architecture based on current device capabilities, and because of the availability of data on both the architecture and target applications. The DHRA architecture consists of RSoPC with an embedded microprocessor. The chip is partially reconfigurable and implements a number of partially reconfigurable PEs. Logic is included in the DHRA to efficiently manage the reconfiguration of the logic cells, and on-chip caches to store reconfiguration bitstreams that are likely to be used soon. Dedicated buffers are provided to store data between context switches.

Figure 3.1 depicts the DHRA. This platform provides a dedicated hard microprocessor for executing software tasks, a varying number of reconfigurable PEs for executing hardware tasks, and static logic to support execution of the application on the platform. The static logic includes the Dynamic Task Scheduling Unit (DTSU), which is a hardware implemented scheduler for the PEs, which is also responsible for initiating the process of partial reconfiguration. The various hardware PEs are fully connected to each other and to the memory elements using a crossbar interconnect, allowing for immediate communication between PEs. In the figure, the crossbar interconnect is depicted as a collection of buses for simplicity.

The general nature of the DHRA system easily lends itself to simulation as well as direct implementation. The island-based PE structure is easily replicated in

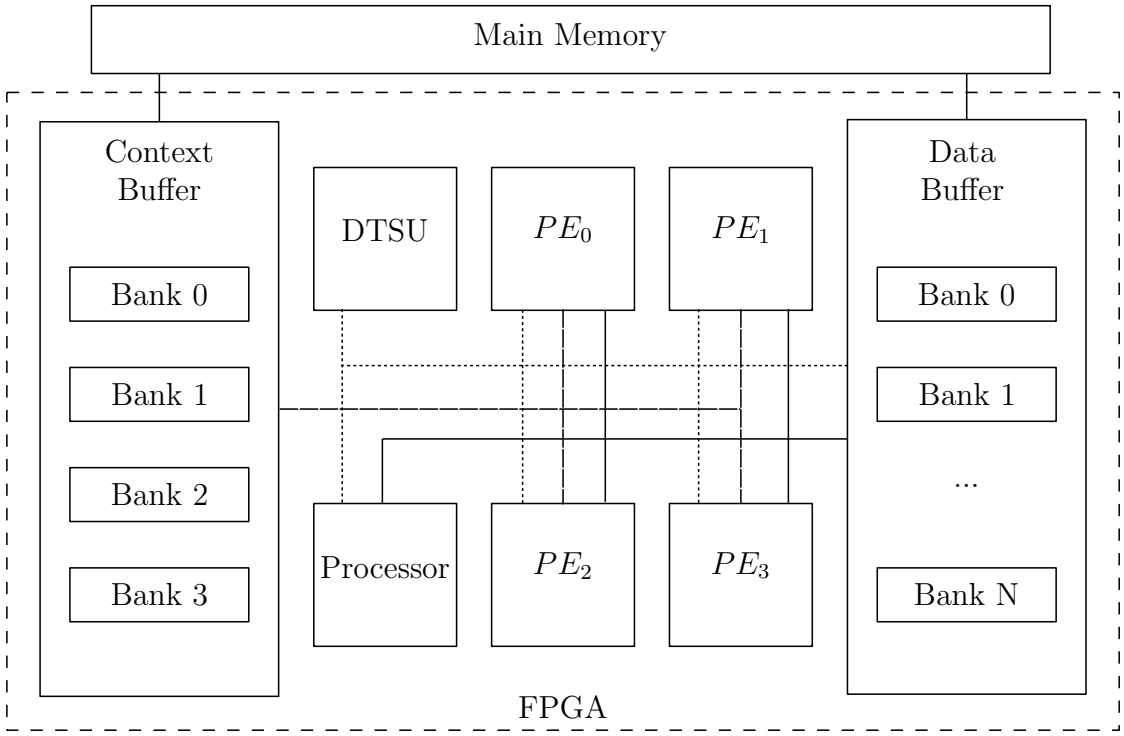


Figure 3.1: Example Reconfigurable System

most recent FPGAs that support dynamic partial reconfiguration, such as the Xilinx Zynq or Altera Stratix. Without loss of generality, the processor may be replaced with a soft-core microprocessor if an on-chip hard processor is not available. A soft microprocessor extends this architecture to less complex FPGAs that do not contain an embedded hard-core microprocessor.

The basic function of the system has been well described, and effective simulations of its performance have been described previously [125]. The crossbar interconnect simplifies the design of the system, so that there is no need to model bus contention within the device. For a moderate number of PEs, modern FPGAs provide enough programmable interconnect to allow implementation of such a fully connected architecture.

Central to this architecture is the DTSU, which is responsible for the management of the hardware PEs. Each PE provides a dedicated set of signals to the DTSU, which uses an internal state machine to properly orchestrate the activities necessary to execute an application. Using a simplified representation of the application's precedence constraints, the DTSU will step through the process of configuring a PE, context switching to the next task, executing the task, and saving the context for the next task or reconfiguring the PE to the next task if required.

To limit the reconfiguration delay, the architecture also includes a context buffer to hold the partial bitstreams. This buffer provides local memory for the DTSU to store partial bitstreams so that accesses to main memory are not required for reconfiguration, which would further increase the reconfiguration time. When reconfiguration of one of the HW PEs is required, the DTSU model assumes the context buffer has an appropriate bitstream available, and there is no penalty for retrieving it beyond the reconfiguration time. The DTSU does not control the movement of bitstreams between the context buffer and main memory. From the DTSU perspective, a task's bitstream is assumed to always be available in the context buffer.

Internal to the DTSU is a pre-fetch module. This module is responsible for configuring idle PEs with tasks that are not yet ready to execute, but are likely to be executed soon. Since the scheduler is aware of the execution state of the hardware PEs, it can determine if an idle PE has a waiting task assigned to it or not. If an idle PE has no waiting task assigned, the DTSU will reconfigure the idle PE with a task that is expected to execute soon.

The DTSU architecture also provides data buffering for the storage of the results of each hardware task. Since the partial reconfiguration process completely erases the contents of the partially reconfigurable PE, external storage is required when data must pass between two different tasks that are executed consecutively on the same PE, or when both PEs will begin reconfiguration before the consuming task starts execution. When a task completes execution on a PE, the results are saved to the data buffer, where they can be retrieved by other tasks that are data dependent on the first task's results. Buffering is not required between two PEs that have been configured with communicating tasks. Due to the fully connected architecture, these PEs may pass data directly between each other if ready to communicate.

The hardware PEs are of identical size and construction. In addition to simplifying the architecture implementation, identically sized PEs allow the context buffer to allocate an equal amount of memory to each PE for reconfiguration data. Since the PEs have the same resource allocations, they are therefore assumed to execute tasks identically. Even though the PEs are identical, the system must still have access to different partial bitstreams for each task and for each PE it executes on.

The final component of the DHRA architecture is the microprocessor. The microprocessor is provided for the execution of software tasks. Assignment of tasks to hardware or software PEs is assumed to be performed as part of the partitioning process prior to executing the application. Software tasks are scheduled and managed by code executing on the microprocessor, generally by a Real-Time Operating System (RTOS). Execution models for software tasks on microprocessors have been considered extensively, and will not be considered further.

3.3 Application Model

Development of the system application is provided by the specification stage of the design process. The output of this process is the application modeled as a directed acyclic graph (DAG) where individual graph nodes are connected via directed edges. In this graph, the nodes represent the individual tasks that the application is composed of. Task execution order is given by the directed edges of the graph. Other than precedence, the graph representation of the application does not impose any execution provision on the individual tasks. Any task may be executed on any available PE subject to the availability of a suitable implementation for that PE.

Although the graph is acyclic, this does not limit the application to strictly acyclic applications. Periodic behavior can be modeled by allowing the graph to represent a single instance of a cyclic graph, such that the graph represents a single instantiation (frame) of a periodic application. The application may be instantiated multiple times to perform periodic processing. Likewise, several minor frames can be concatenated to replicate a periodic application that has modal properties. Interior loops may be unrolled to eliminate cyclic behavior internal to the graph.

The task nodes are defined by their task ID, task type, execution time, and resource usage. A task's execution time is simply the time the task requires to perform processing once it begins execution. For some tasks, the execution time is not a constant value, but rather depends on the set of input data. In this case, the execution time is specified as a range that bounds the actual execution time. Resource usage is related to the amount of computational resources required to complete processing.

For software tasks, this is analogous to physical memory, while it is assumed to be related to FPGA area (logic resources) for tasks implemented in hardware PEs.

In general, the execution time of a task is dependent on several factors. Firstly, it is assumed that a task will execute faster in a hardware PE than in software. However, there are usually many possible implementations of a task in hardware, which may not execute in the same amount of time. Some tasks may execute faster in larger partitions because there are more resources available. Additionally, some partitions may contain heterogeneous resources, such as embedded multipliers and memory that may improve the performance of some tasks mapped to those partitions. The DHRA model assumes that each PE is homogeneous in terms of size and resources, consequently, a single task implementation will be used for each partition. Therefore each task is assumed to execute identically regardless of the partition it is placed in.

Graph nodes are connected by directed edges such that the tasks are executed in the order implied by the direction of the edge. Edges are classified according to whether they imply a data dependency or a control dependency. A data dependency between two tasks implies that the dependent task relies on the data result of the source task. Likewise, a control dependent task relies on a control decision from the source task. Dependent tasks of either type require the completion of the source task before they may begin execution.

Including control dependencies implies a nondeterministic system. A control source task determines that the sink task will or will not run; this introduces nondeterministic execution into the application depending on the result produced by the source task. A bounded nondeterministic model is used to simplify the implemen-

tation. In this model, it is assumed that all possible tasks that could exist in the graph are known a priori. However, it will not be known if a particular control dependent task will be run until all of the task's control dependencies have been met (i.e. once the control dependency source tasks have completed execution). To model the nondeterministic properties, control edges are assigned a probabilistic weight to indicate the likelihood that the sink task will be performed. This weight indicates the proportion of the overall instantiations (frames) that the graph will appear in.

This overall description may be used to accurately describe many real-time and embedded systems where control decisions affect the performance of the system. In a normal execution condition, the application performs a defined set of operations. However, occasionally, an off-nominal execution path is performed (e.g. to recover from a detected error). In this case, all tasks are known prior to executing the application. Based on an examination of either the application or data set, an estimation can be made of the relative frequency of the nondeterministically executed tasks, which can be used to determine probabilities for their inclusion in a given application iteration.

A hypothetical graph is shown in Figure 3.2 in order to illustrate the model. This example shows a directed graph composed of 5 tasks. The nodes of the graph represent the individual tasks of the application. Even though there are 5 nodes, there are 4 individual tasks. Task 2 and Task 5 are the same type, and therefore share the same implementation, but operate on different data. The directed edges represent the dependencies between the tasks. The solid lines indicate data dependencies, while the dashed line represents a control dependency. In the example, the control dependency

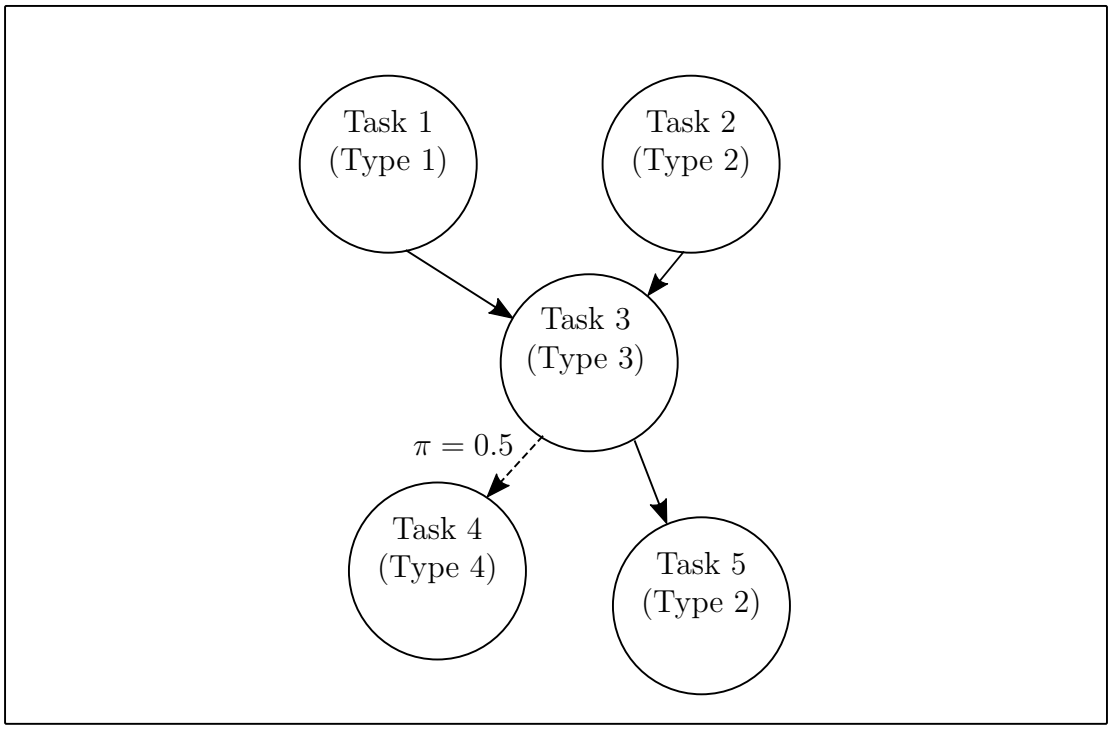


Figure 3.2: Example Application Graph

between Task 3 and Task 4 has an edge weight of 0.5, indicating that this dependency would appear on average in half of the instantiations of this graph.

To formalize the notation, the graph is composed of individual task nodes $n_i, i \in (0, 1, \ldots, m)$. Two nodes n_i and n_j may be joined by the individual directed data edges, $e_{d_{i,j}}$ or individual directed control edges $e_{c_{i,j}}$. The directed control edges have individual probabilities π_k. Therefore, the entire directed graph can be represented by the tuple $\Gamma = \{N, E_d, E_c\}$ where $n_i \in N$, $e_{d_{i,j}} \in E_d$, $\{e_{c_{i,j}}, \pi_k\} \in E_c$.

CHAPTER 4

QUANTIFYING PARTIALLY RECONFIGURABLE SYSTEM CHARACTERISTICS

In order to evaluate the validity of the assumptions made about the reconfigurable system model, an example reconfigurable system was developed and implemented in a commercial-off-the-shelf FPGA system. This system was designed to replicate a reconfigurable computer with a single PE and a single microprocessor. Observations were then made using this reconfigurable system to quantify and justify the model assumptions.

4.1 Reconfigurable Design Flow

The design flow applicable to implementing a reconfigurable system on an FPGA is more complex than a static FPGA system. Design partitioning and placement are required activities in addition to a standard compile process which includes analysis, synthesis, place & route, and post-fitting timing analysis. Additionally, the compiler must be configured to differentiate between the various reconfigurable targets. The example system developed employs a specific FPGA developed by Altera Corporation, therefore, the following information is specific to the Altera com-

piler, Quartus II, and its associated tool QSys [127]. However, at a high level, the Xilinx toolchain can employ a similar process to develop a partially reconfigurable system [27].

The general approach to the design of reconfigurable systems is to compile an independent design for each of the task implementations, which are referred to as design *personas*. Once every persona has been implemented, partial bitstreams are generated, one for each combination of persona and reconfigurable region. The partial bitstreams are uniquely addressed to a specific location within the device, so each partition requires its own set of bitstreams.

In order to help manage the complexity imposed by multiple personas, the compiler uses a concept known as *revisioning* to keep the different implementations separate. Since only a few design components are modified for partial reconfiguration, these modifications can be tracked as differences to a base implementation. The process of revisioning involves establishing an initial design with the components that are common to each revision. Using special compiler commands, the description of the initial design is copied into a new revision. Changes are made to the new revision by adding or removing design components as necessary to implement the intended functionality. Multiple implementations can be managed by repeatedly copying the initial design and making appropriate changes.

It is important to note that for a partially reconfigurable system implemented in an FPGA, the design must be divided into a static component which remains active at all times, and a dynamic component, which is the portion of the design that can be modified during execution. Partitioning is the process that divides the

system into static and reconfigurable regions. It also serves to inform the compiler of the reconfigurable logic boundary. Since the logic may be reconfigured arbitrarily without the compiler's knowledge, the compiler cannot perform optimizations across this boundary. The partitioning process also informs the compiler to not re-compile logic in one partition if it has not been modified.

The *placement* process serves to locate the partially reconfigurable region at a specific location within the device and identifies the region as partially reconfigurable so that partial reconfiguration bitstreams can be generated later. The partial reconfiguration area must be mapped to a specific physical location within the device so that the different personas will not interfere with the static logic implemented in the rest of the FPGA.

To implement the partial reconfiguration partition, an interface is required in the static region to control the input and output that crosses the static/dynamic partition boundary. This can be performed by some special purpose logic that monitors the reconfiguration control signals and sets the I/O lines to a known state when partial reconfiguration is being performed. Such logic can be implemented in an RTL language by instantiating the partial reconfiguration component inside another component so that the reconfigurable component's I/O must pass through this outer wrapper component. Logic inside the wrapper component is responsible for determining when partial reconfiguration is in progress, and setting the I/O appropriately.

Once all of the different revisions of the design have been implemented and compiled, partial bitstreams must be generated for each of the different partial reconfiguration personas. In the case of the Altera FPGA used, a two-step process is

used. First, the full chip bitstream is combined with a mask file in order to derive the portion of the bitstream that is applicable to the reconfigurable partition. The mask file is generated automatically by the compiler using the placement information it was provided in the placement step. After masking, the resultant file is written into a binary format which is appropriate to transfer directly to the on-chip reconfiguration controller. During the translation to binary format, a number of options can be specified, such as bitstream encryption and compression, which modify the data written to the output file. In this example system, neither option was used.

4.2 Implementation Hardware Description

The example reconfigurable system was implemented using a Terasic Technologies/Altera DE–5 development board [128]. This board contains an Altera Stratix V GX FPGA, which contains more than 620,000 logic elements. In addition to the Stratix V FPGA, the DE–5 board provides several general purpose computing facilities which are available to the FPGA. These include dedicated RAM and non-volatile memory, special purpose communication interfaces, and standard display mechanisms. To make the example system as generic as possible, very few of these capabilities are used. The exceptions to this are 4 LEDs used to display status and a USB to JTAG bridge which serves to perform initial FPGA configuration and provide PC-based communication with the system.

The DE–5 board provides several banks of flash memory, which could be used to store the partial bitstream information. However, one of the goals of the system is to measure the time to perform partial reconfiguration without memory access

overhead. To accommodate this, the PR bitstreams are stored within the FPGA. This eliminates the need to retrieve data from external memory, along with the accompanying overhead delay. Since on-chip memory is being used, the size of the reconfigurable region is limited by the amount of memory available to store bitstream data. This constraint can be overcome by storing the bitstreams off-chip and using a pre-fetch and caching scheme to load them into on-chip memory as needed.

4.3 Reconfigurable System Description

Figure 4.1 depicts the architecture of the example system, which was implemented entirely inside a single FPGA. The various components are shown along with their interconnect. Solid lines are used to designate the Processor Local Bus (PLB), while dashed lines indicate a direct point-to-point data path. This architecture was developed to mimic the Dynamic Hybrid Reconfigurable Architecture [125], [126] described in the previous chapter.

There are four main components of this system that model an example reconfigurable computer architecture. These components are: a Nios II system on programmable chip (SoPC), a memory for storing the partial reconfiguration bitstream, a reconfiguration controller, and the reconfigurable region with associated wrapper logic. A fifth component, a timer, has been included to instrument the partial reconfiguration process. Only the Nios processor and reconfigurable logic components contribute to the processing performed by the system. The other components are necessary to implement partial reconfiguration, and represent overhead on the design. Overhead for the reconfiguration controller is incurred only once per design, but the

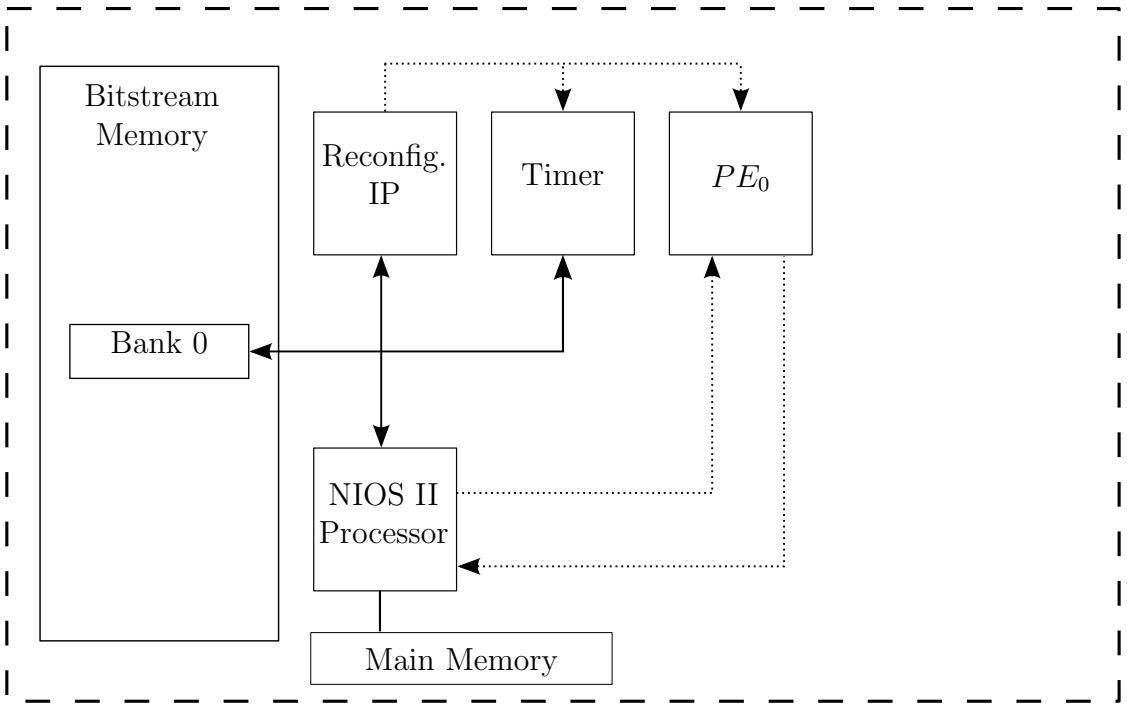


Figure 4.1: Example Reconfigurable Architecture

requirements for the bitstream memory increase with the number of bitstreams that are required to be stored on chip.

The architecture was designed to maximize the flexibility of the reconfiguration process, therefore, the approach taken was to use a soft-core microprocessor to read the partial bitstream from memory and write it to the partial reconfiguration controller. This imposes a delay, as during each iteration data is fetched from memory, data is written to the reconfiguration controller, the reconfiguration status is checked, the loop counter is updated, and the loop is executed again. A more efficient design could be realized by implementing a direct-memory-access (DMA) device to automatically stream data from the bitstream memory to the reconfiguration controller, and should be considered in a real-world, design. However, the use of the

DMA device complicates the ability to do off-nominal testing, such as starting and stopping the flow of data to the reconfiguration controller. These conditions are not currently considered, but the architecture supports examining them in future work.

Further improvements could also be achieved by building custom logic to automatically stream the bitstream data directly to the reconfiguration controller without use of the PLB. The drawback of this approach is that it does not maximize design reuse. Excepting the hardware accelerator component, the entire system was developed from Intellectual Property (IP) components, typical of what would be developed in a real-world system.

The Nios II and it's embedded peripherals were instantiated using the QSys tool in the Altera Quartus II design software [127]. The reconfiguration controller was implemented using an IP core provided by Altera. Bitstream storage was also implemented using an IP memory component. Each of the IP components was instantiated with the QSys tool so that they would be accessible to the Nios processor via the PLB. The LC was instantiated using components that were modeled in VHDL to implement the wrapper logic and arithmetic function. Likewise, the instrumentation timer was modeled in VHDL, but it was instantiated as a custom component in QSys to make it available on the processor local bus.

4.3.1 Nios II Processor System

The Nios system consists of a Nios II Standard microprocessor, on-chip memory, JTAG UART and several parallel I/O modules. The JTAG UART is used to provide a terminal console for the software running on the Nios microprocessor and

is accessible on the development system through the on-board JTAG chain. The soft processor runs a custom console application that allows a user to command reconfiguration events from the terminal. The SoPC also contains a parallel I/O module to drive status LEDs and a timer module to determine the timing of internal processor events. All of the components are available on the Avalon PLB, which is the internal bus standard for Altera SoPCs.

In addition to the basic processor peripherals, two parallel I/O modules are used to interface with the reconfigurable region so that the design of the wrapper logic can be simplified. There is a 16-bit output module and a 32-bit input module, which allows the processor to perform 16-bit writes and 32-bit reads without using the Nios PLB. Since the reconfigurable PE does not use the Avalon bus, it does not require any special logic to ensure consistent bus operation during the reconfiguration event; it only has to ensure the state of the individual output bits.

4.3.2 Bitstream Memory

A dedicated memory is used to store the partial bitstreams used by the system for reconfiguration. This bitstream memory is composed of several memory IP components that are used to store the partial bitstreams for each of the system personas. These memory components are not writeable from the PLB, and therefore look to the processor like a ROM. The system instantiates a ROM for each persona that can be implemented in the LC. Using dedicated ROMs allow the bitstreams to be stored on-chip, without the need to store them in the code segment of the Nios application. Since the ROM's content can be specified at compile time, the design can be syn-

thesized so that the ROM is initialized when the FPGA is programmed, eliminating the need to stream bitstream data into the memory after the system is operational. Furthermore, using a dedicated ROM instead of main memory, which is implemented in RAM prevents accidental corruption of the bitstream data by the soft processor.

The memory is initialized at compile time to contain the partial bitstream. This adds a step to the compile process since partial bitstreams are not available until after the system has been fully compiled. Once each persona has been generated, the partial bitstreams are created using the masking process and are then converted into a memory initialization file. The design is then updated to supply the memory file to ROM component, and the design is recompiled with the initialized ROM.

One drawback to storing the bitstream in on-chip ROM is that there is a limited amount of memory resources available within the FPGA. This limits the size of the partial bitstream that can be stored in the ROM, and consequently the area that can be partially reconfigured is also limited (52 Mbits of block memory in the largest Stratix V GX device). A more realistic partial reconfigurable system would solve this problem by storing the bitstreams in external memory, and use the on-chip memory as a bitstream cache. For this example, the Stratix V device provides enough memory elements to support the model validation.

4.3.3 Reconfiguration Controller IP

The reconfiguration controller is an instantiation of the Altera Partial Reconfiguration Megafunction [129]. This IP component performs all of the processing necessary to reconfigure portions of the device. This includes generating a signal to

activate the I/O freeze logic in the partial reconfiguration wrapper block, and a data bus interface to read data from the Bitstream ROM. Additionally, it abstracts FPGA hardware that performs CRC calculation and low-level reconfiguration hardware that is necessary to implement partial reconfiguration in the Stratix V device.

The Partial Reconfiguration component includes an Avalon bus interface, which is used to allow the processor to control partial reconfiguration. When instantiated with an Avalon interface, the component presents two 16-bit registers to the bus. The first register is used to accept data over the Avalon bus. The appropriate bus master writes bitstream data 16 bits at a time to this address while partial reconfiguration is in process. The second register is a command and status register (CSR), which is used to issue commands to the component, and to monitor for a partial reconfiguration error. The lower two bits of the CSR are used for commands, while bits 4–2 are used to report success or failure of the partial reconfiguration process. The remaining CSR bits are unused.

The reconfiguration controller is activated by writing 0x0001 to the CSR register. The reconfiguration controller will then initiate the partial reconfiguration process by setting a discrete FREEZE signal high. FREEZE is routed to the PE to signal that the I/O should be stabilized prior to partial reconfiguration beginning. The CSR status bits are then set to the value "PR IN PROGRESS". The Avalon bus master (processor) begins writing bitstream data to the data register until all data has been transmitted. If reconfiguration was successful, the CSR status bits will be updated to "PR PASSED", and FREEZE will be deasserted. In the event of a failed reconfiguration, the CSR will contain an enumerated value once the error

has been detected. This error corresponds to one of: PR ERROR, CRC ERROR or INCOMPATIBLE BITSTREAM . When reconfiguration fails, FREEZE remains asserted until the reconfiguration controller's data path has been cleared by writing successive 0x0000 to the data register.

When configured to use the Avalon bus, the PR megafunction does not include a mechanism to implement flow control. The megafuction can accept a 16-bit word on each write cycle. If a 16-bit word is not ready, the megafunction is stalled using the internal Avalon bus mechanisms until the next word is ready.

4.3.4 Processing Element

The PE is divided into two major components. Primarily, it implements one of two arithmetic functions, which are reconfigurable. In addition to the reconfigurable region, a wrapper component is required to isolate the region's I/O from the rest of the design during reconfiguration. The wrapper component has been implemented as a VHDL component, named FREEZE BLOCK, in the top-level hierarchy of the design. The wrapper, in turn, instantiates a VHDL component for the arithmetic function. The wrapper logic is controlled by the FREEZE signal generated by the partial reconfiguration megafunction. When reconfiguration is in process, FREEZE is asserted high by the reconfiguration controller. When the freeze wrapper detects that the FREEZE signal is high, it drives the wrapper's outputs high until FREEZE is de-asserted. When FREEZE is low, the arithmetic component's outputs are passed through unmodified. Driving the outputs in such a way ensures that the I/O is stable

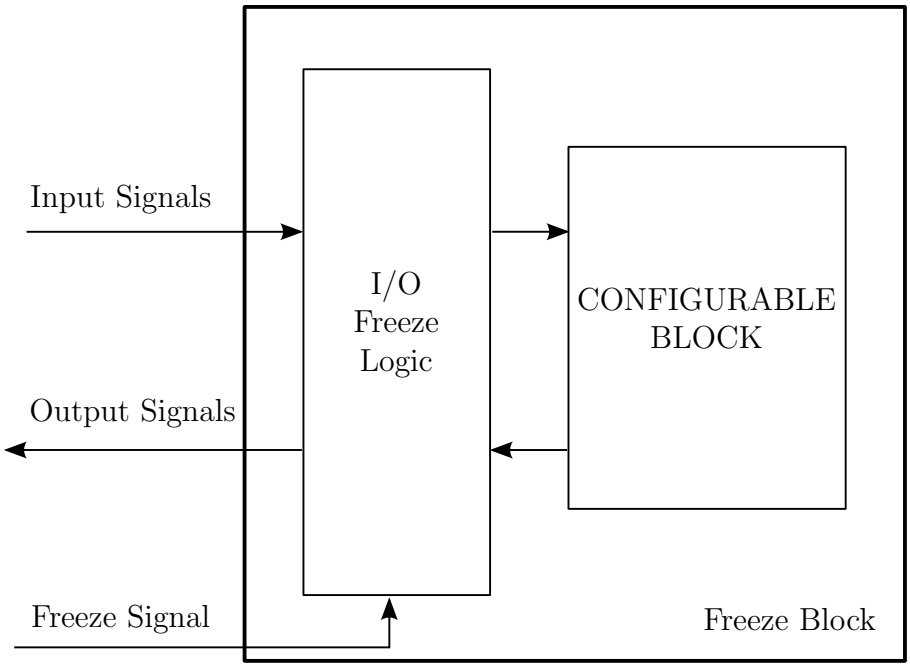


Figure 4.2: Reconfigurable Tile Architecture

while the region is being reconfigured, preventing unstable I/O from disrupting the rest of the system.

Internal to the FREEZE BLOCK component is another VHDL component named CONFIGURABLE BLOCK. This component implements the actual reconfigurable logic representing the application task. Two separate design files are used to provide two reconfigurable functions. Each variant of the logic implements a component named CONFIGURABLE BLOCK, although with different implementations. Each design revision has one version of this component, which implements an arithmetic function appropriate to the persona assigned to the revision. Since the I/O interface between the wrapper and rest of the system must remain fixed, the number and type of I/O's to each arithmetic component must be identical. For this system, the CONFIGURABLE_BLOCK component implements an arithmetic function,

either an adder or a multiplier. Figure 4.2 depicts the architecture of the reconfigurable region, including the Freeze Wrapper component which is instantiated in the static region.

The arithmetic function component takes two 8 bit inputs and produces a 16-bit output. The parallel I/O from the Nios SoPC is mapped so that the lower 8 bits corresponds to the first operand, and the upper 8 bits correspond to the second operand. The functional block also provides a 16-bit identification register so that the application software can determine which functional unit is loaded. Each implementation has a unique ID value hard coded. Since the freeze wrapper sets all output signals high when FREEZE is active, the ID register will read 0xFFFF when reconfiguration is in process. This is useful since the FREEZE signal is not seen directly by the processor. This value should therefore not be implemented in the functional block as an actual ID. Like the input signals, the ID and results values are routed so that the 32-bit Parallel input can be used to read the component output. The lower 16 bits map to the arithmetic result, while the upper 16 bits map to the ID register.

4.3.5 Instrumentation Block

In order to determine the performance of the partial reconfiguration process, an instrumentation block was developed to take timing measurements. Using dedicated hardware, more accurate timing measurements can be taken than by using the microprocessor clock since the hardware does not have to account for the overhead for software to initiate and stop the timer. The timer block takes as its inputs a clock source and an enable signal. When ENABLE is asserted, the timer begins counting

the number of rising edges on the input clock. Once ENABLE is de-asserted, the counter retains its last value. This counter includes a single 32-bit output register, which provides the counter's value to the processor via the PLB. To time the PR function, FREEZE is used to drive ENABLE on the timer module.

The FREEZE signal provides an excellent means to measure reconfiguration time. Since the instrumentation module provides the duration of the FREEZE signal, the reconfiguration time can be established from the value in the instrumentation timer's output register. Using this method gives a system level estimation of the partial reconfiguration time. Although it is possible that the arithmetic function may be available for computation before de-asserting FREEZE, the I/O is unavailable because the freeze signal remains asserted until the entire LC has been configured. Therefore, the partial reconfiguration component cannot be utilized until at least the falling edge of FREEZE. This timing method includes the time to send the data to the partial reconfiguration controller, as well as the time for the reconfiguration controller to process the data.

4.3.6 System Software

The partial reconfiguration process is completely controlled by software running on the Nios II processor. This increases the flexibility of the system since changes to the partial reconfiguration process can be implemented by recompiling the program instead of re-synthesizing the FPGA image.

After initializing all hardware components, the software waits for a user command to start the reconfiguration process. Once the user has initiated partial re-

configuration, the software writes to the PR megafunction CSR to initiate a partial reconfiguration event. It then starts by reading the first word from the ROM containing the partial bitstream, reading the CSR to ensure an error has not occurred, and then writing the bitstream word to the reconfiguration controller's data register. If the CSR indicates that an error has occurred, or that PR is complete, the software writes several NULL values to the data register to clear the controller's data path. Finally, the software reads the timer value from the instrumentation timer and reads the discrete I/O values. It then displays the reconfiguration time, the values read from the arithmetic result register, and ID value register on the console. The user can then determine if partial reconfiguration has occurred successfully.

4.4 Experimental Results

In order to determine the total time associated with performing partial reconfiguration, a series of experiments were conducted. The example reconfigurable system introduced in the preceding section was compiled for the DE–5 development board [128], and used the instrumentation timer to measure the time required to reconfigure the region.

The instrumentation timer does not directly measure the absolute wall clock time required to perform partial reconfiguration, but rather counts clock ticks while the ENABLE signal is asserted. For the experiments, the clock frequency used for the instrumentation timer was 50MHz. Absolute time can be determined by multiplying the timer count by the clock period.

Because of the limited space available in the Bitstream ROM, timing measurements are made on small bitstreams, and this data is used to extrapolate the timing of larger bitstreams. Although there is a practical limit on the size of a partial bitstream that can be stored internal to the FPGA, the compiler can still be used to generate partial bitstreams larger than can be stored on chip.

The experiments conducted are intended to examine both the size of the reconfigurable region, as well as it's location within the device. To accomplish this, bitstreams were generated by both varying the location of the origin in the placement tool and additionally varying the region's height and width independently.

4.4.1 Reconfiguration Timing

Table 4.1 & Table 4.2 report upon the results of the direct timing experiments. In each case, after a successful partial reconfiguration, the Nios processor was used to read the value of the timer component. Two series of experiments were run, the first experiment used a square reconfigurable region, while the second one used a rectangular region. The tables show the bitstream size and timing data for each region size. Each experiment was repeated five times for each region size, and the average timer value is reported in the table.

As can be seen, the size for the partial bitstream is only sensitive to the width of the reconfigurable region, and not its height. Bitstream size also increases in a highly linear manner with respect to the width. In each case, the average partial reconfiguration throughput can be determined by dividing the bitstream size by the timer value to get an average time to process one word of bitstream data. From the

Table 4.1: Bitstream Size and Reconfiguration time for Square Region

Description	Region Size (WxL)	Bitstream Size (Words)	Time (Clock Ticks)
Small	$3x3$	$181,992$	11,127,109
Medium	$5x5$	$295,026$	17,000,987
Large	$7x7$	$397,276$	22,895,702

Table 4.2: Bitstream Size and Reconfiguration time for Rectangular Region

Description	Region Size (WxL)	Bitstream Size (Words)	Time (Clock Ticks)
Small	$3x5$	$191,884$	11,117,708
Medium	$5x7$	$295,026$	17,002,185

data in the table, it can be determined that the time to process each 16-bit word is on average 58 clock cycles. For a system with a 50 MHz clock, this equates to approximately 1.16 μs per word.

4.4.2 Bitstream Size

Altera provides an estimate of partial bitstream size using (4.1) [130]. In this equation, $HSize_{Region}$ is the horizontal size of the reconfigurable region, while $HSize_{Device}$ is the total horizontal size of the device. $Size_{FB}$ is the size of the bitstream for the full device. For some cases, programming the reconfigurable region requires a mask and set operation, which takes two programming operations. Multiplying the partial bitstream size by 2 gives the worst-case estimate where programming requires two passes.

$$Size_{PB} \approx 2 \left(\frac{HSize_{Region}}{HSize_{Device}} \right) Size_{FB} \tag{4.1}$$

As can be seen from the equation, the estimate determines the proportion of the reconfigurable region's width to the total device width. This indicates that the reconfigurable region's height does not affect the size of the partial bitstream. To verify this, experiments were performed by compiling partial bitstreams for various reconfigurable region sizes, and recording the resultant partial bitstream size. The Altera floorplanning tool accepts region size as an integer (X,Y) origin and a size as integer height and width. This corresponds to an initial row, column starting location and the number of rows and columns to include in the region.

Figure 4.3 shows the size of the partial bitstream vs region width for a fixed-origin, fixed-height reconfigurable region. Figure 4.4 shows the equivalent data based on varying the region's height for a fixed-origin, fixed-width region. The data presented in Figure 4.5 provides a more thorough examination of the bitstream size based both on the size of the region as well as the origin of the region. The first point on each curve represents the bitstream size of a region of width 5 located at the indicated horizontal origin for that curve. Successive points on the curve indicate the results of increasing the region width by 1. In this graph, the discontinuities indicate the documented location of the different heterogeneous resource columns (block memory). These locations were verified by examining the device architecture using the Quartus II compiler's floorplanning tool.

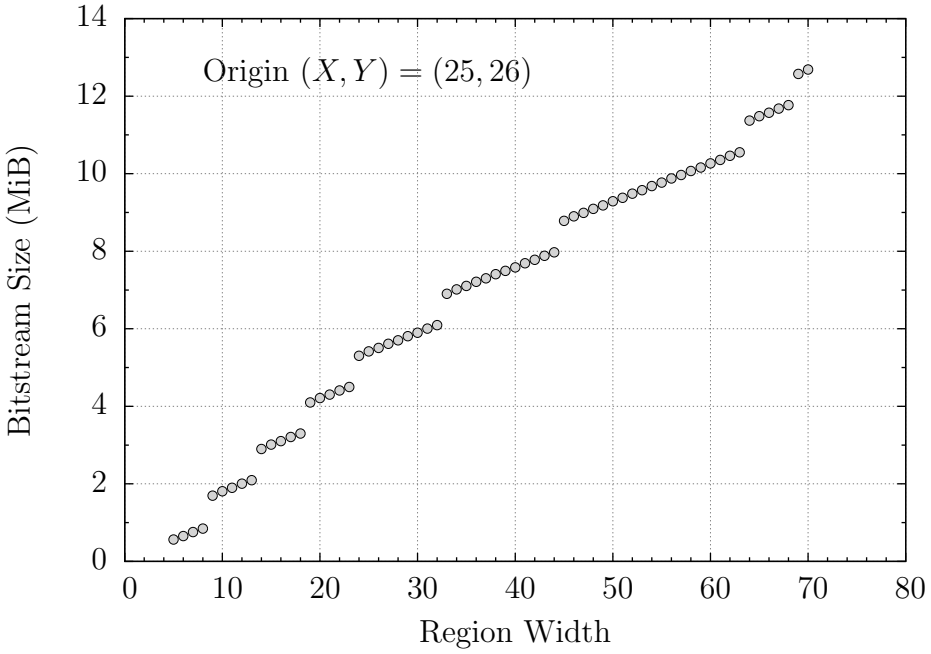


Figure 4.3: Bitstream Size vs Region Width

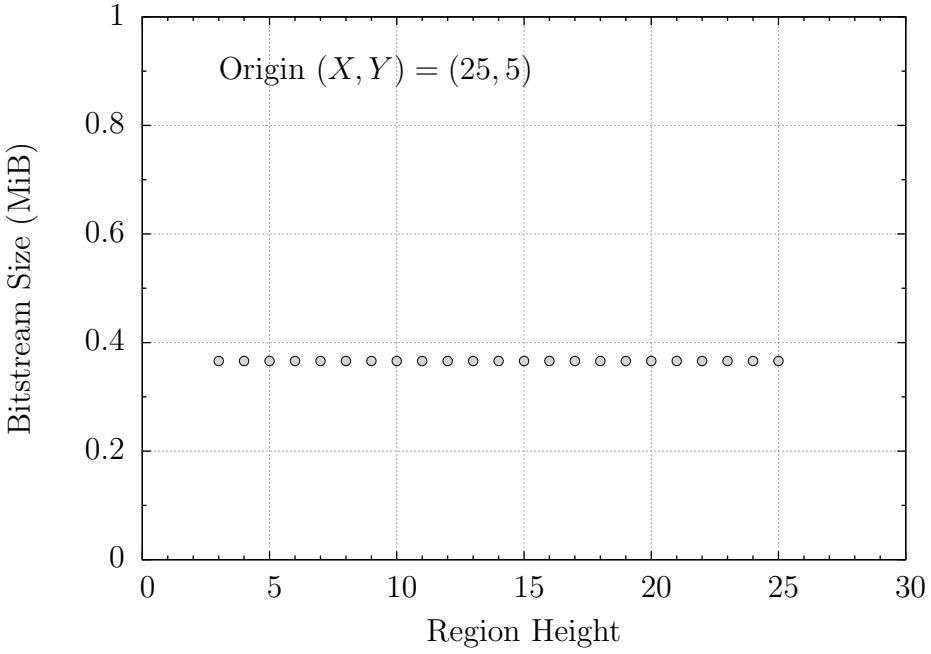


Figure 4.4: Bitstream Size vs Region Height

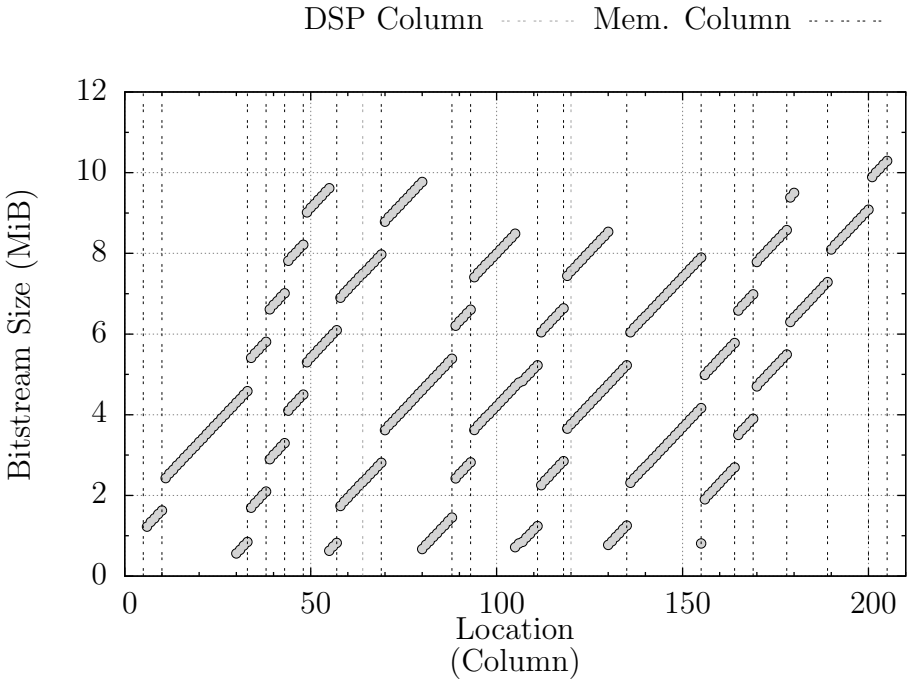


Figure 4.5: Partial Bitstream Size vs Location

4.4.3 Model Conclusions

Based on the results from the above experiments, the following conclusions can be made for the targeted Altera FPGA:

The size of a partial reconfiguration bitstream is indeed dependent on the width of the region it targets. As can be seen by comparing the results in Table 4.1, Table 4.2, and examining Figure 4.4, the size of the bitstream does not depend on the height of the partial reconfiguration region at all. This is in keeping with the model for bitstream size provided by Altera. We also note that partial bitstream size increases regularly by adding columns to the region. The large discontinuities in the graph correspond to the documented location of block memory resources in the Stratix V device, and result from adding the block memory column to the region.

In general, the model for partial bitstream size is a function of the both the width and type of resources contained in the reconfigurable region. An interesting result is that additional logic resources can be had for free by increasing the height of a fixed width region. Regarding resources, the device's overall resources are dominated by logic cells. Examining Figure 4.3 and Figure 4.5 reveals that the bitstream size increases nearly linearly with width. To simplify the partial bitstream size model, it is sufficient to use the dominant resource. This simplifies the model greatly.

In summary, the following conclusions have been developed from examining the example system. The bitstream size grows proportionally to width, not height. Therefore, a partial reconfiguration region should maximize the height of the reconfigurable PE. This will likely result in more resources available in the PE than are needed for most tasks in the application. Finally, since the reconfiguration time is dominated by the time to transfer data to the reconfiguration controller, the reconfiguration time will be proportional to the area of the reconfigurable region.

Although these findings are particular to one type of FPGA from a single vendor, it is reasonable to conclude that adding additional resources to the reconfigurable region results in larger partial bitstreams. Larger bitstreams require longer to read from memory and transfer to the reconfiguration controller. These findings support the assumptions presented in Chapter 3 that reconfiguration time is a function of the region size.

CHAPTER 5

TASK AGGLOMERATION IN RECONFIGURABLE SYSTEMS

Grain Packing techniques have been proposed in both traditional and reconfigurable computers as a method to improve performance for many purposes. The grain packing approach has also been widely applied in multiprocessor software systems for many years. Applications of grain packing have been found for the design of both hardware and software components, especially in a co-design approach. Based on previous results, the use of grain packing techniques in reconfigurable systems looks promising to achieve performance enhancements in dynamically reconfigurable applications.

5.1 Background

It has long been realized that increasing the granularity of an application can improve its performance [131]. For a fixed system and application, increasing the granularity improves locality of data, which reduces the latency due to long communication paths. Communication is latency intense since a processor must generally access an external resource such as a communications bus or shared memory. In software, tasks may be combined by combining two functions into a larger function,

or by assigning tasks to execute on the same execution unit in a multiprocessing architecture.

In the traditional approach, grain packing is implemented by combining tasks in a DAG that are related via data flow dependencies so that data is not required to be transferred amongst (possibly non-local) processing tasks. Finding optimal groupings is known to be an NP-Hard problem [132]. As with most problems of this class, heuristic algorithms are generally proposed to find near-optimal solutions. While heuristic algorithms do not find optimal solutions, their computational complexity is significantly better than NP.

For reconfigurable systems, grain packing was first motivated by the desire to partition the hardware tasks into as few FPGA images as possible via the process of temporal partitioning [133]. Since early FPGAs were not partially reconfigurable, it was important to have as many hardware tasks resident in the hardware as possible, in order to avoid reconfiguration. The process of combining hardware tasks consumes reconfigurable resources in the device, so the grain packing method is subject to the resource availability of the target PE. For tasks with low resource requirements, there is the potential to pack multiple tasks together; for large tasks, few, if any tasks may be combined. There is the potential for significant improvements in run-time by applying such a technique to a dynamically partially reconfigurable device.

5.2 Agglomeration Model

In the context of a reconfigurable system, the grain packing process should produce schedules that are significantly better (smaller makespan) than schedules of

unmodified task systems. Runtime synthesis of FPGA images is a time-consuming task, therefore the framework being proposed is designed to be applied at compile time. This imposes a level of computational overhead, but it is only incurred once for each task system, at compile time only, which is not generally time critical.

The following observations and assumptions are used to develop a model that will be used for the basis of the agglomerative grain packing implementation that will be introduced in the following chapter.

1. The size and number of PEs and partitions is set before run-time.

   This is a practical restriction based on the capabilities of current reconfigurable systems. In order to implement partial reconfiguration on current architectures, the synthesis tools must know the size and location of the reconfigurable regions inside the device.

2. The application's resource requirements are such that the entire application cannot be realized in the device's available reconfigurable resources at one time.

   If the device is large enough to contain the entire application, there is not a strong reason to apply a dynamic reconfiguration approach. Due to the significant penalty associated with dynamic reconfiguration, if the platform is large enough, the entire application should be implemented at once in order to limit the number of partial reconfigurations.

3. The time to reconfigure a partition is a function of the size of the partition.

This assumption arises from two factors. The time necessary to reconfigure a partition is a function of the number of logic cells that must be reconfigured [134], and the assumption that all logic cells in a partition should be configured even if not used. While it may be possible to leave unused logic cells uninitialized, this could result in unanticipated operation. In order to prevent such a case, the configuration of all unused logic cells to known default values is assumed.

4. Tasks with the same type id differ only in the data processed.

   This follows from the definition of task type introduced earlier. This assumption asserts that task types represent a specific sequence of operations and that identical types represent the same sequence of operations.

5. Subsequent executions of tasks with the same type id within the same PE do not require reconfiguration.

   This is an extension of the previous assumption that also follows from the definition of task type. Implicit in the definition of task type is the notion that in a given task graph, the same set of basic operations are often invoked repetitively.

6. Provided the partition contains sufficient resources, more than one task can be present within a partition.

This observation follows from item 3. If the entire partition must be reconfigured, and there is additional unused area, the unused area may be configured using any static configuration.

7. Each PE partition can have exactly one active task at any given time.

   By restricting the partition to only one active task, the chances of incorrect operation are reduced. Although it is technically feasible to separate the logical implementation of multiple tasks, since the multiple tasks reside in the same partition, they would likely be required to share the partition's inputs and outputs. Future extensions could reevaluate this restriction since the PE should be capable of executing multiple tasks with the potential for additional speedup. However, this will require careful consideration of how the combined tasks would share the limited I/O of the PE.

8. At compile time, grain packing can occur by clustering tasks into groups such that each resultant grouping will fit into at least 1 PE partition.

   The resources needed by a task, as well as the size and location of the reconfigurable regions are all known at compile time. This gives sufficient information to determine if a grouping can be established. If no such grouping can be found, then execution of the task graph can continue without combining tasks.

9. Subsequent execution of different tasks within the same cluster does not require a reconfiguration.

This arises as an extension of the task definition to the multiple task partition system. The individual tasks do not lose identity by being combined with another task.

10. Resources within a cluster are only consumed once per instance.

    Each task must be self-contained. Computational resources are not allowed to be shared amongst tasks. This restriction eliminates the possibility of contention between the constituent tasks in a cluster. Although this results in maximum resource usage, it further reduces the chance of incorrect execution due to dependencies between the various tasks.

Based on these observations and assumptions, the application of the grain packing technique to the dynamically reconfigurable platform can be established. The approach is to first identify tasks that do not fully occupy their assigned PE partition. Next, agglomerate tasks into clusters of 2 or more tasks whose combined resource usage is less than the available resources. Finally, combine the tasks in the clusters into a complex clustered task that performs the functions of the individual constituent tasks. This method should improve the runtime of the application by eliminating unnecessary reconfigurations. Clustering the tasks is equivalent to having at least two task types configured simultaneously. Since a configured task does not require reconfiguration, the application may transition between two types in a cluster without a reconfiguration between them.

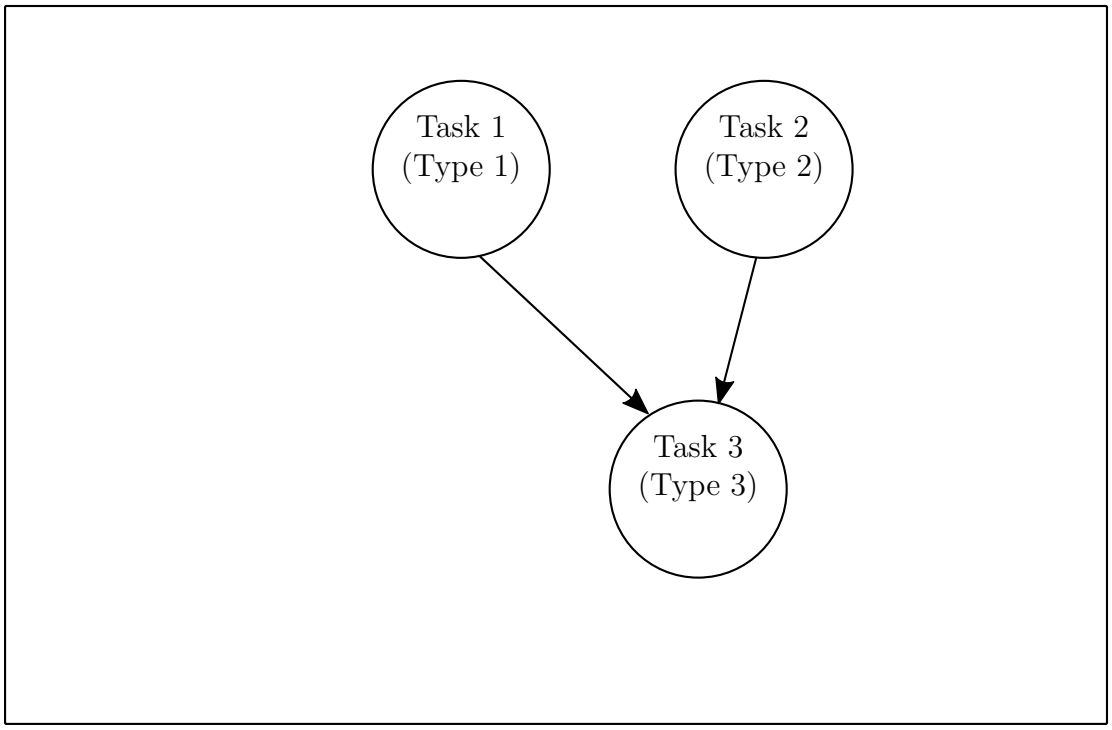
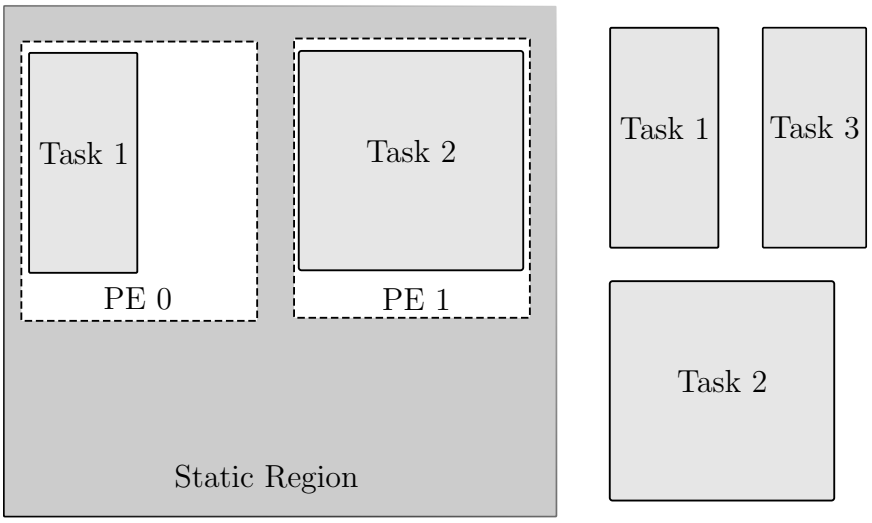


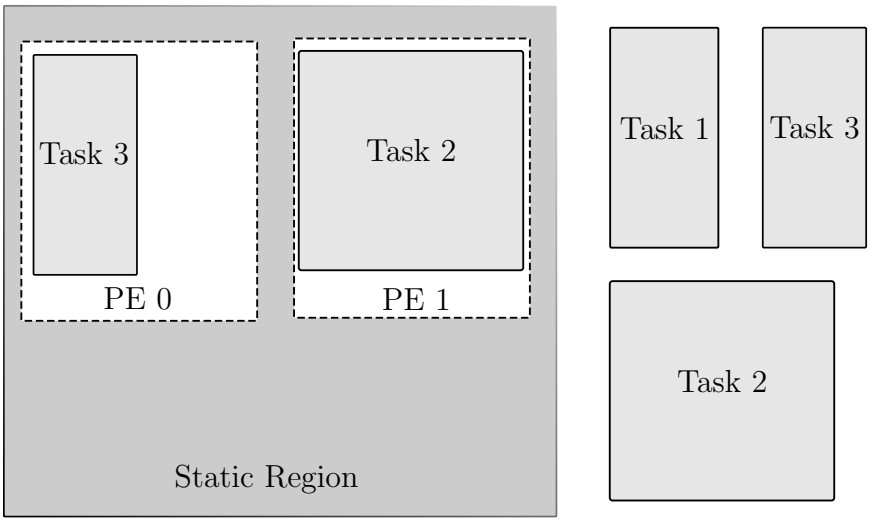
Figure 5.1: Application Task Graph for the Example

5.3 Task Agglomeration Example

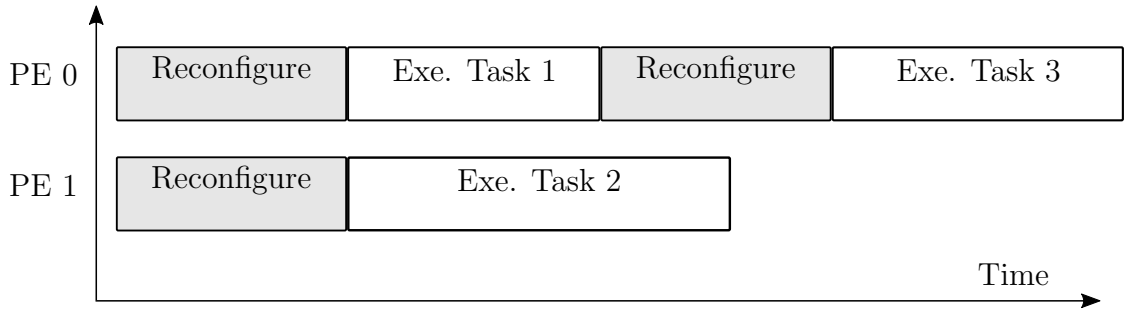
To demonstrate the advantages of the proposed concept, consider the application given by the graph depicted in Figure 5.1 and the reconfigurable platform shown in Figure 5.2a. The task system consists of 3 tasks executing on a single-chip reconfigurable platform consisting of 2 identical hardware PEs. Each task is represented in the figure by a block with width proportional to its resource usage. Each task is also a distinct type, which is indicated by the varying width of the tasks. As can be seen from Figure 5.1, Task 1 and Task 2 are independent of each other, and Task 3 is data dependent on the output of Task 2. For the sake of simplicity, this example does not contain any control dependencies.



(a) Example Platform Initial State



(b) Example Platform after Reconfiguration

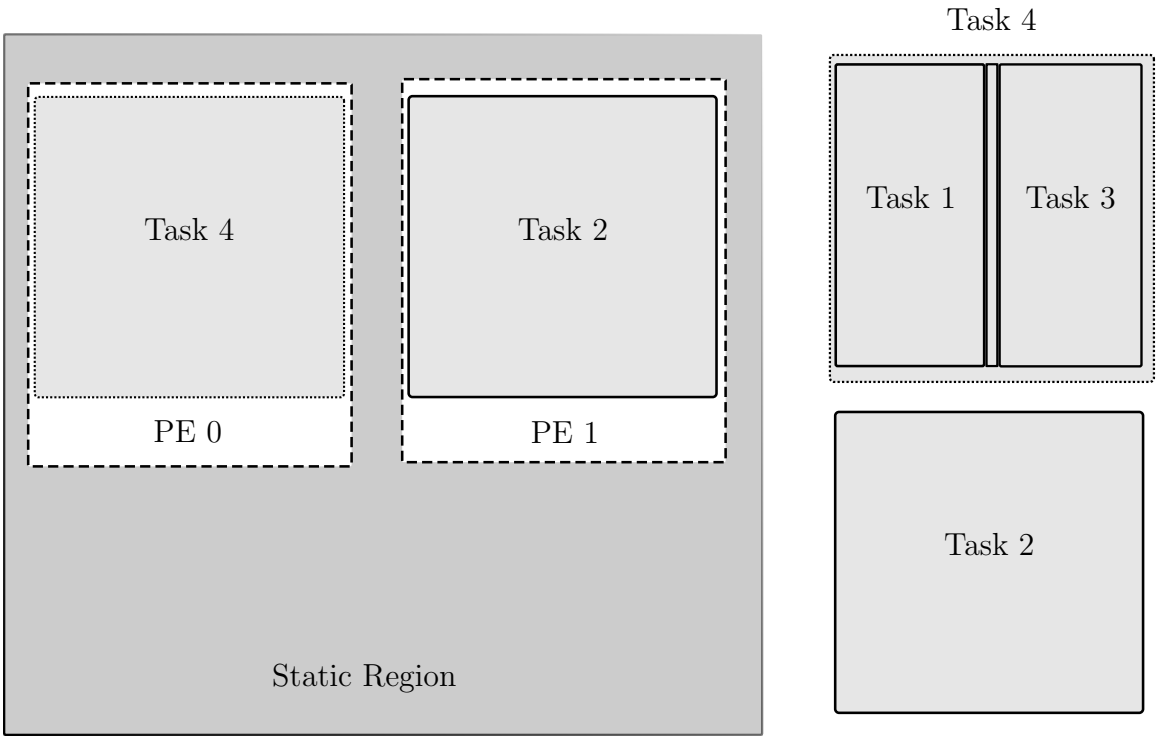


(c) Example Schedule

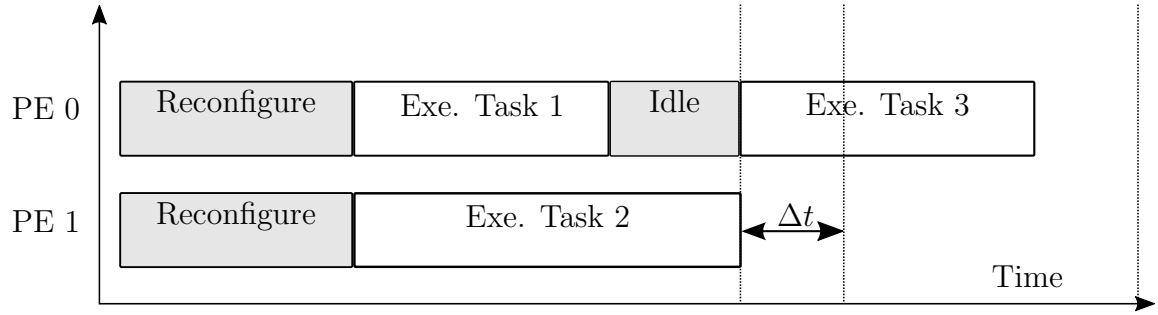
Figure 5.2: Example Before Clustering

Figure 5.2a represents an initial allocation of tasks to the two available hardware PEs. Since Task 1 and Task 2 are independent, they may begin execution once the hardware is configured and ready. Figure 5.2c shows a Gantt schedule depicting how this application would execute. Both PEs begin by reconfiguring for their first task. Note that the finite number of reconfiguration controllers does not affect the initial configuration because it is assumed that the initial configuration is loaded simultaneously when the device is initialized. Once reconfigured, the tasks begin execution. Since Task 1 completes execution before Task 2, PE 0 can begin reconfiguring to execute Task 3. Task 3 can begin execution as soon as both the reconfiguration is complete and Task 2 completes execution. Since Task 2 completes before the reconfiguration is done for Task 3, Task 3 may begin execution as soon as reconfiguration is complete. In this example, it is noteworthy that although Task 1 occupies only a small portion of PE 0's resources, these PE resources are not available for executing other tasks, as shown in Figure 5.2a. This mapping represents poor spatial efficiency because PE 0 has many unused resources when executing Task 1.

Alternatively, there is the approach presented in Figure 5.3a. In this scenario, Task 1 and Task 3 have been combined into a single task cluster, which now represents a new fourth task type. Figure 5.3b depicts the effect of clustering on the schedule of this example task system. As before, Task 1 and Task 2 may begin execution immediately following the completion of the initial reconfiguration, but because Tasks 1 and 3 have been combined into a new task there is no need to reconfigure PE 0 after Task 1 completes execution. However, since Task 3 is dependent on the completion of Task 2, it may not begin to execute until Task 2 has completed. Therefore, an



(a) Example Platform



(b) Example Schedule

Figure 5.3: Example Clustered Application

idle period has been introduced before the start of Task 3 to delay its execution until Task 2 has completed.

As can be seen from this example that although the idle time has been introduced, it is for a shorter period than the reconfiguration delay that would have

normally occurred. Further, this idle period can be of varying length while the reconfiguration time is a constant duration since a fixed size partition is always being reconfigured. The vertical dotted lines in Figure 5.3b show the amount of time that has been saved in this example. The rightmost dotted line represents the original runtime, while the pair of dotted lines near the middle of the graph show the difference between the reconfiguration time and the introduced idle time, Δt. For the presented example, the time saved is equal to the difference between the reconfiguration time and the idle time needed for Task 2 to complete execution.

It is also worth noting that the amount of bitstream storage has been reduced. Originally, this example system required 3 bitstreams of equal size. By combining Task 1 with Task 3, the need for the individual source bitstreams can be eliminated. The bitstream containing the combination of Task 1 and Task 3 is now sufficient to implement either of the tasks. Grain packing this simple example application has resulted in a memory savings of 33%.

5.4 Implementation

There are potentially several possible ways to implement the grain packing technique for hardware tasks on a reconfigurable system. The most straightforward method is outlined below. This method assumes that the hardware tasks can be implemented as a single block with defined inputs and outputs. Since all tasks must execute in identical PE partitions, this method further assumes that all hardware tasks must conform to a common I/O interface.

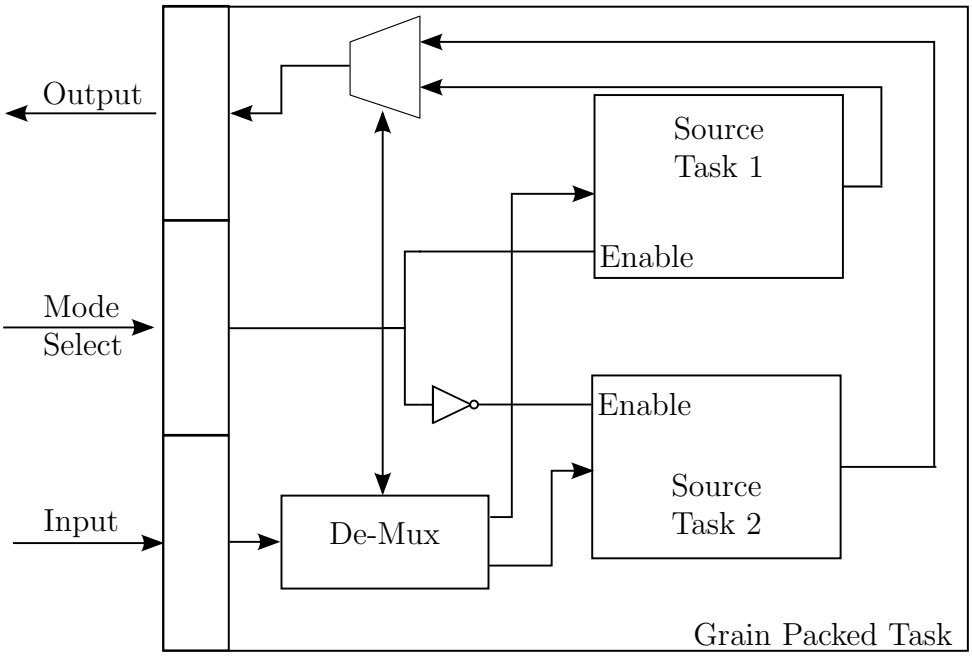


Figure 5.4: Packed Task Implementation with Registered I/O

The proposed grain-packing method is to implement each hardware task as a separate subsystem within a higher level wrapper block. Then, a combination of multiplexer and demultiplexer circuits are used to select the appropriate mode of the packed task. Figure 5.4 shows a simplified example of this approach that implements two source tasks and the associated mux/demux logic. Using this method, the I/O of the tasks are registered at the periphery of the meta-task. The input register feeds the input of a demultiplexer block, which then provides an output to drive the inputs of the individual source tasks. Likewise, the outputs of the source tasks are fed into a multiplexer which then drives the packed task output register.

A mode select register is now provided to select the clustered task mode. For a two task cluster, a single bit is sufficient to select between two tasks. A meta-task containing more than two tasks would also require decode logic to enable the

I/O selection logic appropriately. The mode input drives the select inputs of both the multiplexer and demultiplexer. To enforce the requirement that only one task is active at a time, an enable signal is routed to the individual tasks. Suitable decode logic ensures that only one enable signal is active at once. This implementation can easily be scaled by appropriately modifying the multiplexer/demultiplexer and the width of the mode register.

There are several advantages to this approach. Chiefly, the simplicity of the approach lends itself to an automated implementation. Also, there is no need to modify the source tasks in this approach, which is useful when using third party IP, or task implementations that have previously been optimized. This approach lends itself to straightforward implementation in a register transfer language (RTL), such as VHDL or Verilog. The source tasks may be implemented as components inside an RTL component for the task wrapper which also includes the mux/demux and decode logic. Such an approach would add little overhead to the design.

A disadvantage of this approach is that routing for the I/Os will quickly come to dominate for a large number of source tasks. However, it remains to be seen how this will impact a practical system. If a large number of tasks can fit into a single partition, it is likely that the application graph is not well matched for the target architecture. In such case, it is probably advisable to re-evaluate the application graph. Another drawback is that this implementation is not suitable for tasks that cannot be adapted to a RTL structure. Such tasks will likely require a custom solution based on their implementation.

Another drawback of this approach is the need for additional control bits to determine which mode of operation to select. Implementing this multiplexed input and output scheme will require $\lceil log_2(n) \rceil$ bits, where n is the number of tasks in the cluster. Since the PE's I/O is fixed at design time, this could impact the I/O available for the task. For new tasks targeting a previously designed platform, the control bits will have to be borrowed from the PE's existing I/O, which could be a significant drawback in existing platforms. If the platform is newly developed, these bits can be implemented transparently to the task implementation, greatly reducing the impact of grain packing in these instances.

CHAPTER 6

DYNAMIC EXECUTION SIMULATION FRAMEWORK DESCRIPTION

In order to determine the effectiveness of the two proposed clustering techniques, a simulation framework was developed. This framework applies a designated clustering algorithm and simulates the execution of the resultant modified application using a previously developed parallel execution simulator. The simulation is designed to be consistent with the system and architectural model described previously. Results from the execution simulation can then be used to make quantitative observations of the performance of the clustering methodology.

6.1 Simulation Framework

The simulation framework uses a modular architecture to implement the separate functions of clustering, scheduling under deterministic conditions, and scheduling nondeterministic execution. Two separate software programs are used cooperatively to provide these capabilities. The first program is a stand-alone cluster generator, which clusters a graph using one of the algorithms described in Chapter 7. The cluster generator uses a sophisticated static scheduler to determine an initial sched-

ule without knowledge of the dynamic execution properties of the application. The same static scheduler is also used to calculate the value of the objective functions used by the heuristic clustering algorithm. Once the clusters have been determined, a separate dynamic system execution simulation is used to determine how effective the clustered graphs under dynamic (nondeterministic) execution conditions. This dynamic simulation was adapted from previous research [125], [126] to recognize the output of the cluster generator before performing dynamic execution simulation.

Both the cluster generator and dynamic simulation read a DAG description of the application, as well as a set of parameters that control the behavior of the simulation. The DAG descriptions are generated offline by a third program and read by the simulation programs during execution. The simulation parameter file is read when the simulations are initialized. This file is used to control variable aspects of the simulation, such as the number of PEs, degree of non-determinism, region size and reconfiguration time. Changes can be made to the platform model by modifying the simulation parameters without the need to recompile the simulation. To ensure that both programs execute the same platform model, they have been designed to use the same parameter file. Figure 6.1 illustrates the simulation framework architecture.

6.2 Static Scheduler

A computationally complex heuristic-based static scheduler is used for several purposes in the simulation framework. Since this simulation is critical to the execution of the simulation, it is necessary to understand its operation before describing the simulation programs. The static scheduler is used to: calculate an initial deterministic

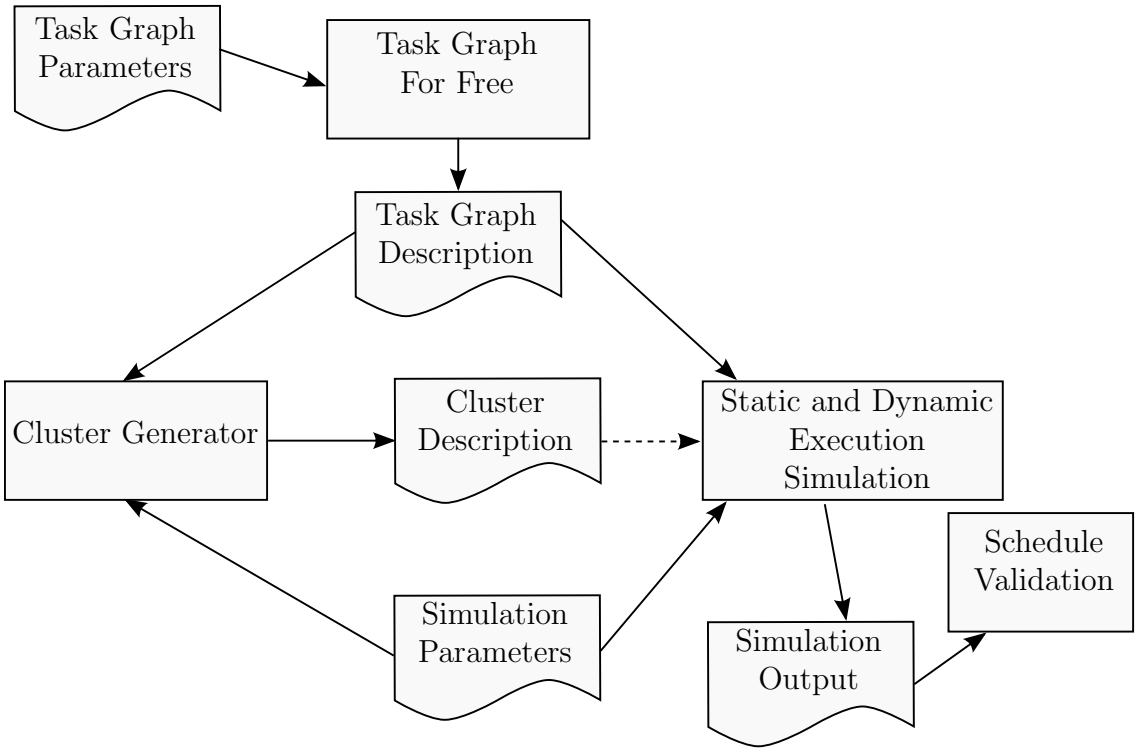


Figure 6.1: Simulation Framework Architecture

schedule, to determine the objective function in the clustering algorithms, and to estimate the best case runtime of a target task graph under dynamic conditions. Previous researchers have used this scheduler to compare with the results of less sophisticated dynamic scheduling methods [25]. Due to the complexity of the static scheduler, it should be capable of determining near optimal schedules in a reasonable time, making it a suitable approximation of an omniscient scheduler.

The core functionality of the static scheduler is made up of two independent components. The first handles task ordering and processor assignment, while the second component produces a detailed schedule based on the results from the first component. The detailed schedule includes the total run-time as well as task execu-

tion, idle, and reconfiguration times for each processor. This architecture allows the core scheduler to determine the performance of a single task ordering and assignment. In order to evaluate multiple candidate schedules, the scheduler searches the solution space using heuristic optimization techniques and returns the best result.

6.2.1 Core Scheduling Algorithm

The static scheduler is based on a list based scheduling approach. An initial priority list is generated using one of several well-known list scheduling methods [94]. This results in a master list structure indexed by task ID. Each task in the list includes a set of task attributes, including a statically assigned execution order (priority). A second list structure is used to sort the task IDs according to this assigned priority. Maintaining a separate prioritized list allows the priority list to easily be sorted without the overhead of copying the other task attributes. Figure 6.2 shows a representation of the two list data structures used by the static scheduler.

Once the initial prioritized task list has been established, the scheduling algorithm searches the prioritized list looking for tasks who's precedence constraints have been met. Precedence constraints are considered met once all of a task's predecessors have have been placed in the schedule for execution. When a task has been selected from the prioritized list, its task ID is used to retrieve the other task parameters from the master list. These parameters are then used to update the schedule information. Algorithm 1 presents an overview of how the list based assignment routine works, while Algorithm 2 describes the task selection methodology.

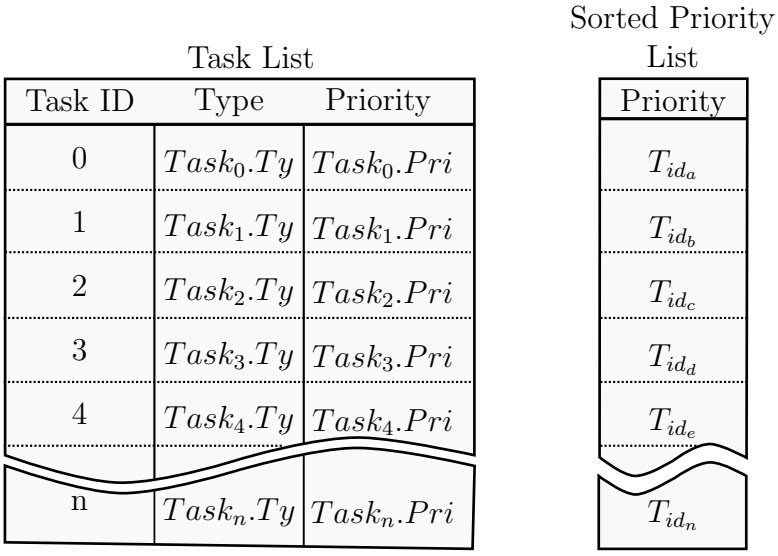


Figure 6.2: Example List Based Data Structures

Algorithm 1 Priority based Task Assignment

```
1: while UnscheduledTasks > 0 do
2:    Task ← DetermineHighestPriorityTask()
3:    PE ← GetShortestSchedulePE(Task)
4:    AssignTaskToPE(Task, PE)
5: end while
```

Algorithm 2 Determine Highest Priority Task

```
1: for i = 0 to NumberOfTasks − 1 do
2:    if Task_{Priority_i}.RemainingPredecessors = falseand Task_i.Unscheduled =
      true then
3:       return Task_i
4:    end if
5: end for
6: return false // Return value if no tasks can be run
```

Algorithm 3 Get Shortest Schedule PE

```
1: ShortestSchedulePE ← −1
2: ShortestSchedule ← Integer_Max
3: for i = 0 to NumberOfPEs − 1 do
4:    if AssignTaskToPE(Task, i) < ShortestSchedule then
5:       ShortestSchedulePE ← i
6:       ShortestSchedule ← AssignTaskToPE(Task, i)
7:    end if
8: end for
9: return ShortestSchedulePE
```

Once the algorithm selects a task to be scheduled, it must be assigned to a processor. Each processor is evaluated to determine how the total application execution time will be affected by adding the task to that processor's schedule. The task is assigned to the processor that corresponds to the least increase in total application runtime. The effects of processor reconfiguration are considered in this calculation; a processor that does not require reconfiguration will generally result in a shorter schedule. Algorithm 3 provides an overview of processor selection.

After processor selection, the task is scheduled for execution on the selected processor. The execution will be scheduled for when the latter of when the processor becomes free, or the task's earliest start time, based upon the task's precedence constraints. If the task type of the new task is different than the type of the processor's previously scheduled task, a reconfiguration is also scheduled before the new task may begin execution. Once the task is scheduled for execution, the precedence constraints are updated for the remaining unscheduled tasks, and the algorithm restarts by looking for the next highest priority unscheduled task that is ready for execution.

The detailed scheduling component of the scheduler is relatively straightforward. Once the ordering and assignment component has completed, it produces a data structure representing the order of tasks for each processor, which is provided to the scheduling component. The scheduling component reads the detailed task information from the attribute list and calculates the expected runtime of the provided schedule. If any task's precedence constraints are not yet met, the scheduler adds idle time to the schedule until the task may be run. Likewise, if the task requires PE reconfiguration, an appropriate amount of reconfiguration time is added to the

schedule for the specified processor. Once the detailed schedule calculations have been performed, idle time is added to the end of each processor's schedule until the final task completes execution.

The computational complexity of the entire static scheduler is $\mathcal{O}(PEn^2)$, where PE is the number of processors, and n is the number of tasks. This complexity includes the processing necessary to maintain the sorted priority list, as well as the effort to search the list and evaluate the runtime on each of the processors.

6.2.2 Heuristic Optimization

Since the core static scheduling algorithm discussed in the previous section produces only one candidate schedule, the static scheduler employs a set of heuristic optimization algorithms to evaluate multiple schedules. These algorithms include Simulated Annealing (SA), Genetic Algorithm (GA), and Particle Swarm Optimization (PSO). Each optimization technique records the shortest schedule that was found using that method. A global optimization is then performed, which returns the shortest schedule found by each of the three techniques.

Heuristic optimization algorithms are commonly used to find reasonable solutions to NP-complete problems [135]. Since an exhaustive search of the solution space would take a prohibitive amount of time, a heuristic optimization algorithm searches only a subset of the total search space. The characteristics of individual algorithms vary, but in general, they attempt to optimize some objective measure of the solution by moving in the search space. Such algorithms examine this space to determine the direction of maximum improvement of the objective function and

generally move toward that improvement. In order to cover the search space and escape local extrema, most algorithms also include a nondeterministic component in their movement as well. An advantage of heuristics is that they run in a bounded amount of time. However, this limits the total number of solutions that they may examine. This limitation means that a heuristic algorithm may find a good solution, but is not guaranteed to find the global best solution.

Each of the three heuristics used by the scheduler seeks to optimize the objective function by modifying the priority lists that are used in the list scheduler described earlier. In the context of the scheduling problem, the optimization function is minimizing the makespan of the generated schedule. Starting from a baseline candidate schedule, optimization is performed by modifying the priorities assigned to individual tasks. Each optimization method performs priority modification based on the specific properties of the particular heuristic. Once the priorities have been adjusted, the list scheduler is invoked again to establish the new value of the objective function. The heuristics are applied iteratively and return the best solution found.

6.2.3 Simulated Annealing

An SA algorithm is designed to mimic the controlled physical process of annealing metal [136]. The general approach relies on an objective function to grade a proposed solution. A parameter, known as the *temperature*, is used to control the operation of the algorithm. The algorithm is performed iteratively, reducing the temperature geometrically each iteration until it reaches a pre-assigned *freezing temperature*, at which point the algorithm is terminated. In order to increase the number

of evaluated solutions, multiple iterations are often performed at a given temperature (isothermally). At each iteration, the proposed solution is updated, and the objective function is reevaluated. If the objective function has improved from the previous iteration, the current solution is retained for the next iteration. If the objective function has worsened from the previous iteration, the solution is generally discarded. However, a worsening solution may be preserved probabilistically so that the algorithm can escape a local optimum. Probabilistic acceptance is based on the current value of the temperature; the freezing point is chosen such that as the temperature is reduced, the likelihood of probabilistic acceptance decreases to zero.

Applying the SA algorithm to the scheduling problem, an initial priority list is generated and evaluated using the static scheduler. During each iteration, the static scheduler is invoked to generate the next value of the objective function. A movement routine is invoked at each iteration, which generates a new priority list. Algorithm 4 provides an overview of the SA approach used in the static scheduler.

Algorithm 4 Simulated Annealing Optimization

```
1: InitializePriorityList()
2: Objective_Best ← StaticSchedule(DAG, Platform)
3: InitializeTemperature()
4: while T > T_Freezing do
5:    for i = 0 to Iter_Max do
6:       Objective_Iter. ← StaticSchedule()
7:       if (Objective_Iter. < Objective_Best)or(AcceptProbabilistic(Objective_Iter.))
         then
8:          Objective_Best ← Objective_Iter.
9:       end if
10:      UpdatePriorityList()
11:   end for
12:   UpdateTemperature()
13: end while
```

Table 6.1: Static Scheduler SA Optimization Parameters

Parameter	Value
$T_{initial}$	100.0
T_{final}	0.36786
λ	0.9999
i	50

The scheduler's SA algorithm relies on two movement strategies. The first strategy is a pairwise swap. In this method, two tasks in the priority list are selected at random and their task IDs are swapped. The second movement strategy is a random modification of the priority of a randomly selected task. After modifying the priority list, the static scheduler then determines a new objective function value from the revised priority list. Based on the value of the objective function, the assigned task priorities are rejected, accepted, or accepted probabilistically according to a geometric cooling schedule. Generally, the cooling schedule is given by $T' = \lambda T$, where T' is the temperature for the next iteration, T is the current Temperature, and λ is the cooling constant. The computational complexity of the SA algorithm is given by (6.1). Where T_{final} is the stopping temperature, $T_{initial}$ is the starting temperature, i is the number of isothermal iterations, and n^2PE is the complexity of the core scheduling algorithm. For this SA implementation, the parameters are given in Table 6.1

$$\mathcal{O}\left(\frac{ln\left(\frac{T_{final}}{T_{initial}}\right)}{ln(\lambda)}n^2PEi\right) \tag{6.1}$$

6.2.4 Genetic Algorithm

The second heuristic employed to optimize the schedule was a classical Genetic Algorithm. Genetic Algorithms are a class of heuristics that are designed to replicate the biological process of natural selection [137]. GAs operate on a population of candidate solutions, which are updated over several generations (iterations). In each iteration, a child population is derived from the current parent generation. The binary operation of *crossover* is used to create children from combinations of the parents. After crossover, individual children may be altered at random by a *mutation* process, which varies the solution, preventing the algorithm from continually revisiting the same solutions.

Each member of the population is referred to as a *chromosome*. Each chromosome encodes a possible solution as a string. In the case of the static scheduler, a possible task priority list is used to represent a chromosome. After a population of chromosomes have been initialized, the algorithm pairs the members of the population for crossover. Crossover is a binary operation that combines one part of the first parent solution with the remaining part of the other parent solution. The single-point crossover scheme used in the static scheduler takes the two chromosome strings and divides them at the same point in each chromosome. The upper half of one parent chromosome is then concatenated with the lower half of the other, to produce each child chromosome. This genetic algorithm uses a steady-state methodology, where all the individuals are replaced each generation.

Table 6.2: Static Scheduler GA Optimization Parameters

Parameter	Value
G	1,000
P	100
$P[Mutation]$	0.05
$P[Crossover]$	0.90

The GA used in the scheduler first generates a population of candidate task priority lists to initialize the algorithm. Next, objective function values are calculated for each individual priority list in the population. The algorithm selects two candidate priority lists at random, and then performs binary crossover. In this technique, a crossover point is determined randomly; the two lists are divided at this point, and two child lists are generated from combining the upper portion of one list with the lower portion of the other, and vice versa. After crossover, the mutation function is applied to the child task lists. The random mutation is similar to the random movement operation used by the SA method; a task from one of the child lists is selected at random, and its priority is assigned a random value.

For the GA, the computational complexity is given by (6.2), where G is the number of generations, P is the number of individuals in the population, and n^2PE is again the complexity of the core scheduler. The GA used for static schedule optimization uses the parameter values shown in Table 6.2.

$$\mathcal{O}\left(GnPn^2PE\right) \tag{6.2}$$

6.2.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a relatively new optimization heuristic [138]. In the PSO algorithm, individuals in a PSO population are imagined to be particles moving freely in an n-dimensional space. Individuals are grouped into sets of neighbor particles. Between iterations, each individual particle moves according to a nondeterministic velocity function. After completing an iteration, new objective functions are calculated for each particle's new position. Each group's best objective function and the global best objective function are determined. Every individual's velocity is then updated. The updated velocity is calculated by the vector addition of the particle's weighted previous velocity (its inertia), the weighted direction vector to the group best, and the weighted direction vector to the global best. Weights are implementation specific but are assigned such that the particle tends to generally follow it's initial path, with a slight pull toward the group and global best solutions. Over time, the particle's inertia weight is decreased linearly until it eventually reaches 0.

Like the GA, the PSO scheduling optimization algorithm relies on a population of candidate task lists. Each n-element priority list is viewed as a coordinate in an n-dimensional space. A particle's initial position in n-space is given by interpreting its corresponding priority list as a coordinate, and each particle's initial velocity is determined randomly. The particles are divided into single-member groups. The static scheduler is again employed to establish the objective function for each particle based on its priority list. During each iteration, objective functions are calculated

Table 6.3: Static Scheduler PSO Optimization Parameters

Parameter	Value
P	100
i	1,000
Initial Inertia Weight	0.40
Final Inertia Weight	0.00
Group Weight	2.0
Global Weight	2.0

for each particle, and the global shortest schedule is found along with the shortest schedule for each group/individual. The velocity functions are updated, and the new particle positions are calculated.

Complexity of the PSO algorithm is given by (6.3). The parameters of this equation are identical to the complexity of the GA, with the exception that i represents the total number of iterations for which the PSO runs. Table 6.3 lists the values of the PSO optimization parameters.

$$\mathcal{O}\left(niPn^2PE\right) \tag{6.3}$$

6.3 Cluster Generator

The cluster generator is a separate application that is used to generate the clustered meta-tasks from the initial task graph description. The application begins by reading the task graph file and the simulation parameters. The static scheduler is then used to determine the initial task ordering and create an assignment that

is based on the PE information contained in the simulation parameters. The input parameters also determine which of the two clustering algorithms should be used.

Both clustering methods are based upon the combination of tasks into the same task type only if they fit within the same cluster. To do this, the clustering routine first selects tasks and adjusts their type to a new value that differs from any other assigned task type. Next, all graph tasks are searched to find any tasks that correspond to either of the base types of the clustered tasks. These are then updated to also correspond to the newly assigned cluster type. For example, if base types 1 and 2 are selected for clustering, all other instances of base type 1 and 2 in the DAG are also converted to the new type. This ensures that if a task with a different task ID but the same base type occurs either immediately before or after these tasks, the PE will not have to undergo a reconfiguration if transitioning from the clustered task to one of its constituent tasks or vice versa.

After executing the desired clustering method, the application outputs a cluster information file, which associates the task's initially assigned task ID with the new task type that was generated by the clustering algorithm. This cluster information file is subsequently used by later portions of the simulation framework. The results of the clustering operation can be recreated by first reading in the original task graph information, and then overwriting the original type information from the cluster file. Using this approach allows later simulations to apply or remove clustering by reading the initial graph file or the cluster file as appropriate.

It is assumed that that the combination of tasks using the grain packing approach will also include resource usage overhead beyond the simple linear combination

of individual task resources. Primarily, these overheads arise from two areas, the internal complexity of the multiplexing logic described in the previous chapter, and overhead introduced by the compiler when synthesizing a combined task. The cluster generator component does not take these overheads into consideration because they are difficult to quantize and the simple model appears to be a good predictor of resource usage when combining modules on FPGAs.

6.4 Dynamic Execution Simulation

The dynamic system simulator has been used in previous research to model the execution of reconfigurable systems under nondeterministic conditions [25], [126]. This simulator models dynamic behavior using the discrete event paradigm. The model is designed to replicate the execution of a micro-architecture for dynamic task graphs using the bounded nondeterministic model described in Chapter 3. Simulation parameters for the dynamic simulation are derived from the same initial task graph and parameter files used by the cluster generator. After reading the initial task graph file information, the dynamic simulation determines from the simulation parameters if it should simulate the graph as is, or use the cluster information from the cluster generator. If the simulation schedules a clustered graph, the simulation reads the cluster information file generated by the cluster generator and overwrites the type information read from the initial graph. The ability to retype tasks is the only modification that was made to the dynamic simulator that affected its base operation. The simulator and the actual (dynamic scheduler) task's identity was not lost.

In order to validate the correct operation of the dynamic scheduler, an additional logging facility was added so that the simulation would output detailed schedule information once the simulation was complete. This facility provides a trace of the internal state machine used by the dynamic simulator which can be used to reconstruct the schedule based on the execution states for each task in the application.

Once the parameters for the simulation have been established and the task information has been parsed, the simulation assigns each task a static priority. One of two methods is used to assign the static priority. The first is based on the amount of slack the task has, tasks with less slack are assigned a higher priority. The second method assigns a priority based on the number of successors of each task, tasks with more successors are given a higher priority. This static priority is used in part by the scheduler architecture to select the best task to execute at run-time.

Two data structures are used to drive the simulation; Events and Tasks. Both data structures are stored in a linked list format used to implement a queue. The simulation is a loop that pops the first event off of the event queue, does the appropriate processing, deletes the current event, and pops the next event. Additional events are pushed on the queue by the event processing as necessary. The task queue represents the tasks that have met the conditions to be ready to execute. Tasks are pushed onto the task queue when an event causes one of the simulated PEs to update state.

Event processing is controlled by a state machine which is invoked each time a new event is removed from the event queue. Simulation events are encoded using a data structure to capture event type, time, and affected PE. The event queue is a linked list of these data structures sorted according to the execution time of

the event. States are determined by the event type and are based on the allowed PE states: Reconfig, TaskSwitch, CPU_TaskSwitch, Running, & CPU_Running. An additional state, ChkLogic, is provided to simulate the behavior of the dynamic execution manager's logic.

The simulation keeps track of tasks that are ready to execute using another linked list containing tasks that have met execution conditions. During each of the states TaskSwitch, CPU_TaskSwitch, Running & CPU_Running, the simulation checks the dependencies of the remaining tasks in the graph, assigns a dynamic priority and adds them to the task queue if they can be executed.

Simulation begins by finding all tasks with no predecessors and enqueueing a simulation event of type Reconfigure for each type for as many predecessorless tasks as can be assigned to available PEs. If there are more tasks than available PEs, the tasks are placed on the task queue, and will be assigned to free PEs according to the execution rules of the reconfigurable architecture. The simulation proceeds by entering a loop that continues to dequeue the first event from the event queue until the event queue is empty. Since the reconfiguration events have been enqueued, these are the first events processed. After processing each event, a ChkLogic event is enqueued for immediate execution. The simulation will process through the event states for each task on the queue. Executing one of the TaskSwitch, CPU_TaskSwitch, Running, or CPU_Running states will update precedence constraints, and add tasks to the task queue as they become ready for execution. The ChkLogic state will create simulation events as appropriate based on the new tasks that have been added to the task queue.

The ChkLogic Event removes tasks from the end of the task queue and creates new simulation events based on the task removed. It then places these new events in the event queue according to the event's scheduled time. ChkLogic determines this by examining tasks in the task queue. A parameter known as SearchDepth controls how deep into the Task Queue ChkLogic is allowed to look when it is scheduling events. Allowing ChkLogic to look ahead in the Task Queue allows ChkLogic to make some optimizations, such as configuration locking, based on the needs of upcoming tasks. Previous research [125] showed that little improvement in the simulated runtime was achieved for values of SearchDepth greater than 7. Therefore, this value was used for all values of the dynamic simulation.

Another feature of the simulation is the ability to control optimization of the initial reconfiguration. Since the initial task to PE assignment can be determined before execution by examining precedence constraints, the initial reconfiguration of these tasks can be skipped. Rather than initializing the device with empty PEs, the device can be initially programmed with each PE configured its initially assigned task. This can be accounted for one of two ways. The first is to set the initial reconfiguration time to 0 to indicate that the system has been configured previously, and has been waiting to begin execution. In the second approach, each PE is assumed to be programmed in parallel with identical reconfiguration times. This method assumes that the entire reconfigurable device was programmed just prior to executing the application. However, this time will be longer than the time to reconfigure a single PE. In this case, each PE will see an initial reconfiguration time that is identical to the reconfiguration time of the entire device. The dynamic simulation implements

the second approach, but conversion between the two methods is allowed by adding or subtracting the entire device configuration time from the application's runtime.

The dynamic simulation is also capable of simulating multiple execution frames. In this case, the graph represents a single frame of execution. First, the entire graph is executed in its entirety. Once the final task of the first frame has completed, the subsequent frame is simulated by adding the initial tasks in the graph to the task queue. The remainder of the graph is then scheduled according to the rules of the dynamic system simulation, which completes the second frame. Additional frames are simulated in the same manner as the second.

6.5 Validation Component

The simulation framework also contains automated results validation programs. These programs can be used to check the results of the cluster generator and dynamic simulation to identify errors in their operation. Principally, this consists of using a modified version of the static scheduler that validates the schedule produced by the dynamic scheduling simulation does not contain deadlock conditions. The modified static scheduler examines the detailed schedule trace produced by the dynamic scheduler's logging facility. This validation program utilizes a cycle detection algorithm, which verifies that the produced schedule is acyclic. The validation program also checks that the computed schedule length is correct, and all the precedence constraints of the graph have been met. As a final check, it determines that reconfigurable hardware resources in each PE are sufficient to execute all assigned tasks.

6.6 First Order Simulation Validation

Both the static and dynamic schedulers have been used in previous research efforts. However, the cluster generator is newly developed. To determine that the clustering technique is truly functioning as intended, the simulation requires validation. As part of the validation effort, a series of simplistic task graphs are used to effect specific behavior in the simulation. Analysis of the simulation is provided both in a qualitative manner by examining the structure of the graph, and in a quantitative manner by comparing the results to an independently developed clustering algorithm, referred to hereafter as the validation algorithm. The PE size is constrained for these examples to be twice the average task size. This keeps the number of base types in each meta-task to on average two. Allowing the region size to be too large reduces the solution set to trivial solutions where all types fit into a single meta-type, and no PE is required to perform run-time reconfiguration. Since reconfiguration time is related to the size of the region, size is specified using two parameters. The first is the width of the region in logic columns, the second is the number of bitstream bits per column. Reconfiguration time is determined by the number of columns times the number of bits per column times the effective clock period.

Consider the example application shown in Figure 6.3. The optimal execution of this graph on a system of 3 PEs, including reconfiguration times, is shown as a Gantt chart in Figure 6.4. For short reconfiguration times, the critical path of this graph is Task 0 $\rightarrow$ Task 1 $\rightarrow$ Task 3. As long as the reconfiguration time is less than the available slack on task 2, clustering will not improve the optimal execution of

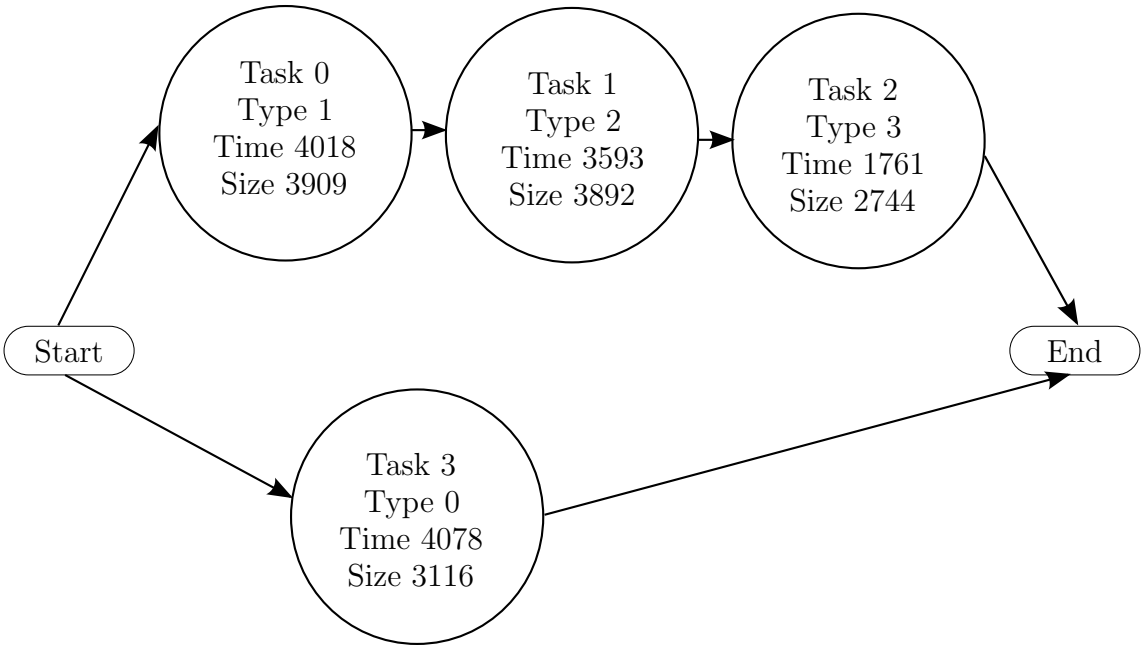


Figure 6.3: Test Graph 1

this graph. For this graph, this slack is equal to the difference in the execution time between Task 3 and Task 0 and Task 1 ($T_{Exec_3} - (T_{Exec_0} - T_{Exec_1})$), and is 3,953 clock ticks. However, if the reconfiguration time exceeds this available slack, clustering task 2 with task 3 will then eliminate the reconfiguration time, and consequently improve execution time. Table 6.4 records the observed execution time for several values of the reconfiguration time. As can be seen, the clustering algorithm behaves as expected.

The next example makes use of the graph shown in Figure 6.5. This graph has the property that it's critical path is directly dependent on the reconfiguration time of the PE. Eliminating the reconfiguration on each path will reduce the execution time to the initial reconfiguration time plus the execution time of the longest path. Additionally, there is no benefit to having more than 2 PEs for this graph once two

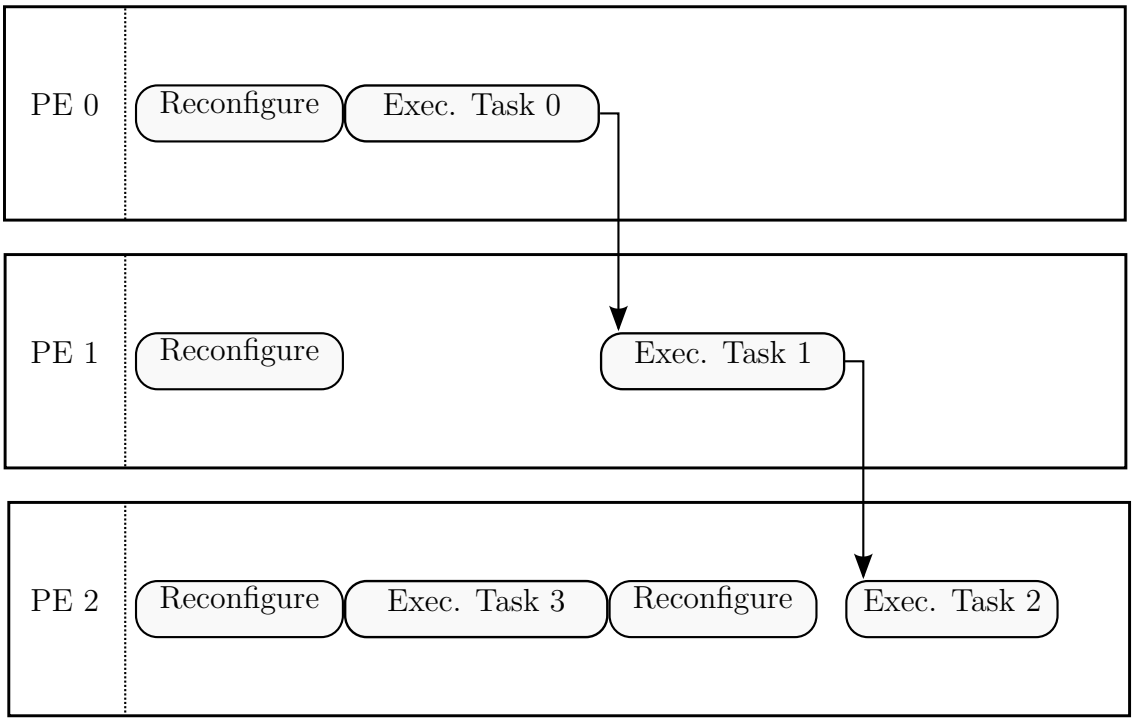


Figure 6.4: Graph 1 Gantt Chart

Table 6.4: Test 1 Graph Clustering Results

			Baseline		Validation	
Reconfiguration Time	Critical Path	Initial Schedule	Clusters (Types)	Clustered Schedule	Clusters (Types)	Clustered Schedule
Results for PE = 2, Region Size = 9,500						
350	9,372	10,072	None	9,722	(1,2),(0,3)	9,722
3,500	9,372	16,372	None	12,872	(1,2),(0,3)	12,872
3,590	9,372	16,552	(0,1),(2,3)	12,962	(1,2),(0,3)	12,962
4,000	9,372	17,372	(1,2),(0,3)	13,372	(1,2),(0,3)	13,372
Results for PE = 3, Region Size = 9,500						
350	9,372	9,722	None	9,722	(1,2),(0,3)	9,722
3,500	9,372	12,872	None	12,872	(1,2),(0,3)	12,872
3,590	9,372	12,962	None	12,962	(1,2),(0,3)	12,962
4,000	9,372	13,372	(0,3)	13,372	(1,2),(0,3)	13,372

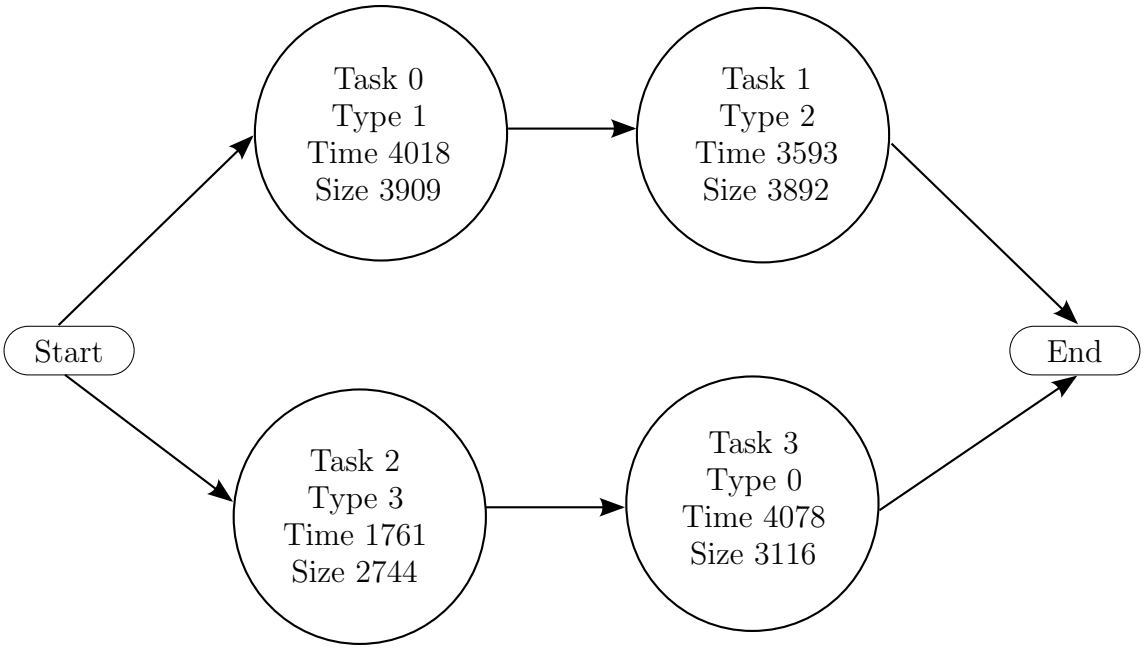


Figure 6.5: Test Graph 2

clusters have been produced. Table 6.5 shows that the simulated execution time behaves as expected.

The third test graph, shown in Figure 6.6, provides no benefit for performing clustering with 3 PEs for any value of reconfiguration time. In this graph, each of the 3 PEs is initialized with one each of the 3 initial tasks. Task 3 is the same type as task 2, so therefore does not require reconfiguration. The critical path of this graph is equal to the length of the longest path *Start* →Task 0 plus the initial reconfiguration time for all values of reconfiguration time. The data in Table 6.6 agrees with the expected behavior over 3 orders of magnitude.

The validation case shown in Figure 6.7 shows a graph with a more complex relationship among the tasks, where the generated schedule impacts performance. When executed on platforms with 2 PEs, the critical path is Task 0 → Task 3.

Table 6.5: Test 2 Graph Clustering Results

			Baseline		Validation	
Reconfiguration Time	Critical Path	Initial Schedule	Clusters (Types)	Clustered Schedule	Clusters (Types)	Clustered Schedule
Results for PE = 2, Region Size = 9,500						
170	7,611	7,951	(1,2),(0,3)	7,781	(1,2),(0,3)	7,781
1,700	7,611	11,011	(1,2),(0,3)	9,311	(1,2),(0,3)	9,311
1,770	7,611	11,151	(1,2),(0,3)	9,381	(1,2),(0,3)	9,381
1,800	7,611	11,211	(1,2),(0,3)	9,411	(1,2),(0,3)	9,411
Results for PE = 3, Region Size = 9,500						
170	7,611	7,781	(1,2),(0,3)	7,781	(1,2),(0,3)	7,781
1,700	7,611	9,311	(1,2),(0,3)	9,311	(1,2),(0,3)	9,311
1,770	7,611	9,381	(1,2),(0,3)	9,381	(1,2),(0,3)	9,381
1,800	7,611	9,411	(1,2),(0,3)	9,411	(1,2),(0,3)	9,411

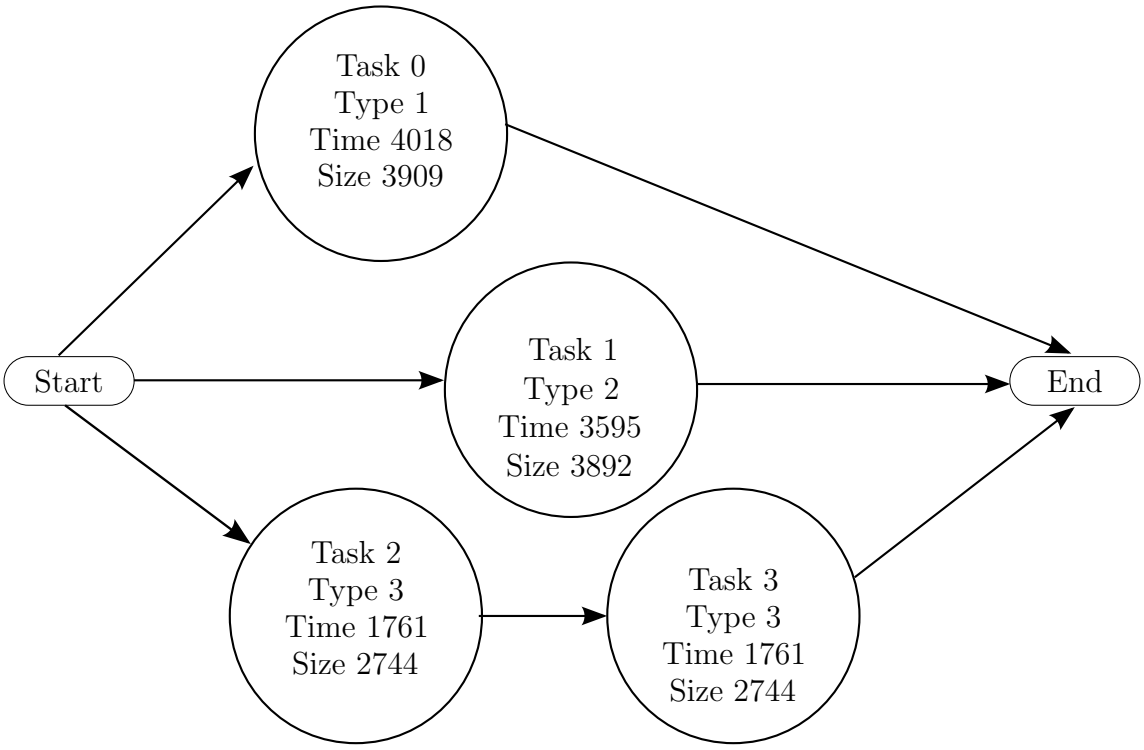


Figure 6.6: Test Graph 3

Table 6.6: Test 3 Graph Clustering Results

			Baseline		Validation	
Reconfiguration Time	Critical Path	Initial Schedule	Clusters (Types)	Clustered Schedule	Clusters (Types)	Clustered Schedule
Results for PE = 3, Region Size = 9,500						
350	4,018	4,368	None	4,368	None	4,368
3,500	4,018	7,518	None	7,518	None	7,518
35,000	4,018	39,018	None	39,018	(1,3),(2,3)	40,779
350,000	4,018	354,018	None	354,018	(1,3),(2,3)	355,779

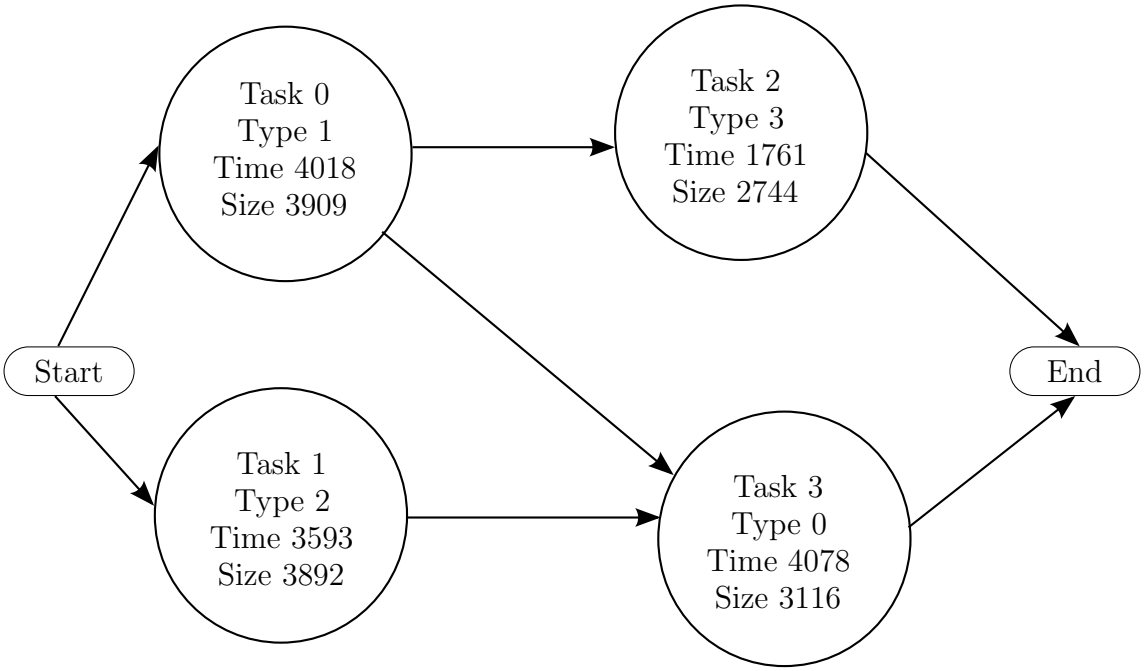


Figure 6.7: Test Graph 4

Combining types 0 and 1 will keep the execution time to the critical path as long as the reconfiguration time is less than the available slack. Once the reconfiguration time exceeds this value, then there is an additional benefit to combining Task 1 and 2 (types 2 and 3).

Table 6.7: Test 4 Graph Clustering Results

			Baseline		Validation	
Reconfiguration Time	Critical Path	Initial Schedule	Clusters (Types)	Clustered Schedule	Clusters (Types)	Clustered Schedule
Results for PE = 2, Region Size = 9,500						
100	8,096	8,196	(0,1)	8,196	(0,1),(2,3) (0,2),(1,3)	8,196
1,000	8,096	10,096	(0,1),(2,3)	9,671	(0,1),(2,3) (0,2),(1,3)	9,096
5,000	8,096	18,096	(0,1),(2,3)	16,689	(0,2),(1,3)	13,096
10,000	8,096	27,671	None	27,671	(0,2),(1,3)	18,096
Results for PE = 3, Region Size = 9,500						
100	8,096	8,196	None	8,196	(0,1),(2,3) (0,2),(1,3)	8,196
1,000	8,096	9,096	None	9,096	(0,1),(2,3) (0,2),(1,3)	9,096
5,000	8,096	17,671	(1,2),(0,3)	14,857	(0,2),(1,3)	13,096
10,000	8,096	27,671	(1,2)	18,096	(0,2),(1,3)	18,096

Based on the data collected from the set of validation test cases, we conclude that the clustering algorithms are behaving as expected. Test cases 1 and 3 indicate that there are cases where clustering does not improve the system performance. The results from test case 2 indicate that clustering does improve performance for cases where reconfiguration is on the critical path of the graph.

6.6.1 Execution Timing

A set of timing measurements were made of each clustering method. The results are presented in Table 6.8, which shows the average run-time of both of the presented algorithms in comparison to the validation algorithm. The values in the

Table 6.8: Average Runtime of Clustering Algorithms

Graph	SA Algorithm	Greedy Algorithm	Validation Algorithm
Graph 1	35.39 s	782 ns	80.15 ms
Graph 2	35.14 s	713 ns	80.04 ms
Graph 3	35.40 s	816 ns	78.49 ms
Graph 4	35.09 s	817 ns	80.45 ms

table represent the average execution time of the specified algorithm over 100 samples. These measurements were made on an quad-core Intel Core i7 based PC with 8 GB of RAM running at 3.6 GHz. The software timer used measures total elapsed time (wall clock time), and has an experimentally determined average resolution of approximately 30 ns. From the table, it can be seen that the greedy algorithm has the best execution time performance, while the validation algorithm is the next fastest, and the simulated annealing algorithm is significantly slower. The simulated annealing algorithm takes considerable time because it invokes the static scheduler multiple times for each temperature in the cooling schedule. The greedy algorithm executes the static scheduler exactly once, as does the validation algorithm. Both the SA and greedy algorithms execute a single method of the static scheduler, while on the other hand, the validation algorithm executes all three static scheduling methods sequentially.

CHAPTER 7

TASK AGGLOMERATION ALGORITHMS

Given the possibilities for run-time improvement based on the grain packing technique, the method for selecting and generating task clusters is a topic of interest. The use of complex heuristic methods is generally applied to the class of problems to which grain packing belongs. However, simple list-based methods can also be applied and have the advantage of lower computational complexity. Since it is unlikely that tasks will be packed at runtime, complexity is not a driving concern, but rather goodness of solution is preferred.

7.1 Selection of Candidate Tasks

To determine whether a proposed clustering is effective, the optimal schedule would first need to be computed for a given application. Then, the optimal clustering would need to be found, and then the optimal schedule for the clustered application would need to be calculated. Clearly, this becomes prohibitive for applications with just a moderate number of tasks. Two algorithms have been developed to generate candidate task clusters. The first uses a simulated annealing heuristic to generate and

evaluate clusters, while the second uses a less complicated list based methodology to develop a proposed task clustering.

As previously noted, grain packing is a problem of the order NP [104]. As such, no polynomial time solution is known to identify optimum task clusters. It is therefore not reasonable to expect that a reconfigurable system with hard real-time constraints can dynamically cluster the task system at run-time. Additionally, since each reconfiguration bitstream is generated for a specific location within the reconfigurable device, dynamic clustering would also require the synthesis of a new bitstream for the clustered meta-task. Such synthesis has a runtime on the order of several minutes, imposing an unacceptable amount of latency. Given these constraints, the analysis and generation of task clusters will be performed at compile time, when system performance will not be impacted.

Given an input task graph, a method is required to select candidate tasks for clustering, a means must then be provided to perform task clustering. Once the tasks have been selected and combined, a metric must be determined to evaluate the relative improvement in the application. This chapter discusses the specifics of the task selection and combination algorithms. Both of the proposed algorithms rely on the static scheduler described in the previous chapter to generate a grading metric in order to perform their task. Both algorithms use the makespan returned by the static scheduling simulation for this purpose.

Once tasks have been selected, a task combination function is required to combine two or more individual tasks into a task cluster. This function takes as its argument a set of tasks and returns a proposed task graph in which the input tasks

have been combined using some combination criteria. This new graph is required to meet all of the original graph's precedence constraints. Although a new task graph has been generated, it is merely a proposed solution, and the actual task graph may not be updated unless the proposed graph represents an actual improvement in execution time.

To determine the relative improvement of the modified task graph, a metric is required to provide a quantitative evaluation of the proposed clustered graph. In order to generate this metric, an objective function is used to examine the proposed graph. For the purposes of the scheduling discussion, the runtime of the proposed graph can be used. The objective function takes as its arguments a clustered task graph and a description of the platform, and returns the value of the execution time of the scheduled task graph on that platform. In the grain packing problem, a proposed cluster is better than another proposed cluster if the objective function returns a shorter execution time.

Since NP problems cannot be solved in polynomial time, a class of solutions known as heuristic algorithms has been developed to address such problems. Two such algorithms are proposed to select the candidate tasks for clustering. These selection algorithms work in a broader framework with the combination and objective algorithms to iteratively converge on a clustered task graph that represents a significant improvement over the original unpacked graph.

7.2 Simulated Annealing Technique

The first task selection algorithm uses a simulated annealing (SA) heuristic to control the task clustering process. SA is a well-known algorithm that is often applied to NP problems because it can be run in deterministic time and produces good, but not necessarily optimum, solutions. These algorithms use a hill-climbing approach to tend to an optimum value. In order to escape possible local extrema, a probabilistic element is employed to enable solution space exploration. Over time, this probabilistic element decreases so that the search stays in the vicinity of the best solution found so far.

Figure 7.1 depicts the flow of the SA based clustering algorithm. The algorithm begins with a list of all of the tasks sorted numerically according to their task ID. Then, the algorithm selects a task ID randomly from the list. Beginning with the next task ID, each subsequent task is evaluated to determine if the linear combination of the two task's resource usage will exceed the available resources in the hardware PE. If the partition's resources are not exceeded, the task combination function is invoked to combine the tasks.

The algorithm then evaluates the next task ID in the list to see if it can also be added to the cluster. If so, the task is added using the same clustering logic and advances to the next task. It proceeds in this manner until it reaches the numerically last task ID on the list.

Once the clustering phase is complete, a new objective function value is calculated for the updated task system. The objective value is compared to the schedule

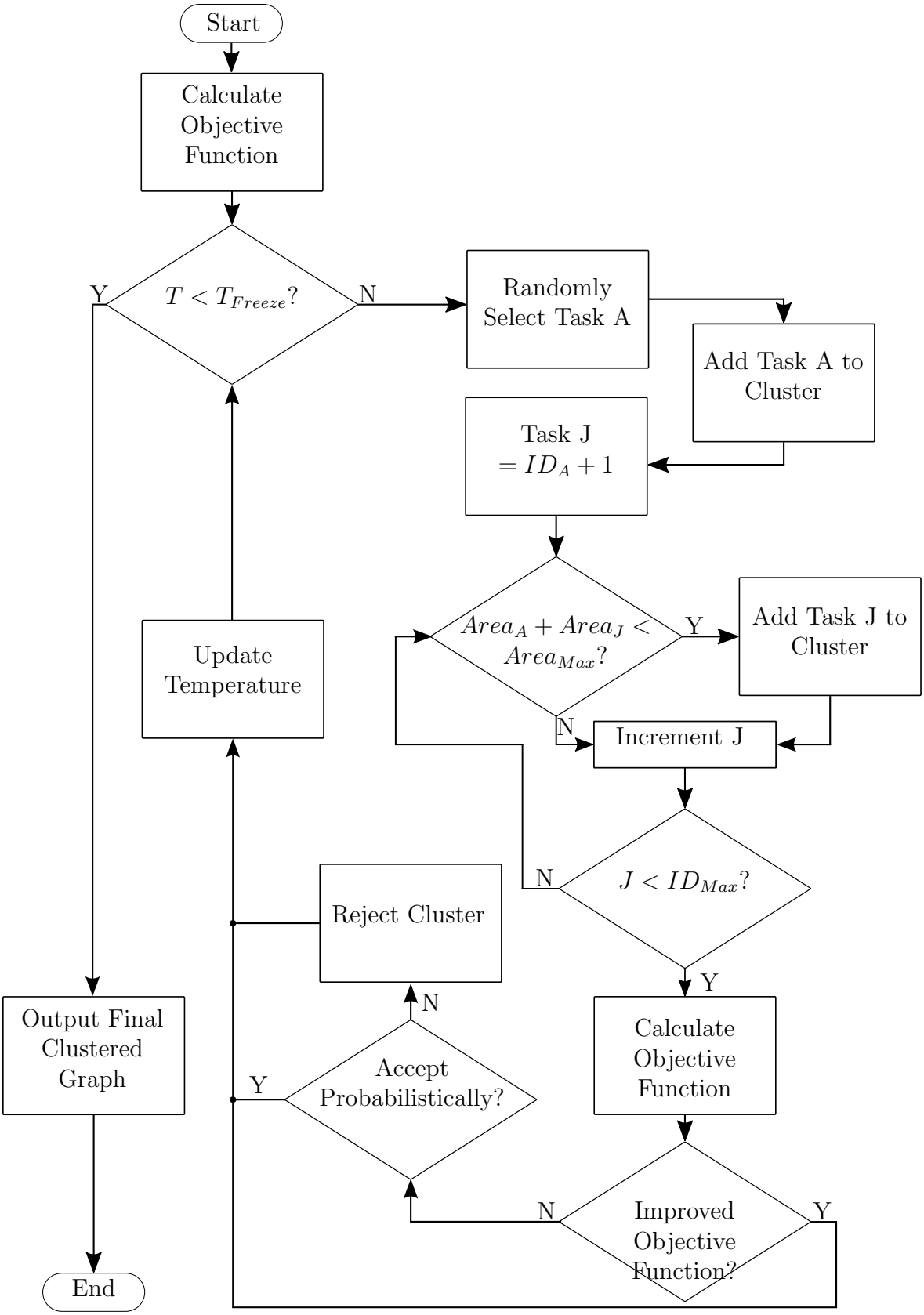


Figure 7.1: Simulated Annealing Algorithm

length of the previous task system. If the new system's schedule length is shorter than the previous, it is accepted, and the algorithm proceeds to the next iteration by randomly selecting a new start task ID. In the case that the new schedule is not improved, the new system may be accepted probabilistically. The probability of accepting a worse schedule decreases exponentially with each iteration of the SA algorithm. The algorithm continues to execute until the temperature is less than or equal to the freeze value.

The cooling schedule used in this SA algorithm dictates that the temperature parameter is reduced to 95% of its current value each cooling step. T was set initially to 50, with a freezing temperature of 0. At each isothermal step, clustering is performed once per iteration. The probabilistic acceptance criterion was to accept a worse solution if a uniform random number was less than the value returned in (7.1), where T is the current temperature, Δu is the difference between the current objective function value and the previous objective function value, and C is 1,000.

$$\mathsf{P}[accept] = e^{-C/(T\Delta u)} \tag{7.1}$$

7.3 Greedy Algorithm

The second clustering technique uses a simple greedy list based clustering algorithm, as shown in Figure 7.2. This clustering algorithm makes use of the static scheduler's ordering and PE allocation phase. Once the static scheduler has determined a task to PE mapping, the clustering algorithm examines the task schedule for each PE. Starting with the first task allocated to the first PE, the algorithm examines

the resource requirements for the task. It then examines the next task in the schedule to see if both task's requirements can be met simultaneously by the PE. If so, the two tasks are combined using the same type conversion logic as the SA approach. The algorithm then proceeds through the remaining tasks assigned to the PE, adding each task to the cluster if its requirements can be met in the PE with the cluster. The clustering algorithm then proceeds to the tasks assigned to the next PE and tests these tasks for clustering in the same fashion, and so on for all remaining PEs.

7.4 Analysis

While the two clustering algorithms perform similar functions, their complexity is significantly different. The greedy algorithm is significantly better, requiring two simple loops, one iterating over the number of PEs and the other iterating over the number of tasks assigned to each PE. Assuming that on average, each PE is assigned the same number of tasks, $\frac{n}{PE}$, the average complexity of the algorithm is given by (7.2).

$$\mathcal{O}(PE\frac{n}{PE}) \tag{7.2}$$

which reduces to (7.3)

$$\mathcal{O}(n) \tag{7.3}$$

On the other hand, the SA algorithm is more computationally complex. The SA algorithm also consists of two loops, the temperature loop, and the isothermal loop. However, internal to these loops, the static scheduler is used to estimate the

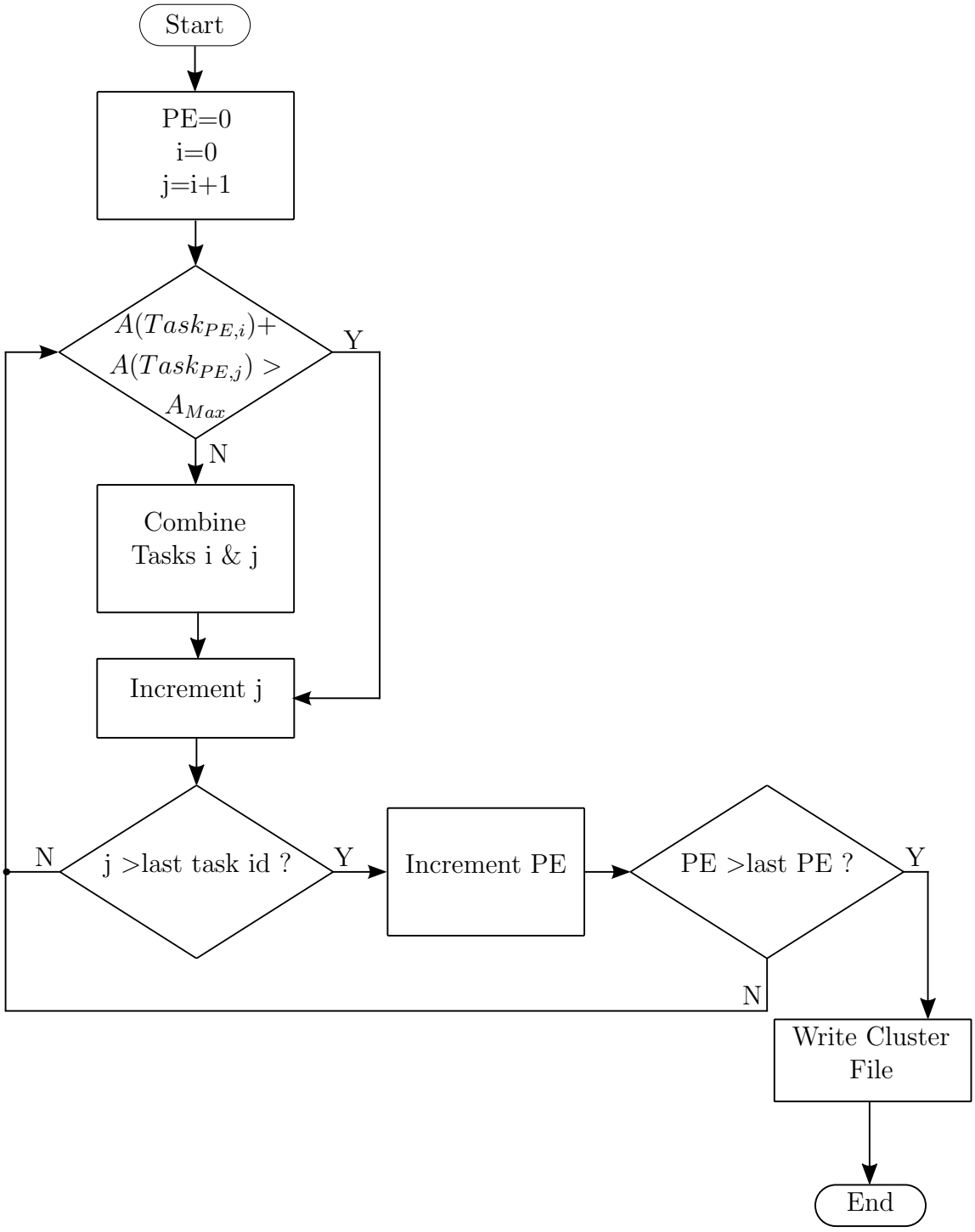


Figure 7.2: Greedy Algorithm

objective function each iteration. The performance of the static scheduler is highly dependent on the number of tasks in the graph since it must generate candidate priority lists and evaluate them.

For the purposes of the clustering algorithm, the static scheduler was restricted to using only the GA scheduling approach when evaluating the objective function. Although better than performing all 3 optimization methods, this is still very complex. As shown in the previous chapter, the complexity of the GA static scheduler is $\mathcal{O}(GnPn^2PE)$. Letting T represent the total number of temperature steps, $ln(T_{final}/T_{initial})/ln(\lambda)$, the total complexity of the SA clustering algorithm is shown in (7.4).

$$\mathcal{O}(TiGnPn^2PE) \tag{7.4}$$

Although the greedy algorithm is computationally less complex, it does not evaluate it's solutions to determine if they have improved or not. This is an advantage that the SA algorithm does offer. However, this comes at a significant penalty, since invoking the static scheduler significantly increases the runtime of the SA algorithm.

CHAPTER 8

SIMULATION RESULTS

This chapter presents a discussion of the conditions that were used to determine results as well as a discussion of the statistical methods that were used to analyze the simulation outputs.

8.1 Simulation Setup

In order to determine the effectiveness of applying the grain packing method over a wide array of parameters, the simulation was run using a variety of synthetically generated task systems. Synthetic task graphs were generated using an open source tool, Task Graph for Free (TGFF) [139]. This software was originally developed by researchers at Princeton University to provide a flexible and portable way to generate pseudo-random task graphs for use in scheduling and allocation research. This allows researchers to replicate other researcher's synthetic task graph structure by entering the same input parameters and input seeds.

Variation in the simulation results is produced by altering key parameters associated with the simulation. These parameters are generally provided in the input file discussed in Chapter 6. Broadly, these parameters can be divided into three

Table 8.1: Variable Graph Parameters

Graph Family	Num. Graphs	Num. Types	Num. Successors
1	1	5	2
2	1	5	5
3	1	20	2
4	1	20	5
5	3	5	2
6	3	5	5
7	3	20	2
8	3	20	5

Parallelism

groups. First, are the parameters affecting the generation of task graphs. The second group of parameters affects the available resources of the simulated platform, while the remaining parameters affect the execution of the simulation. The first group is used as inputs to the TGFF program to control the properties of the graphs. The remaining parameters are provided directly to the dynamic system simulation in order to affect its execution.

The synthetic task generation parameters are shown in Table 8.1. As can be seen from the table, there are three parameters at two levels each, resulting in 8 sets of parameters. The effect of varying these particular parameters result in the graph families ranging from highly sequential to highly parallel graphs. Five graphs were generated for each combination of parameters, resulting in 8 families of 5 graphs, for a total of 40 graphs. These same graphs have been used with the dynamic simulation previously to evaluate its performance over differing degrees of parallelism.

Each of the test cases is run for all combinations of the platform parameters shown in Table 8.2. Primarily this consists of altering the number of PEs available.

For a given simulation, the number of resources in each PE is determined by dividing the total device resources by the number of processors. The amount of resources in the total device is set to a fixed value, given by the size parameter, which is used for all simulation cases. Device size was fixed because resource constraints are a function of both device size and task resource requirements. Since resource requirements are varied in the task graph, device size can be set at a constant value.

The available platform resources should be somewhat restrictive, in order to eliminate the trivial case where all tasks can be programmed concurrently in the device. In most cases, the resource requirements for the entire graph are a function of the number of PEs, the number of concurrently executing tasks, and the number of types. Since graph properties vary, two methods were considered for selecting this parameter. In the extreme case, the resource limits can be set based solely on the total number of tasks in the graph. Another option would be to constrain the platform based on the number of types in the graph. For this simulation, the platform resources based on the second option, as this represents a more restrictive model.

In the proposed application model, resource usage is an attribute associated with the task type. Each graph has a unique set of types with varying resource requirements as generated by the TGFF program. Since the per task resource usage is a nondeterministic parameter, each graph has unique resource constraints. In the previously used synthetic application graphs, this as a uniform random variable on $[2,000-4,000]$ with mean 3,000 for all graphs. Synthetic task graphs contain either 25 or 75 tasks each. If the platform resource capabilities are set based on the number of tasks in the graph, this would result in platform resources between $[75,000-225,000]$.

Table 8.2: Platform Parameters

Parameter	Values
Num. of PEs	3, 4, 5
Total Device Size	30,000

On the other hand, the number of types per graph is set at either 5 or 20 depending on the graph. Using this as the parameter for platform resources yields a total device size between $[15,000 - 60,000]$.

Instead of using the the number of tasks to set the platform resources, the more restrictive values were used. The fixed device size was chosen so that the average task requirements would be exactly met by a graph with 10 types. This value was chosen since it is between the minimum and maximum values of type quantity, 5 and 20 respectively. The number of tasks was allowed to vary with the graph parameters. Multiplying 10 types by 3,000 resources per type yields a total device size of 30,000. Using this fixed size, half of the graphs are over-constrained, while the remaining graphs are under-constrained based on task typing.

In keeping with the platform model, the reconfiguration time is derived from the amount of resources in the reconfigurable region. For the simulation, the total device size is a fixed parameter, as previously noted. To derive the reconfiguration time, the simulation first divides the total resources by the number of reconfigurable PEs to determine the amount of resources in each PE. The simulation assumes that each device corresponds to one word of bitstream memory, and that the simulated reconfiguration controller can transfer one word per clock. Another parameter, the

clock scale factor, multiplies this base reconfiguration time and can be used to adjust either the number of devices per bitstream size or words per clock. The simulation was performed with clock scale factors of 1 and 2.

A dynamic execution simulation was performed for each of the 40 graphs on simulated platforms corresponding to all combinations of platform parameters. The dynamic execution simulation was controlled using the simulation parameters shown in Table 8.3. The first of these parameters controls how the task execution time is interpreted by the simulation. In the deterministic case, the task execution time is a parameter associated with the task type, and all tasks of the same type execute in the same fixed amount of time. For nondeterministic task execution, all tasks of the same type have the same average execution time, $\bar{t}_{exec}$. Each frame, a uniform random number is generated on the interval $(.8\bar{t}_{exec}, 1.2\bar{t}_{exec})$, this random number is used as the actual execution time for the frame it is generated in.

The second simulation parameter controls the degree of graph nondeterminism π. This parameter is proportional to the number of control edges in the graph. Before execution of the simulation, a number of the existing edges in the graph are selected at random to represent control dependencies. The total number of selected tasks is controlled by the relation $\frac{\pi}{10} n_{tasks}$. When an edge is selected to be a control edge, the probability it will be executed is also determined. Execution probability is between 0.05 and 0.9. As described previously, this probability represents the proportion of frames in which the control edge will appear. During simulation of each frame, the edge will be evaluated after its source completes, but before its sink is scheduled. This evaluation will determine if the edge will appear in the frame or not.

Table 8.3: Simulation Parameters

Parameter	Values
Task Execution Time	Deterministic, Nondeterministic
π	0, 1, 2

Table 8.4: Simulation Classification

Classes	Intertask Dependencies	Task Runtime
Class 1	Deterministic	Deterministic
Class 2	Deterministic	Nondeterministic
Class 3	Nondeterministic	Deterministic
Class 4	Nondeterministic	Nondeterministic

The combination of the simulation parameter values results in a range of simulations from fully deterministic to highly dynamic. Table 8.4 presents a classification based on these parameter combinations. The combination of deterministic graphs combined with $\pi = 0$ results in fully deterministic execution, where every frame is executed identically. At the other extreme, the combination of nondeterministic task graphs with $\pi > 0$ results in a highly dynamic simulation cases where both the task graph and individual tasks vary from frame to frame. It is important to note that there are twice as many samples in classes 3 and 4 vs classes 1 and 2. This is due to the value of π having two nondeterministic values (1 and 2) but only a single deterministic value (0). For this reason, the results presented in the following section will be presented with value of π being explicitly given.

Algorithm 5 Simulation Execution

```
1: for Algorithms in No Clustering, Simulated Annealing, Greedy do
2:   for GraphDeterminism in Deterministic, Nondeterministic do
3:     for Pi in 0 to 2 do
4:       for PEs in 3 to 5 do
5:         for Graph in 1 to 8 do
6:           for Sample in 1 to 5 do
7:             for Frame in 0 to 39 do
8:               if Algorithm != No Clustering then
9:                 ClusterGenerator()
10:              end if
11:              DynamicSimulation()
12:            end for
13:          end for
14:        end for
15:      end for
16:    end for
17:  end for
18: end for
```

Finally, 40 separate frames are simulated for each combination of the above parameters for each of the 40 total task graphs. In all, nearly 29,000 simulations are run between all combinations of parameters, graphs, and frames. Running multiple frames allows the dynamic simulation to show the long-run effects of the dynamic simulation for a given set of parameters. Algorithm 5 presents a basic description of how the simulation parameters are modified. During each frame, the dynamic simulation is run first. After completion of the dynamic schedule, the actual task graph that was executed is recorded. This task graph is then provided to an instance of the static simulator, which is used to bound the best case performance of the runtime. Once each frame is completed, the total simulated runtime of the graph is reported from both the static scheduler and dynamic scheduler. Additionally, the time spent in each dynamic scheduler state is recorded on a per PE basis.

Table 8.5: Deterministic Simulation Cases

Simulation Case	Partition Size	Algorithm
1	Small	Simulated Annealing
2	Medium	Simulated Annealing
3	Large	Simulated Annealing
4	Small	Greedy
5	Medium	Greedy
6	Large	Greedy

8.2 Deterministic Execution

The grain packing method has been shown previously to be effective for fully deterministic execution in reconfigurable systems [140]. To summarize these results, the cluster generator was run without enabling the dynamic execution simulator. Since the cluster generator runs the static scheduler before performing clustering, a speedup value can be calculated by comparing the initial static schedule to the final objective function value returned by each of the clustering algorithms.

Deterministic schedule estimation was performed for each of the input task graphs described previously. Since deterministic execution is being considered, the simulation was run for a single frame only, with the conditions shown in Table 8.5. In these test cases, the partition size was varied to contain 5%, 15%, and 30% of the total device logic resources, corresponding respectively to a small, medium, and large partition size.

Figure 8.1 shows the results of clustering on the runtime of the synthetic graphs. The total speedup for each test case was calculated relative to the original runtime, and then averaged over the total set of graphs.

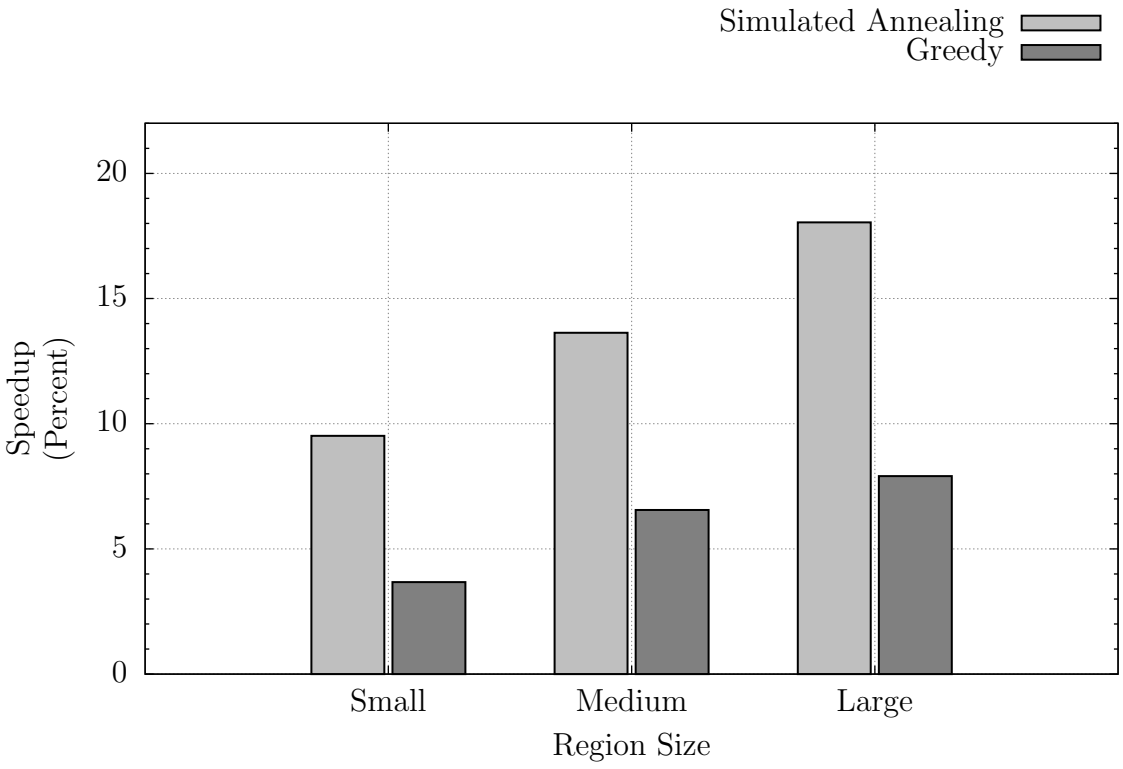


Figure 8.1: Average Speedup of Clustering Methods under Deterministic Conditions

The static scheduler decomposes the total application runtime into execution time, reconfiguration time, and idle time. Using these categories, an efficiency metric can be calculated by taking the ratio of the total execution time to the total runtime. Figure 8.2 shows the result of this efficiency calculation for these test cases. It is important to note that in the baseline case, runtime is not affected by the size of the reconfigurable region. However, the region size affects the number of tasks that can be clustered together, which reduces reconfiguration time after clustering. The results of clustering on the reconfiguration time are shown in Figure 8.3. As noted in the example discussed previously, reducing reconfiguration time may result in increased

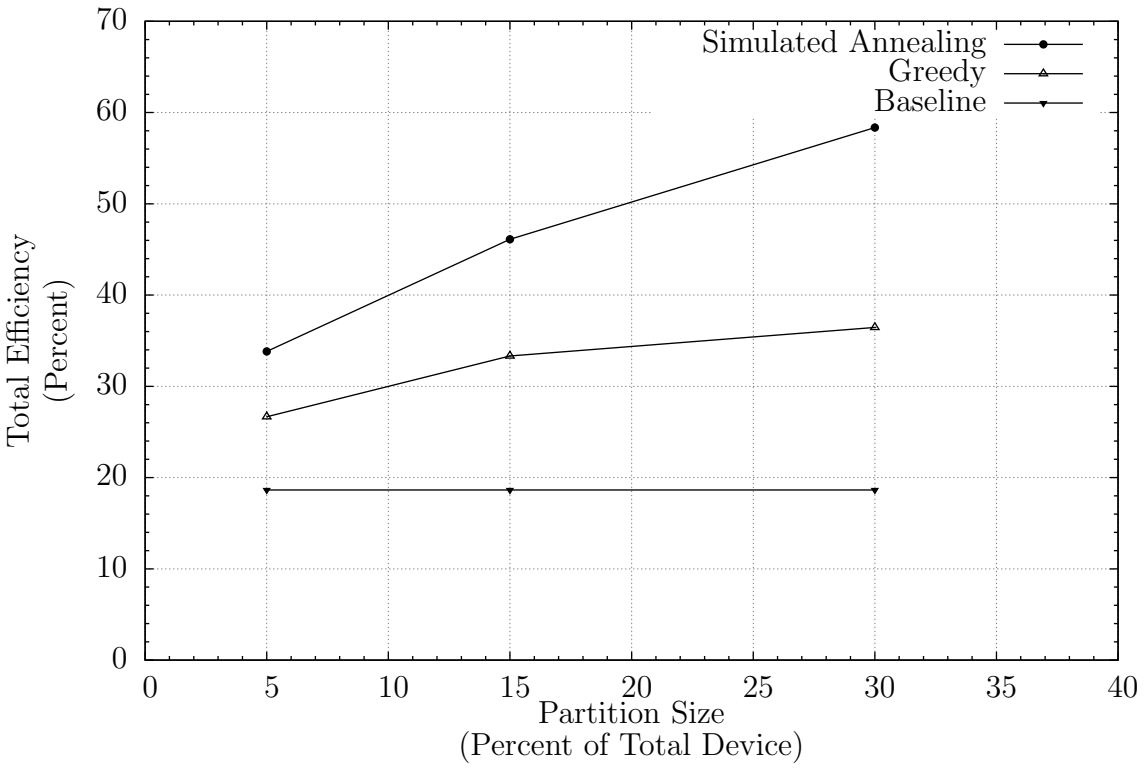


Figure 8.2: Average Efficiency under Deterministic Conditions

idle time. In order to help quantify this increase, the effects of clustering on the idle time component of the runtime are shown in Figure 8.4.

8.3 Dynamic Scheduling Results

In the fully deterministic case, the results from the dynamic scheduler are identical from frame to frame. For this reason, the fully deterministic cases are excluded from the dynamic scheduler results. Results from the dynamic scheduler are presented in Figure 8.5 through Figure 8.10. In these figures, the average graph runtime has been calculated over 40 frames for each of the 40 graphs. Each box shows the range of the average runtime over the 5 graphs in each graph set, with a box for

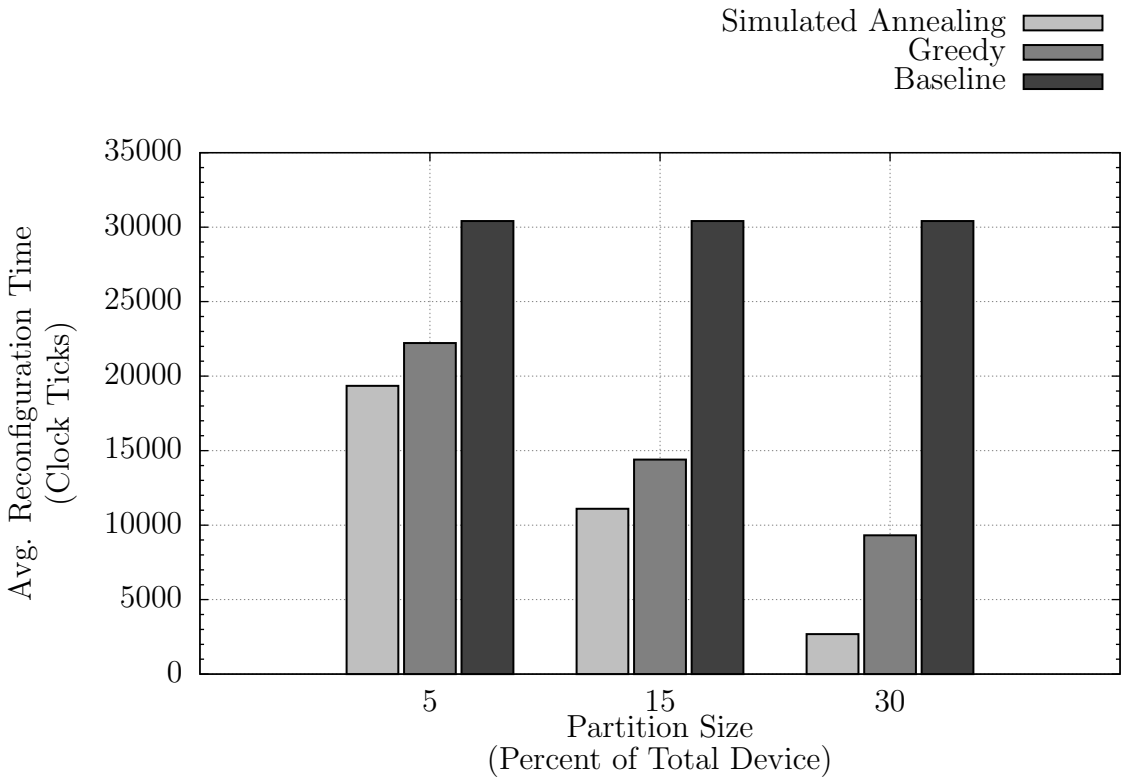


Figure 8.3: Average Reconfiguration Time under Deterministic Conditions

each of the 8 graph families. For each family, the center box shows the initial runtime without clustering, while the boxes showing runtimes after applying the Simulated Annealing and greedy clustering methods are shown respectively to the left and right of this baseline.

8.3.1 Statistical Analysis

Given the large amount of data produced by the simulation, it is difficult to analyze the results based on the raw data from the simulation. Traditionally, a solution would be to create a randomized multiple factor designed experiment [141]. An Analysis of Variance (ANOVA) procedure could then be used to identify statistically

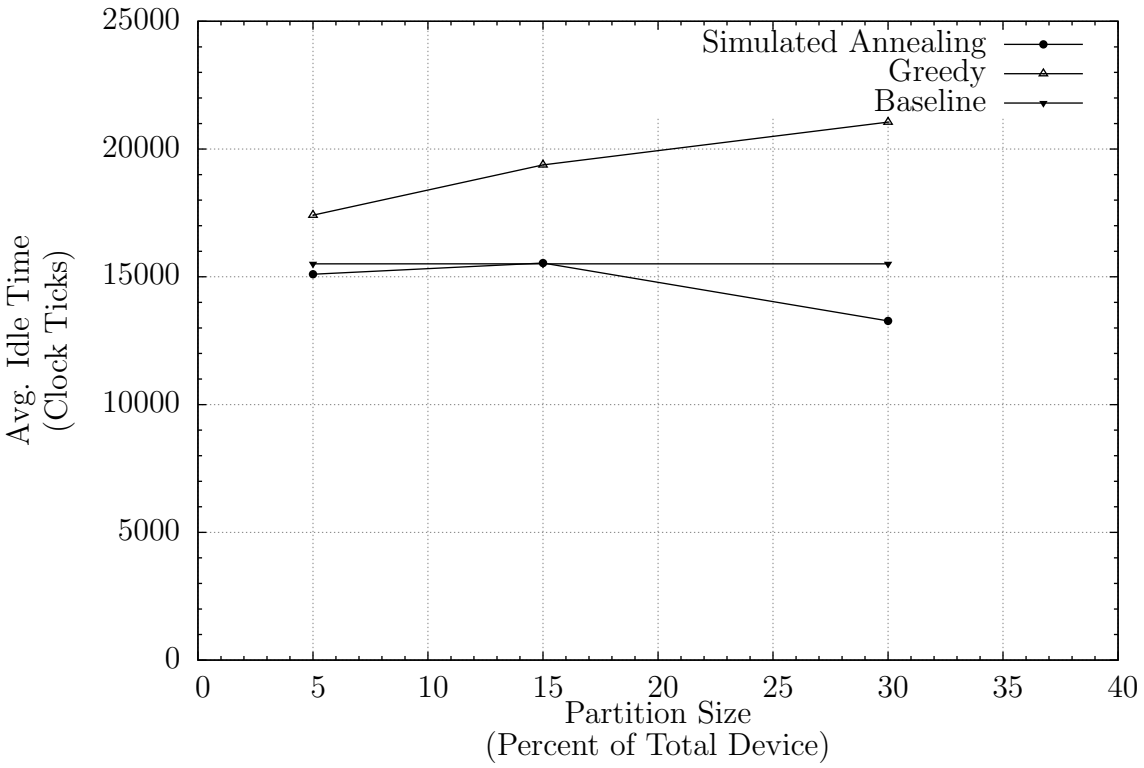


Figure 8.4: Average Idle Time under Deterministic Conditions

significant effects. However, the randomized multiple factor design approach may not be appropriate for data sets generated by computer simulations where there is a lack of random effects [142].

The simulation relies on several pseudo-random parameters to generate variability. Since these parameters are not fully random, the results of individual runs are exact; the same set of inputs results in the same output. This results in a lack of variability between runs with the same set of inputs. As there is not any variance due to experimental error, the results are compared without using a randomized experimental design.

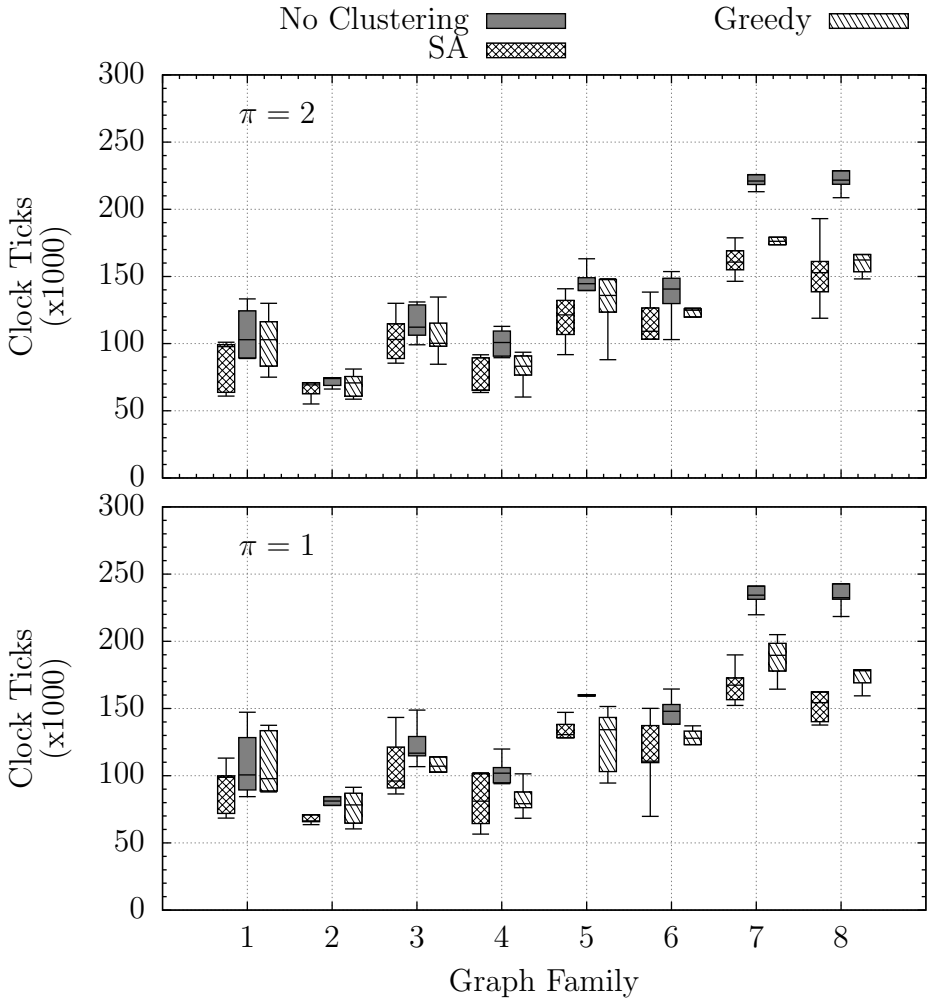


Figure 8.5: Dynamic Scheduler Results with 3 PEs and Deterministic Task Weights

Rather than using a fully randomized designed experiment, classical hypothesis testing was used to identify significant effects generated by the simulation. In hypothesis testing, a treatment is applied to a set of samples and compared to a reference treatment. A response variable is recorded, and responses are analyzed statistically. If a treatment truly affects the response variable, the responses will be drawn from different populations, and statistical methods can be employed to show

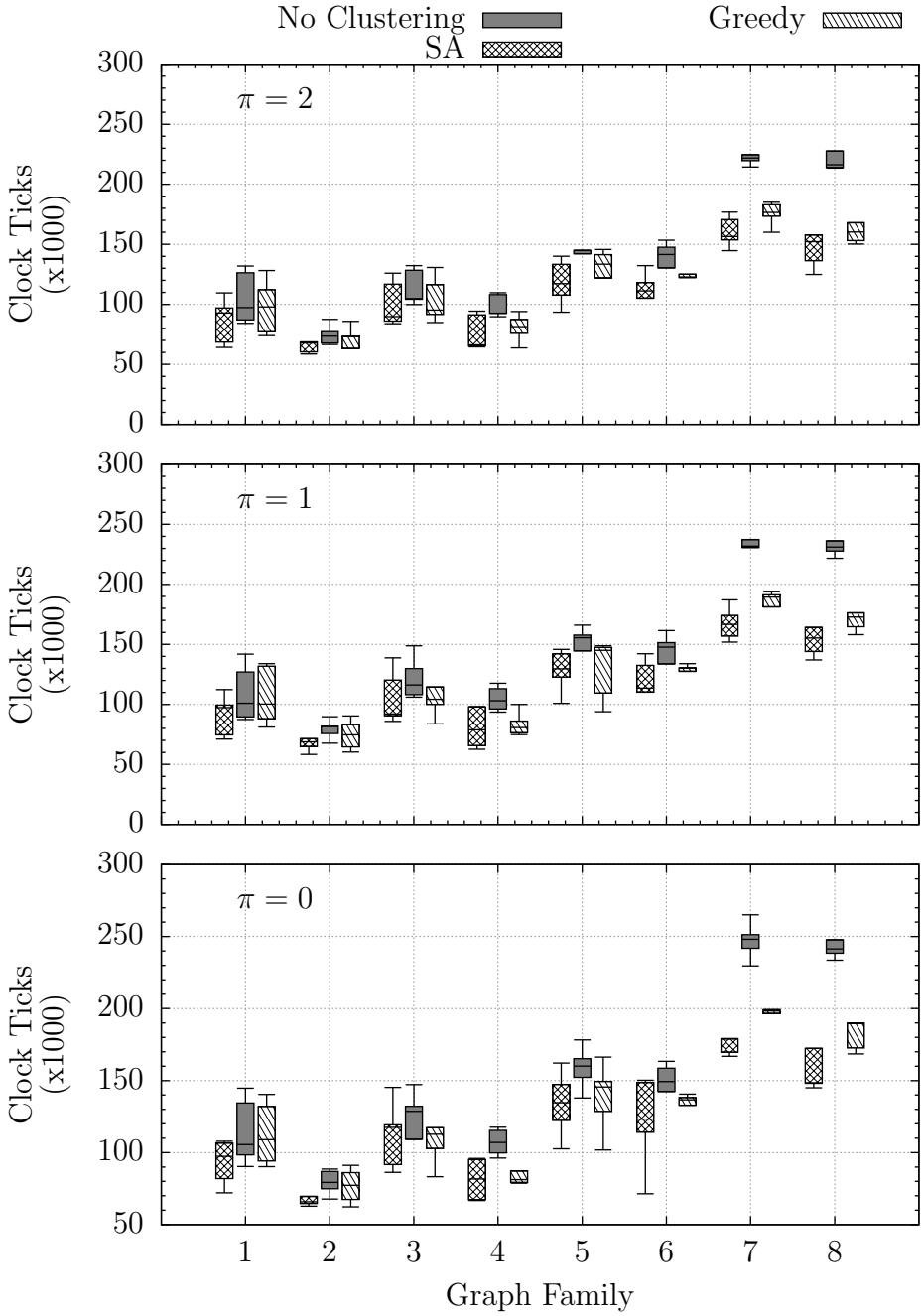


Figure 8.6: Dynamic Scheduler Results with 3 PEs and Non-deterministic Task Weights

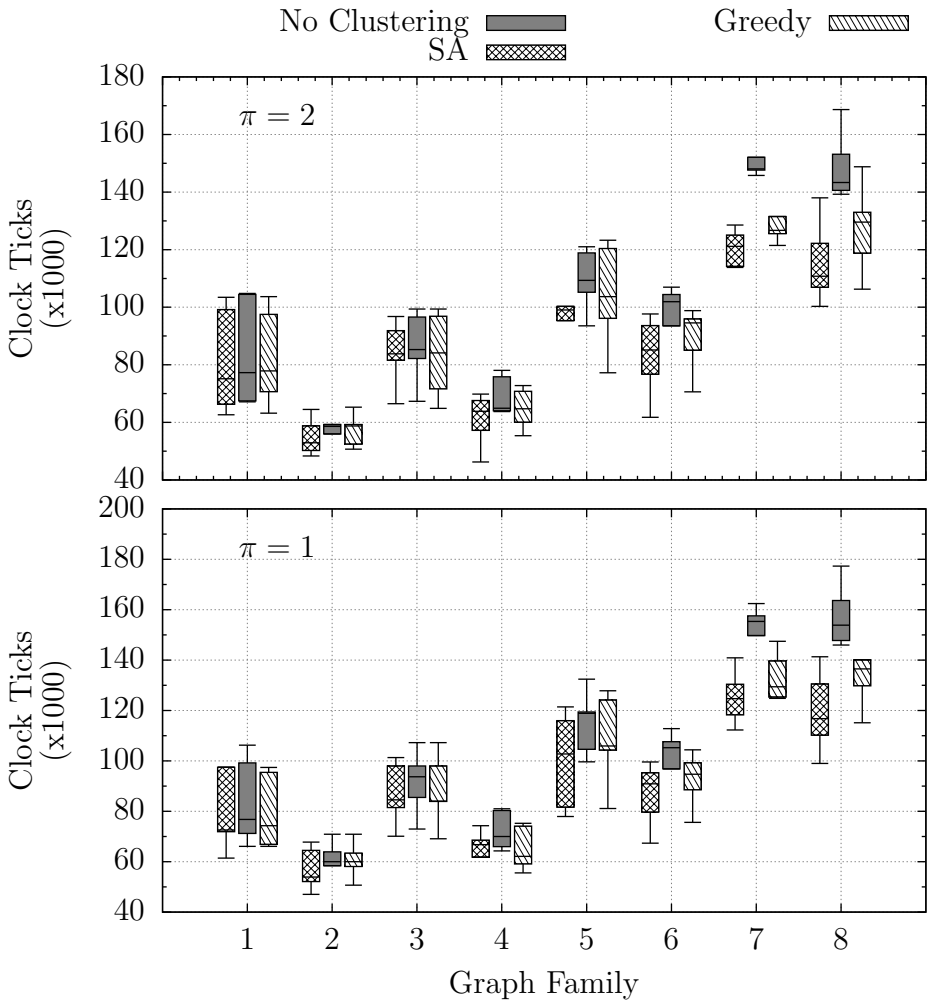


Figure 8.7: Dynamic Scheduler Results with 4 PEs and Deterministic Task Weights

this. If a statistical difference is detected, the treatment is said to be statistically significant.

Results were analyzed using a directional two sample t-test. A two sample t-test directly compares the means of the response variable resulting from different treatments. A directional two sample t-test tries to determine if one population's mean response is greater than another's, the hypotheses being tested are:

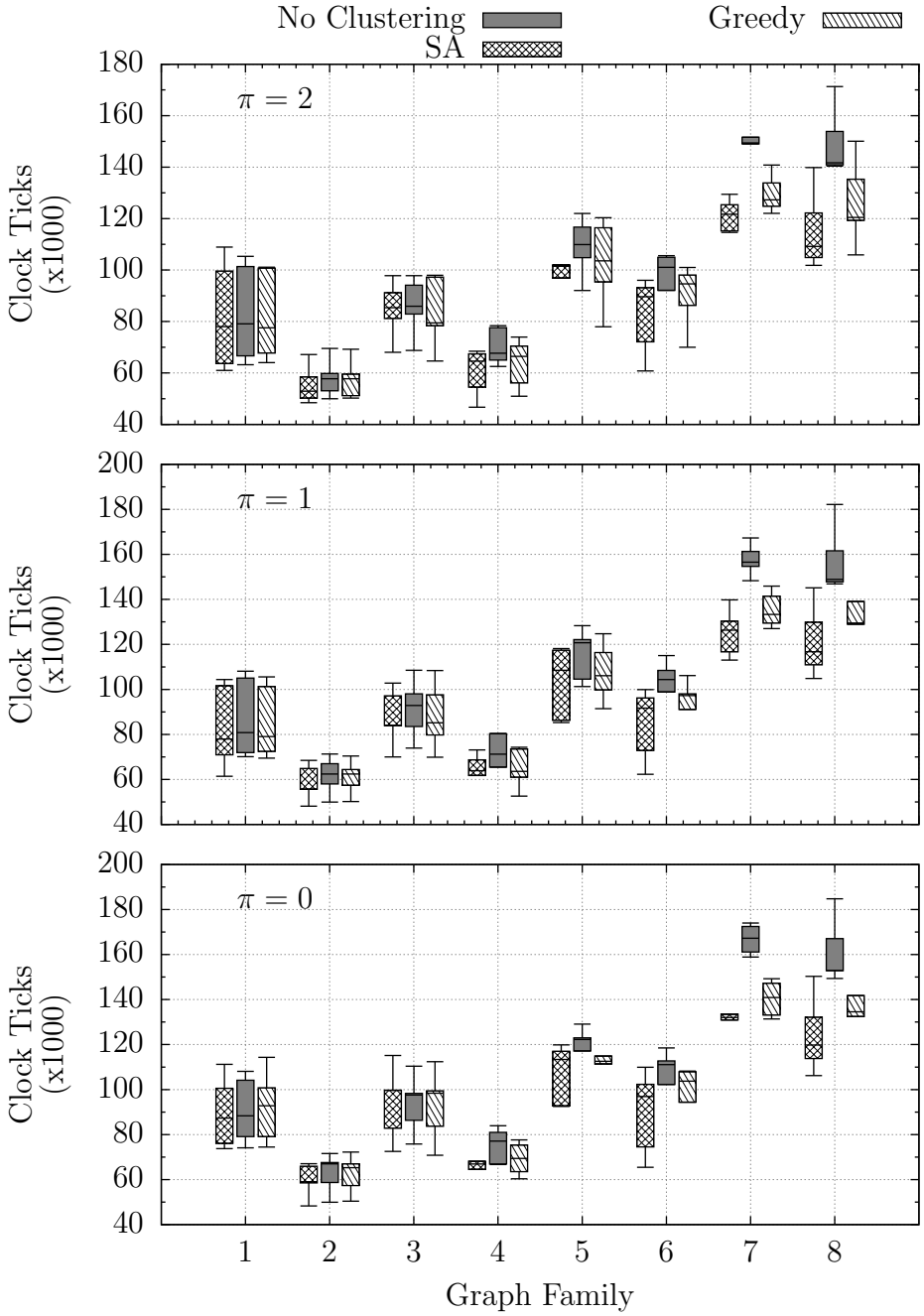


Figure 8.8: Dynamic Scheduler Results with 4 PEs and Non-deterministic Task Weights

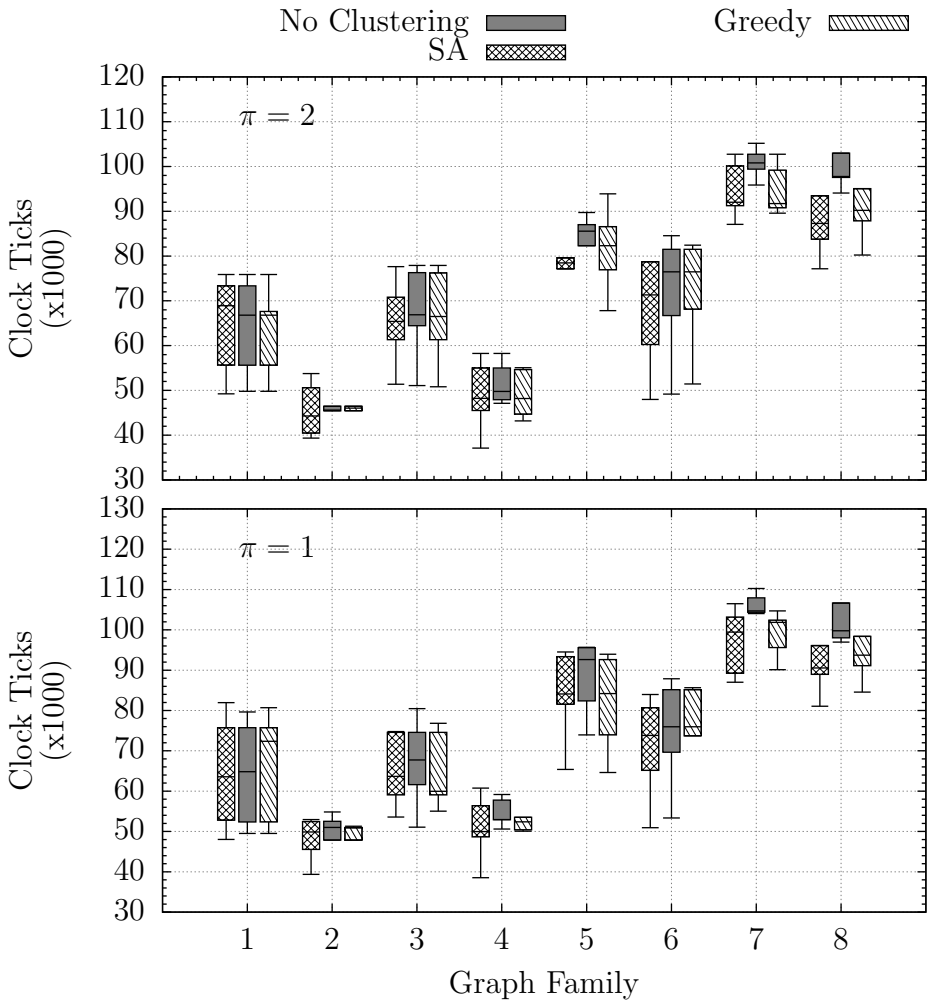


Figure 8.9: Dynamic Scheduler Results with 5 PEs and Deterministic Task Weights

$H_0 : \mu_1 \leq \mu_2$

$H_1 : \mu_1 > \mu_2$

Where μ_1 and μ_2 are the population means of the unclustered and clustered runtimes respectively. Stated another way, the test is designed to determine if the results that have been obtained are drawn from different populations where one population has a greater mean response. Hypothesis testing initially assumes that the null hypothesis, H_0 is true. If the responses of the clustered and unclustered graphs

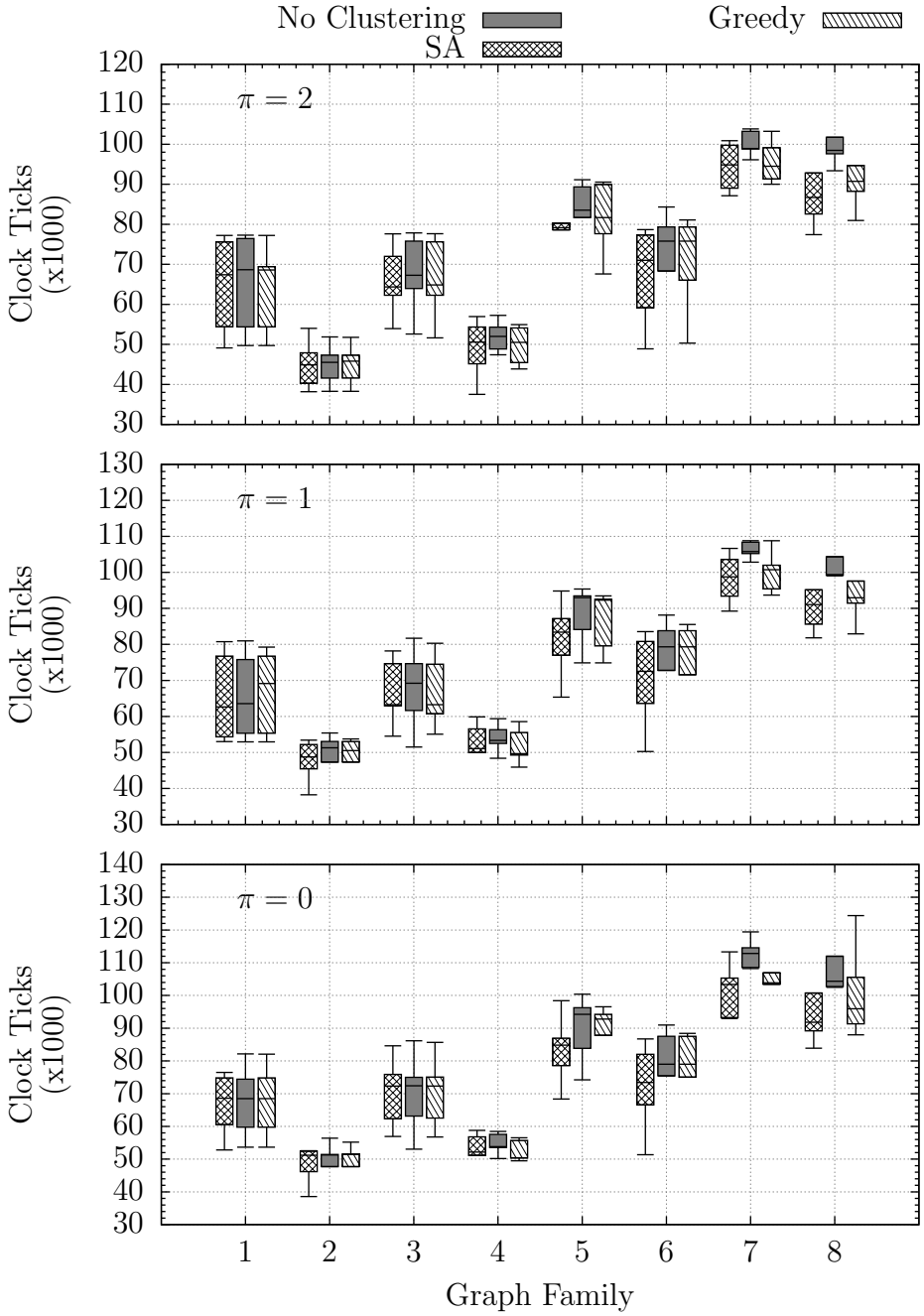


Figure 8.10: Dynamic Scheduler Results with 5 PEs and Non-deterministic Task Weights

differ by a statistically significant amount, H_0 can be rejected as not true, leading to the conclusion that the alternative hypothesis, H_1, can be accepted.

To determine if the populations are identical, a significance level is first chosen. Typically 0.05 is chosen as the significance level. Rejecting the null hypothesis at this level of significance implies that there is less than a 5% chance that the samples are from the same population. The calculations for the t-test involve calculating a statistic involving the difference of the sample means, and comparing the results to the value of the Student's t probability distribution for the appropriate degrees of freedom and significance. If the calculated statistic is less than the value of the t distribution with 0.05 significance, there is less than 5% chance the samples are from the same population. In this case, H_0 can be rejected

The hypothesis testing model does not identify compounded treatment effects, but rather examines each treatment separately. Independent hypothesis tests must be performed for all combinations of each treatment to identify significant effects. To eliminate excessive computations, the hypothesis test was performed to establish significance for the clustering approach. Additional significant factors were identified by graphing the number of significant improvements vs. treatment variables.

A standard t-test assumes that the treatment samples are drawn from normally distributed populations with similar variances. The t-test is reasonably robust to non-normal populations, but not robust to unequal variances [143]. Since the population variances may or may not be equal, it is advisable to determine mathematically if the samples are drawn from populations with the same variance. This involves another hypothesis test, this time with the assumption that the variances are equal. If the

sample statistic is less than the F distribution with the appropriate degrees of freedom and significance, there is good evidence that the variances are equal. If the variances prove to be unequal, the t-test may still be employed, but the degrees of freedom are adjusted to reduce the possibility of rejecting the null hypothesis when it is true (known as committing a Type I error).

Although there are 5 graphs in each set, the graphs were found to vary significantly in their key properties. This can be seen in the figures shown in the previous sections for both the static and dynamic schedules. For this reason, each of the 40 graphs was tested independently, with each frame representing a sample for the test, resulting in a sample size of 40 for each hypothesis test. An F-test was initially performed on a sample of graphs to identify if the samples have equal variances. The F-test indicated that at least some of the variances are unequal. Therefore, the t-test was performed assuming that all variances were unequal. Testing was performed with a significance of 0.05.

Since there is a large set of input variables, a large number of outputs are generated. In order to more efficiently analyze the outputs, MATLAB was used to automate the statistical analysis of the simulation results. The two-sample t-test for independent samples in MATLAB returns a binary result, indicating if the null hypothesis was rejected or not rejected at the specified level of significance. The results from MATLAB indicate that the Simulated Annealing clustering method produced a statistically significant runtime improvement in 498 out of 600 total cases or 83%. Meanwhile, the greedy method showed statistically significant runtime improvement in 417 out of 600 total cases or 69.5%. A sensitivity analysis was performed by re-

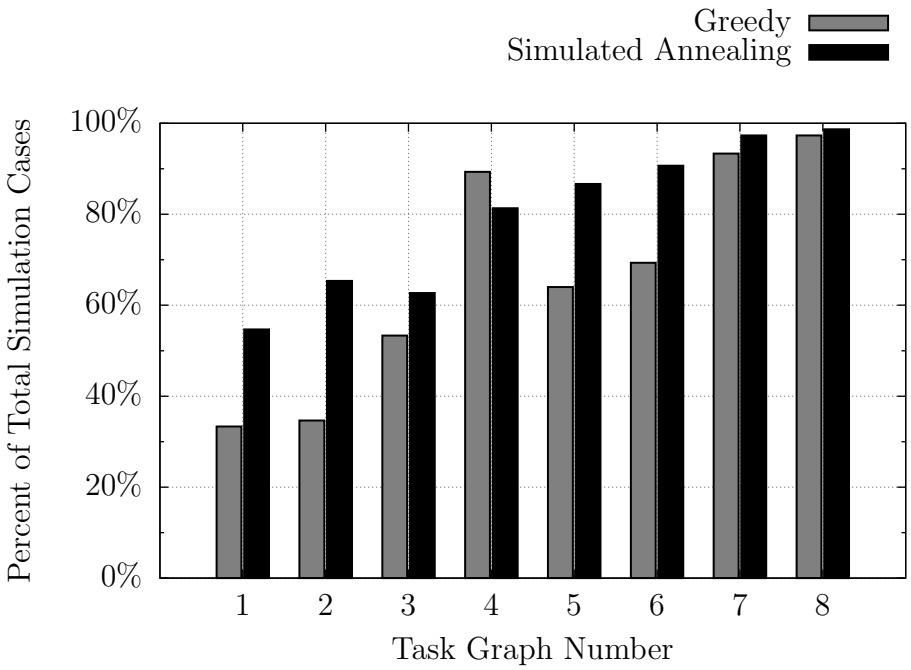


Figure 8.11: Statistical Significance of Clustering vs Graph Properties

peating the tests with a significance of 0.01. This resulted in the number of significant test case improvements being decreased to 478 and 401, respectively.

Figure 8.11 shows the statistical significance of grain packing as a function of the graph type. The lines in the figure plot the number of cases where runtime was found to be significantly lower after graph clustering than before graph clustering for each of the two algorithms. The horizontal line at the top shows the total number of test cases for each graph.

Figure 8.12 shows the results of graph clustering vs number of PEs.

Figure 8.13 shows the results of graph clustering vs the degree of graph nondeterminism. Graph nondeterminism is shown for 3 values, 0, 1 , & 2. A value of 0 indicates that all dependencies in the graph are executed every frame. A value of 1

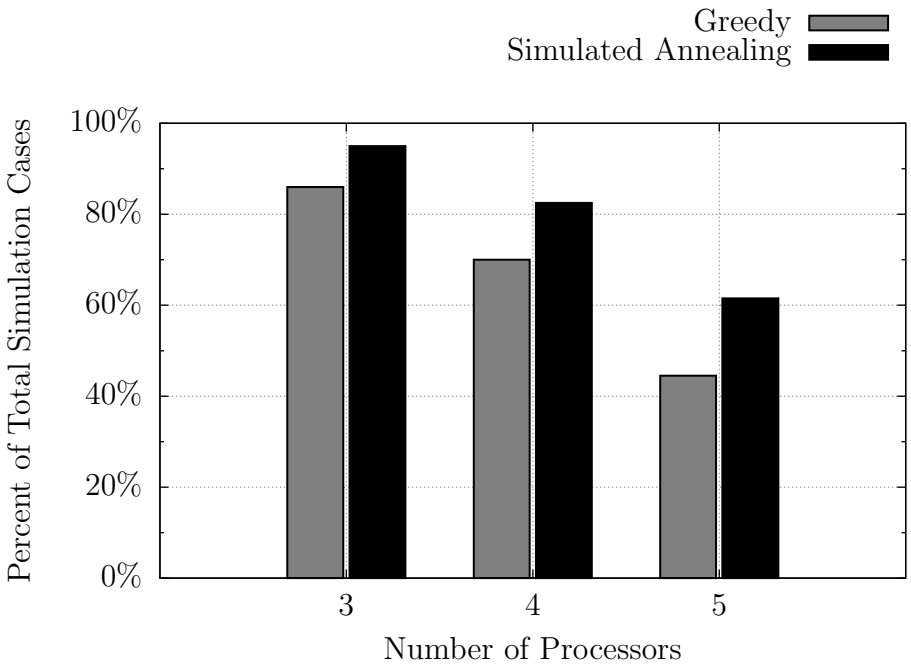


Figure 8.12: Statistical Significance of Clustering vs Number of PEs

or 2 indicates that some graph dependencies are conditionally executed, and are not present in some frames. The proportion of total edges is controlled by the value for graph nondeterminism as outlined previously.

Figure 8.14 shows the results of graph clustering vs the degree of task nondeterminism. This variable is computed at two levels, a value of 0 indicates that task execution times are deterministic, while a value of 1 indicates that task execution times are nondeterministic according to the rules outlined previously.

The plot of significant results vs graph type indicates that that graph clustering has the greatest effect on reducing runtime for highly parallel graphs. The simulation produced more significant results at higher parallelism. The parallelism generally increases as graph number from 1 to 8. Likely this results from having more tasks

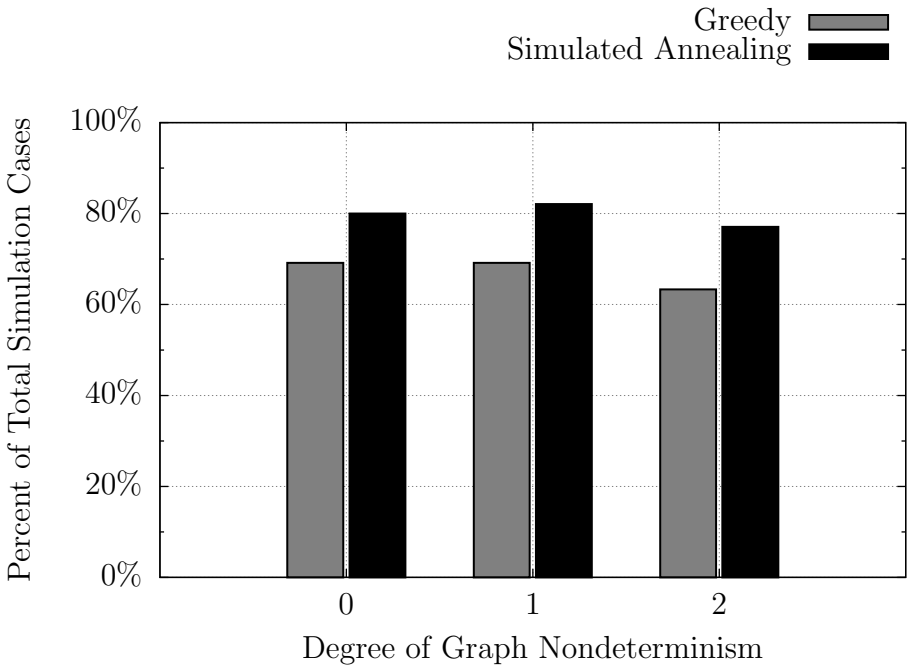


Figure 8.13: Statistical Significance of Clustering vs Graph Determinism

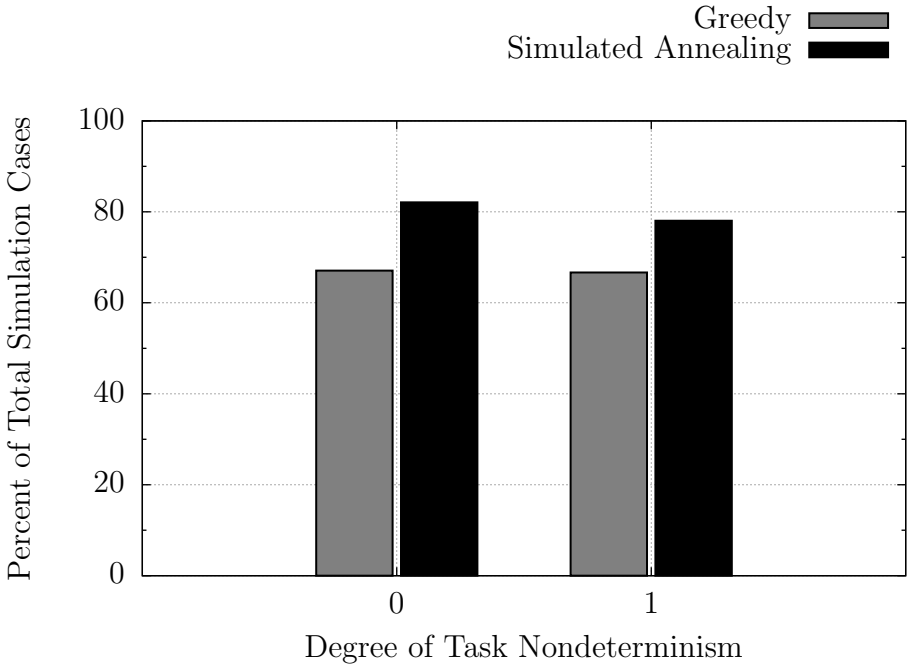


Figure 8.14: Statistical Significance of Clustering vs Task Runtime Determinism

available to cluster. Further, it would appear that the Simulated Annealing method produces more consistent results than the greedy algorithm. The SA algorithm is free to explore a wide variety of options, however, the greedy algorithm is constrained to the ordering produced by the static scheduler.

As can be seen in Figure 8.12, the effectiveness of both algorithms is negatively correlated with the number of available PEs. Likely this is not a result of poor performance of the clustering algorithms, but increased effectiveness of the dynamic scheduler under baseline conditions. As the number of PEs increase, the dynamic scheduler has more options to schedule tasks.

The remaining graphs would indicate that the results are not highly related to the nondeterminism of the graph. Since the fully deterministic cases are excluded, the number of test cases for the nondeterministic variables differs, as can be seen from the total line in each graph. For the nondeterministic task weight condition, there are one-third fewer test cases in the deterministic case vs the nondeterministic case. Likewise, the deterministic task graph case has only half of the number of test cases as are in the nondeterministic cases. For both algorithms, the line tends to stay proportional to the number of total graphs. In both cases, the largest change tends to follow the transition from fully deterministic to nondeterministic.

8.4 Speedup

As a final measure of effectiveness, the overall speedup of the graph execution will be examined. Table 8.6 and Table 8.7 show the average percent difference of speedup. To calculate the values in the table, the absolute speedup for a given

Table 8.6: SA Average Percent Speedup

		Graph							
ND Tasks	π	1	2	3	4	5	6	7	8
3 PEs									
0	$\pi = 1$	0.86	0.55	0.67	1.11	0.80	0.96	1.45	1.68
0	$\pi = 2$	1.07	0.37	0.49	1.29	0.84	1.01	1.40	1.64
1	$\pi = 0$	0.90	0.58	0.57	1.23	0.81	0.81	1.39	1.68
1	$\pi = 1$	0.79	0.55	0.70	1.18	0.72	0.98	1.42	1.64
1	$\pi = 2$	0.89	0.42	0.61	1.15	0.84	1.05	1.43	1.63
4 PEs									
0	$\pi = 1$	0.22	0.35	0.24	0.63	0.70	0.65	0.95	1.22
0	$\pi = 2$	0.16	0.25	0.12	0.61	0.61	0.66	1.02	1.13
1	$\pi = 0$	0.07	0.21	0.08	0.69	0.53	0.65	1.05	1.16
1	$\pi = 1$	0.19	0.23	0.21	0.65	0.53	0.76	1.03	1.15
1	$\pi = 2$	0.14	0.20	0.07	0.71	0.43	0.66	1.00	1.15
5 PEs									
0	$\pi = 1$	0.05	0.14	0.20	0.39	0.26	0.23	0.45	0.48
0	$\pi = 2$	0.00	0.14	0.16	0.26	0.38	0.27	0.31	0.56
1	$\pi = 0$	0.08	0.10	0.03	0.31	0.34	0.35	0.50	0.57
1	$\pi = 1$	0.00	0.15	0.13	0.29	0.41	0.35	0.35	0.52
1	$\pi = 2$	0.00	0.00	0.12	0.28	0.30	0.26	0.30	0.53

combination of the input variables was calculated. This value was then scaled by the baseline unclustered runtime. An average was calculated over all statistically significant speedups. Slowdowns were not considered in the calculation, nor were speedups that were not statistically significant.

The values in the table indicate that the Simulated Annealing algorithm outperforms the greedy algorithm in terms of speedup. Simulated Annealing is also more consistent in producing a significant improvement across the combination of

Table 8.7: Greedy Algorithm Average Percent Speedup

		Graph							
ND Tasks	π	1	2	3	4	5	6	7	8
3 PEs									
0	$\pi = 1$	0.14	0.35	0.51	1.01	0.93	0.67	1.05	1.18
0	$\pi = 2$	0.29	0.34	0.44	0.99	0.53	0.65	1.08	1.30
1	$\pi = 0$	0.09	0.20	0.53	1.01	0.67	0.50	1.03	1.15
1	$\pi = 1$	0.14	0.34	0.51	1.01	0.72	0.59	1.06	1.22
1	$\pi = 2$	0.39	0.17	0.46	0.91	0.55	0.58	1.09	1.27
4 PEs									
0	$\pi = 1$	0.21	0.01	0.18	0.50	0.34	0.34	0.69	0.70
0	$\pi = 2$	0.13	0.06	0.18	0.33	0.30	0.32	0.71	0.73
1	$\pi = 0$	0.03	0.06	0.10	0.39	0.31	0.27	0.79	0.77
1	$\pi = 1$	0.04	0.04	0.18	0.53	0.31	0.32	0.70	0.79
1	$\pi = 2$	0.04	0.04	0.19	0.47	0.30	0.24	0.72	0.79
5 PEs									
0	$\pi = 1$	0.00	0.10	0.17	0.13	0.35	0.03	0.35	0.37
0	$\pi = 2$	0.08	0.00	0.08	0.21	0.10	0.02	0.30	0.40
1	$\pi = 0$	0.00	0.02	0.01	0.20	0.07	0.07	0.30	0.41
1	$\pi = 1$	0.00	0.03	0.09	0.18	0.08	0.03	0.29	0.39
1	$\pi = 2$	0.09	0.00	0.07	0.21	0.07	0.07	0.23	0.38

task graphs examined. However, this comes at a cost; The runtime of this algorithm is significantly longer than the runtime of the greedy algorithm.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

From the data presented in the previous chapter, graph clustering is a promising method to improve the runtime of nondeterministic applications executing on reconfigurable platforms. The overall effectiveness in excess of 60% for both algorithms indicate that graph clustering would produce a runtime improvement in most nondeterministic applications.

9.1 Conclusions

In summary, the work presented in this dissertation discussed a task agglomeration approach for reconfigurable computers typified by two algorithms. One of the algorithms used a complex heuristic optimization technique, while the other used a simple greedy list based grouping mechanism. These algorithms were applied to a sophisticated dynamic system simulation that has been validated by previous researchers to model nondeterministic applications exhibiting control and data-driven nondeterministic behavior.

Additionally, a real-world example reconfigurable system was developed to evaluate model assumptions that were the basis of the agglomeration methods de-

veloped. This system demonstrated that the reconfiguration time of a partially reconfigurable device partition is proportional to the amount of resources contained in that partition. Likewise, the amount of resources is a 1-dimensional function of the partition's size.

Finally, the task agglomeration methods were applied across a wide selection of simulated application and platform parameters using the dynamic system simulation. The results obtained indicate that the task agglomeration method is effective at reducing application runtime for nondeterministic applications in partially reconfigurable systems. This runtime improvement is principally attributable to a reduction in the number of required reconfigurations due to the agglomeration method.

9.2 Future Work

Although this research found good results in applying grain packing to a wide variety of task systems, there are several areas which could be examined for further improvements. Firstly, the examination was limited to homogeneous reconfigurable architectures with identically sized processing elements. Many realizable reconfigurable architectures are not constrained by homogeneity. While this limitation was imposed to maintain relationships with previous work, the grain packing technique should also be examined in terms of heterogeneous architectures. This includes both hard and soft microprocessors that may improve the performance of some graphs.

The constraint of limited simultaneous reconfigurations should also be examined in future efforts. Again, this constraint was not considered so that parallels could be drawn with previous research. However, practical devices only allow a lim-

ited number of such reconfigurations. Scheduling multiple reconfigurations around a key resource would impose a significant penalty when implemented on a real-world device.

Previous work also identified that the dynamic scheduler used in this research is outperformed by the static scheduler under most cases. Future work could investigate a method for integrating the knowledge of the static scheduler to provide scheduling improvements to the dynamic scheduler. Included as one possible improvement to the dynamic scheduler is the concept of configuration locking based on knowledge of the static schedule. This would prevent a commonly used task type from being reconfigured in favor of a less commonly used task.

REFERENCES

[1] G. Estrin and C. R. Viswanathan, "Organization of a "Fixed-Plus-Variable" Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 41–60, Jan. 1962.

[2] S. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.

[3] P. Garcia, K. Compton *et al.*, "An Overview of Reconfigurable Hardware in Embedded Systems," *EURASIP Journal on Embedded Systems*, vol. 2006, no. 1, pp. 1–19, Jan. 2006.

[4] W. Najjar and J. Villarreal, "Reconfigurable Computing in the New Age of Parallelism," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, K. Bertels, N. Dimopoulos *et al.*, Eds. Springer Berlin Heidelberg, 2009, no. 5657, pp. 255–262.

[5] L. Liu, "A Hardware and Software Cooperative Design of SoC IP," in *Proceedings of the 2010 International Conference on Computing, Control and Industrial Engineering (CCIE)*, vol. 1, Jun. 2010, pp. 77–80.

[6] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.

[7] M. Pellauer, A. Parashar *et al.*, "Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures," *ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 10:1–10:32, Sep. 2015.

[8] W. A. Najjar and P. Ienne, "Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 4–6, Jan. 2014.

[9] P. J. Denning, "Virtual Memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, Sep. 1970.

[10] D. Bertozzi, G. Dimitrakopoulos *et al.*, "The Fast Evolving Landscape of On-Chip Communication," *Design Automation for Embedded Systems*, vol. 19, no. 1, pp. 59–76, Apr. 2014.

[11] T. Marescaux, J.-Y. Mignolet *et al.*, "Networks on Chip as Hardware Components of an OS for Reconfigurable Systems," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, P. Y. K. Cheung and G. A. Constantinides, Eds. Springer Berlin Heidelberg, Sep. 2003, no. 2778, pp. 595–605.

[12] R. Tessier, K. Pocek *et al.*, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, Mar. 2015.

[13] C. H. Roth, Jr, *Fundamentals of Logic Design*, 4th ed. Boston: Pws Pub Co, Jul. 1995.

[14] Xilinx, "7 Series FPGAs Configuration User Guide (UG470)," Aug. 2016.

[15] S. Corbetta, M. Morandi *et al.*, "Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650–1654, Nov. 2009.

[16] S. A. Guccione and D. Levi, "JBits: A Java-Based Interface to FPGA Hardware," in *Proceedings of the 1999 Military and Aerospace Programmable Logic Device International Conference*, 1999.

[17] T. D. Nguyen and A. Kumar, "PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 149–158.

[18] H. Walder and M. Platzner, "Online Scheduling for Block-Partitioned Reconfigurable Devices," in *Proceedings of the Conference on Design, Automation and Test in Europe Conference and Exhibition*, ser. DATE '03. Munich: IEEE Computer Society, Mar. 2003.

[19] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[20] D. Lim and M. Peattie, "An Implementation Flow for Active Partial Reconfiguration," May 2002.

[21] T. Marconi, "Online Scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays," *Computers & Electrical Engineering*, vol. 40, no. 4, pp. 1215–1237, May 2014.

[22] P. Sedcole, B. Blodget *et al.*, "Modular partial reconfigurable in Virtex FPGAs," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 3, pp. 157–164, Aug. 2005.

[23] A. A. Sohanghpurwala, P. Athanas *et al.*, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs." Shanghai: IEEE, 16-20 May 2011, pp. 228–235.

[24] F. Dittmann and S. Frank, "Hard Real-Time Reconfiguration Port Scheduling," in *Proceedings of the 2007 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 16-20 April 2007, pp. 1–6.

[25] Z. Pan and B. Wells, "Hardware Supported Task Scheduling on Dynamically Reconfigurable SOC Architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1465 –1474, Nov. 2008.

[26] J. A. Clemente, C. González *et al.*, "A task graph execution manager for reconfigurable multi-tasking systems," *Microprocessors and Microsystems*, vol. 34, no. 2-4, pp. 73–83, Mar. 2010.

[27] Xilinx, "Partial Reconfiguration User Guide v14.3," Oct. 2012.

[28] T. Becker, W. Luk *et al.*, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," in *Proceedings of the 2007 IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 2007, pp. 35 – 44.

[29] P. Sedcole, B. Blodget *et al.*, "Modular dynamic reconfiguration in Virtex FPGAs," *Computers and Digital Techniques, IEE Proceedings* -, vol. 153, no. 3, pp. 157–164, May 2006.

[30] W. Wolf, "Hardware-software co-design of embedded systems [and prolog]," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, Jul. 1994.

[31] G. De Micheli and R. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[32] R. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design Test of Computers*, vol. 10, no. 3, pp. 29–41, Sep. 1993.

[33] R. Ernst, J. Henkel *et al.*, "Hardware-software cosynthesis for microcontrollers," *IEEE Design Test of Computers*, vol. 10, no. 4, pp. 64–75, Dec. 1993.

[34] J. Noguera and R. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, Aug. 2002.

[35] K. S. Chatha and R. Vemuri, "Hardware-Software Codesign for Dynamically Reconfigurable Architectures," in *Field Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, P. Lysaght, J. Irvine *et al.*, Eds. Springer Berlin Heidelberg, 1999, no. 1673, pp. 175–184.

[36] T. Wiangtong, P. Y. Cheung *et al.*, "A unified codesign run-time environment for the ULTRASONIC reconfigurable computer," in *Field Programmable Logic and Application*. Springer, 2003, pp. 396–405.

[37] M. Chiodo, P. Giusto *et al.*, "Hardware-software codesign of embedded systems," *Micro, IEEE*, vol. 14, no. 4, pp. 26–36, 1994.

[38] D. Gajski and F. Vahid, "Specification and design of embedded hardware-software systems," *IEEE Design Test of Computers*, vol. 12, no. 1, pp. 53–67, Spring 1995.

[39] J. Keinert, M. Streubühr *et al.*, "SystemCoDesigner—an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications," *ACM Transactions on Design of Automated Electronic Systems*, vol. 14, no. 1, pp. 1:1–1:23, Jan. 2009.

[40] T. Wiangtong, P. Cheung *et al.*, "Multitasking in hardware-software codesign for reconfigurable computer," in *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03*, vol. 5, May 2003, pp. V–621–V–624 vol.5.

[41] V. Carchiolo, M. Malgeri *et al.*, "Hardware/Software Synthesis of Formal Specifications in Codesign of Embedded Systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 399–432, Jul. 2000.

[42] P. R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*. Springer Science & Business Media, Sep. 2010.

[43] Z. Á. Mann, A. Orbán *et al.*, "Finding optimal hardware/software partitions," *Formal Methods in System Design*, vol. 31, no. 3, pp. 241–263, Oct. 2007.

[44] L. Huang, F. Yuan *et al.*, "On Task Allocation and Scheduling for Lifetime Extension of Platform-Based MPSoC Designs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 2088–2099, 2011.

[45] R. Cordone, F. Redaelli *et al.*, "Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, May 2009.

[46] T. Wiangtong, P. Y. K. Cheung *et al.*, "Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware Software Codesign," *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–449, Jul. 2002.

[47] J. Harkin, T. M. McGinnity *et al.*, "Genetic algorithm driven hardware–software partitioning for dynamically reconfigurable embedded systems," *Microprocessors and Microsystems*, vol. 25, no. 5, pp. 263–274, Aug. 2001.

[48] A. B. Hadj-Alouane, J. C. Bean *et al.*, "A hybrid genetic/optimization algorithm for a task allocation problem," *Journal of Scheduling*, vol. 2, no. 4, pp. 189–201, Jul. 1999.

[49] J. Clemente, V. Rana *et al.*, "A Hybrid Mapping-Scheduling Technique for Dynamically Reconfigurable Hardware," Sep. 2011, pp. 177 –180.

[50] K. B. Chehida and M. Auguin, "HW / SW Partitioning Approach for Reconfigurable System Design," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 247–251.

[51] P. Liu, J. Wu *et al.*, "Hybrid algorithms for hardware/software partitioning and scheduling on reconfigurable devices," *Mathematical and Computer Modelling*, vol. 58, no. 1–2, pp. 409–420, Jul. 2013.

[52] K. Bertels, V.-M. Sima *et al.*, "HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms," *IEEE Micro*, vol. 30, no. 5, pp. 88–97, Sep. 2010.

[53] X. S. Hu and J. G. D'Ambrosio, "Hardware-Software Partitioning for Real-Time Embedded Systems," *Design Automation for Embedded Systems*, vol. 2, no. 3-4, pp. 339–358, May 1997.

[54] H. Topcuoglu, S. Hariri *et al.*, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, 2002.

[55] M. A. Khan, "Scheduling for heterogeneous Systems using constrained critical paths," *Parallel Computing*, vol. 38, pp. 175–193, 2012/4/1/.

[56] B. K. Hamilton, M. Inggs *et al.*, "Scheduling Mixed-Architecture Processes in Tightly Coupled FPGA-CPU Reconfigurable Computers," in *Proceedings of the 2014 22nd Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 11-13 May 2014, pp. 240–240.

[57] W. Hardt and W. Rosenstiel, "Prototyping of Tightly Coupled Hardware/Software-Systems," *Design Automation for Embedded Systems*, vol. 2, no. 3-4, pp. 283–317, May 1997.

[58] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, Apr. 2003.

[59] C. Pilato, A. Cazzaniga *et al.*, "On the automatic integration of hardware accelerators into FPGA-based embedded systems," in *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL),*, 2012, pp. 607–610.

[60] N. Tredennick and B. Shimamoto, "The inevitability of reconfigurable systems," *Queue*, vol. 1, no. 7, p. 34, 2003.

[61] ——, "Prospects for Reconfigurable Systems," *IEEE Micro*, vol. 34, no. 1, pp. 72–78, Jan. 2014.

[62] D. Andrews, D. Niehaus *et al.*, "Programming models for hybrid CPU/FPGA chips," *Computer*, vol. 37, no. 1, pp. 118–120, Jan. 2004.

[63] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 1997 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines,*. IEEE, 1997, pp. 12–21.

[64] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proceedings of the 1996 IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 1996, pp. 126–135.

[65] C. Chang, J. Wawrzynek *et al.*, "BEE2: A high-end reconfigurable computing system," *IEEE Design Test of Computers*, vol. 22, no. 2, pp. 114–125, Mar. 2005.

[66] S. D. Haynes, H. G. Epsom *et al.*, "UltraSONIC: A Reconfigurable Architecture for Video Image Processing," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ser. Lecture Notes in Computer Science, M. Glesner, P. Zipf *et al.*, Eds. Springer Berlin Heidelberg, Sep. 2002, no. 2438, pp. 482–491.

[67] M. Liu, W. Kuehn *et al.*, "Run-time Partial Reconfiguration speed investigation and architectural design space exploration," in *Proceedings of the International Conference on Field Programmable Logic and Applications, FPL 2009*, Aug. 2009, pp. 498–502.

[68] M. French, E. Anderson *et al.*, "Autonomous system on a chip adaptation through partial runtime reconfiguration," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2008, pp. 77–86.

[69] H. Singh, M.-H. Lee *et al.*, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[70] S. Hauck, T. Fry *et al.*, "The Chimaera reconfigurable functional unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 206–217, Feb. 2004.

[71] A. Agne, M. Happe *et al.*, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan. 2014.

[72] S. Bayar and A. Yurdakul, "A dynamically reconfigurable communication architecture for multicore embedded systems," *Journal of Systems Architecture*, vol. 58, no. 3–4, pp. 140–159, Mar. 2012.

[73] T. Pionteck, R. Koch *et al.*, "Applying partial reconfiguration to networks-on-chips," in *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[74] M. Koester, W. Luk *et al.*, "Design Optimizations for Tiled Partially Reconfigurable Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1048–1061, Jun. 2011.

[75] M. Abdelfattah and V. Betz, "The Case for Embedded Networks on Chip on Field-Programmable Gate Arrays," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, Jan. 2014.

[76] H. Walder and M. Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner *et al.*, Eds. Springer Berlin Heidelberg, 2004, no. 3203, pp. 831–835.

[77] P. C. Santos, G. L. Nazar *et al.*, "A fully dynamic reconfigurable noc-based mpsoc: The advantages of total reconfiguration," Berlin, 2013, pp. 232–237.

[78] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[79] M. H. Neishaburi and Z. Zilic, "A Fault Tolerant Hierarchical Network on Chip Router Architecture," *Journal of Electronic Testing*, vol. 29, no. 4, pp. 485–497, Aug. 2013.

[80] D. U. Becker, "Efficient microarchitecture for network-on-chip routers," Ph.D. dissertation, Stanford University, 2012.

[81] R. Hesse and N. E. Jerger, "Hierarchical Clustering for On-Chip Networks," in *Proceedings of the 1st International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*, ser. AISTECS '16. New York, NY, USA: ACM, 2016, pp. 2:1–2:6.

[82] Z. Li, "Configuration management techniques for reconfigurable computing," Ph.D. dissertation, Citeseer, 2002.

[83] Z. Li, K. Compton *et al.*, "Configuration caching management techniques for reconfigurable computing," in *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 22–36.

[84] S. Sudhir, S. Nath *et al.*, "Configuration caching and swapping," in *Field-Programmable Logic and Applications*. Springer, 2001, pp. 192–202.

[85] K. Papademetriou and A. Dollas, "Performance evaluation of a preloading model in dynamically reconfigurable processors," in *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 1–4.

[86] R. Kalra and R. Lysecky, "Configuration Locking and Schedulability Estimation for Reduced Reconfiguration Overheads of Reconfigurable Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 4, pp. 671–674, 2010.

[87] S. Narayanan, D. Chillet *et al.*, "Hardware OS Communication Service and Dynamic Memory Management for RSoCs," in *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 30 -Dec 2 2011, pp. 117 –122.

[88] J. Resano and D. Mozos, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the 41st Annual Design Automation Conference*, San Diego, CA, USA, 2004, pp. 119–124.

[89] K. Danne and M. Platzner, "An EDF schedulability test for periodic tasks on reconfigurable hardware devices," in *ACM SIGPLAN Notices*, vol. 41, 2006, pp. 93–102.

[90] S. Loo and B. Wells, "Task scheduling in a finite-resource, reconfigurable hardware/software codesign environment," *INFORMS Journal on Computing*, vol. 18, no. 2, pp. 151–172, 2006.

[91] D. Andrews, "Operating Systems Research for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 54–58, Jan. 2014.

[92] F. Say and C. F. Bazlamaçcı, "A reconfigurable computing platform for real time embedded applications," *Microprocessors and Microsystems*, vol. 36, no. 1, pp. 13–32, Feb. 2012.

[93] C. Jing, Y. Zhu *et al.*, "Energy-efficient scheduling on multi-FPGA reconfigurable systems," *Microprocessors and Microsystems*, May 2013.

[94] T. L. Adam, K. M. Chandy *et al.*, "A Comparison of List Schedules for Parallel Processing Systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[95] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 399–409, Apr. 2008.

[96] J. A. Clemente, I. Beretta *et al.*, "A Mapping-Scheduling Algorithm for Hardware Acceleration on Reconfigurable Platforms," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 9:1–9:27, Jul. 2014.

[97] G. Durelli, C. Pilato *et al.*, "Automatic run-time manager generation for reconfigurable MPSoC architectures," in *2012 7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip*, 2012, pp. 1–8.

[98] A. Al-Wattar, S. Areibi *et al.*, "An Efficient Evolutionary Task Scheduling/Binding Framework for Reconfigurable Systems," *International Journal of Reconfigurable Computing*, vol. 2016, Mar. 2016.

[99] S. J. Olakkenghil and K. Baskaran, "An FPGA Task Placement Algorithm Using Reflected Binary Gray Space Filling Curve," *International Journal of Reconfigurable Computing*, vol. 2014, p. e495080, Apr. 2014.

[100] J. Resano, D. Mozos *et al.*, "Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, P. Y. K. Cheung and G. A. Constantinides, Eds. Springer, 2003, no. 2778, pp. 585–594.

[101] X. Iturbe, K. Benkrid *et al.*, "Runtime Scheduling, Allocation, and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence," *International Journal of Reconfigurable Computing*, vol. 2013, p. e905057, Mar. 2013.

[102] J.-Y. Mignolet, V. Nollet *et al.*, "Enabling Run-time Task Relocation on Reconfigurable Systems," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, 2005, pp. 69–80.

[103] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 686–701, Jun. 1993.

[104] M. Palis, J. Liou *et al.*, "A greedy task clustering heuristic that is provably good," in *Proceedings of the 1994 International Symposium on Parallel Architectures, Algorithms and Networks*, 1994, pp. 398–405.

[105] T. Wiangtong, P. Cheung *et al.*, "Cluster-Driven Hardware/Software Partitioning and Scheduling Approach for a Reconfigurable Computer System," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, P. Y. K. Cheung and G. Constantinides, Eds. Springer Berlin / Heidelberg, 2003, vol. 2778, pp. 1071–1074, 10.1007/978-3-540-45234-8_122.

[106] I. T. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, 1st ed. Reading, Mass: Pearson, Feb. 1995.

[107] M. C. V. Nascimento, "Community detection in networks via a spectral heuristic based on the clustering coefficient," *Discrete Applied Mathematics*, vol. 176, pp. 89–99, Oct. 2014.

[108] G. Karypis, R. Aggarwal *et al.*, "Multilevel hypergraph partitioning: Applications in VLSI domain," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 1, pp. 69–79, 1999.

[109] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[110] P. M. Gleiser and L. Danon, "Community structure in jazz," *Advances in complex systems*, vol. 6, no. 04, pp. 565–573, 2003.

[111] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, Feb. 2004.

[112] M. Gaertler, R. Görke *et al.*, "Significance-Driven Graph Clustering," in *Algorithmic Aspects in Information and Management*, ser. Lecture Notes in Computer Science, M.-Y. Kao and X.-Y. Li, Eds. Springer Berlin Heidelberg, Jan. 2007, no. 4508, pp. 11–26.

[113] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[114] U. Brandes, D. Delling *et al.*, "Maximizing Modularity is hard," *arXiv:physics/0608255*, Aug. 2006.

[115] B. Ouni, R. Ayadi *et al.*, "Temporal partitioning of data flow graph for dynamically reconfigurable architecture," *Journal of Systems Architecture*, vol. 57, no. 8, pp. 790–798, Sep. 2011.

[116] J. Cardoso, "On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1362–1375, Oct. 2003.

[117] F. Fangfa, B. Yuxin *et al.*, "An Objective-Flexible Clustering Algorithm for task mapping and scheduling on cluster-based NoC," in *2010 Academic Symposium on Optoelectronics and Microelectronics Technology and 10th Chinese-Russian Symp. Laser Physics and Laser Technology*, Harbin,China, 2010, pp. 369–373.

[118] S. Ostadzadeh, R. Meeuws *et al.*, "A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems," in *Proceedings of the 2009 International Conference on Complex, Intelligent and Software Intensive Systems.*, 2009, pp. 663 –668.

[119] I. Beretta, V. Rana *et al.*, "Run-time mapping of applications on FPGA-based reconfigurable systems," in *Proc. 2010 IEEE International Symp. Circuits and Systems (ISCAS)*, May 30-June 2 2010, pp. 3329 –3332.

[120] ——, "A Mapping Flow for Dynamically Reconfigurable Multi-Core System-on-Chip Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1211 –1224, Aug. 2011.

[121] ——, "Island-Based Adaptable Embedded System Design," *Embedded Systems Letters, IEEE*, vol. 3, no. 2, pp. 53 –57, Jun. 2011.

[122] Z. Chen, M. Qiu *et al.*, "Clustering scheduling for hardware tasks in reconfigurable computing systems," *Journal of Systems Architecture*, May 2013.

[123] S. Koh and O. Diessel, "Configuration Merging in Point-to-Point Networks for Module-Based FPGA Reconfiguration," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pp. 4:1–4:36, Jan. 2010.

[124] D. Koch, C. Beckhoff *et al.*, "Fine-Grained Partial Runtime Reconfiguration on Virtex-5 FPGAs," in *2010 18th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 69–72.

[125] Z. Pan, *Hardware Supported Task Scheduling on Dynamically Reconfigurable SOC Architectures: A Dissertation*, 2006, 2006.

[126] J. Noguera and R. M. Badia, "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 2, pp. 385–406, May 2004.

[127] A. Corporation, "Quartus II Handbook," May 2015.

[128] I. Terasic Technologies, "DE5-Net FPGA Development Kit User Manual," 2012.

[129] A. Corporation, "Partial Reconfiguration IP Core User Guide," Jan. 2015.

[130] ——, "Design Planning for Partial Reconfiguration," May 2013.

[131] Z. Cvetanovic, "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 421–432, 1987.

[132] M. Palis, J.-C. Liou *et al.*, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46–55, 1996.

[133] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, Jun. 1999.

[134] K. Papadimitriou, A. Anyfantis *et al.*, "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 59, no. 6, pp. 1642–1651, 2010.

[135] T. D. Braun, H. J. Siegel *et al.*, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[136] F. Hillier and G. Lieberman, *Introduction to Operations Research*, 8th ed. McGraw-Hill Science/Engineering/Math, Aug. 2005.

[137] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Mass.: Addison-Wesley Pub. Co., 1989.

[138] J. Kennedy and R. Eberhart, "A discrete binary version of the particle swarm algorithm," in *Proceedings of the 1997 IEEE International Conference on Systems, Man, and Cybernetics Computational Cybernetics and Simulation*, vol. 5, Oct. 1997, pp. 4104–4108 vol.5.

[139] R. P. Dick, D. L. Rhodes *et al.*, "TGFF: Task graphs for free," in *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, ser. CODES/CASHE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 97–101.

[140] D. Austin and B. E. Wells, "Heuristically Driven Task Agglomeration in Limited Resource Partially-Reconfigurable Systems," in *Proceedings of the 2013 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, T. Plaks, Ed., Las Vegas, Nevada, 22-25 July 2013, pp. 29–35.

[141] D. C. Montgomery, *Design and Analysis of Experiments*, 6th ed. Wiley, Dec. 2004.

[142] A. J. Booker, "Design and analysis of computer experiments," in *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, St. Louis, 1998, p. 118.

[143] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures: Fourth Edition*, 4th ed. Boca Raton: Chapman and Hall/CRC, Jan. 2007.