# Generating the Communications Infrastructure for Module-based Dynamic Reconfiguration of FPGAs

**Shannon Koh**

Bachelor of Computer Science (Honours), UNSW Sydney, 2003

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

**Doctor of Philosophy**

School of Computer Science and Engineering

THE UNIVERSITY OF
NEW SOUTH WALES

SCIENTIA

MANU ET MENTE

SYDNEY · AUSTRALIA

February 2008

# Originality Statement

# Acknowledgements

I would like to thank my supervisor, Dr. Oliver Diessel, for his unwavering support in this project. His supervision was exemplary, he made the entire experience of getting a PhD rich and full, and he encouraged me to think critically in ways I would not have previously imagined. I would also like to thank my co-supervisor Prof. Sri Parameswaran for his excellent insights. I would like to thank my wife, Molly Hu, for all her support and understanding throughout the pursuit of my degree, especially at the most critical moments. I would also like to thank my mother who encouraged me to pursue my PhD and supported me throughout. I would like to thank all of my fellow PhD students on the 5th floor in the Architecture Group, especially Jorgen Peddersen, who, not only being the best friend one might have, also encouraged me to think. The rest of my fellow students Jeremy Chan, Krutartha Patel, Anjelo Ambrose, Carol He, Michael Chong all made my research experience in UNSW Sydney the best anyone could have. Last but not least I would like to thank the Australian Government for the Australian Postgraduate Award, the School of Computer Science and Engineering for being an excellent academic institution, and National ICT Australia for supporting me through the NICTA research scholarships throughout and especially for the last six months.

## Abstract

Current approaches to supporting module-based FPGA reconfiguration focus on various aspects and sub-problems in the area but do not combine to form a coherent, top-down methodology that factors low-level device parameters into every step of the design flow. This thesis proposes such a top-down methodology from application specification to low-level implementation, centered around examining the problem of generating a point-to-point communications infrastructure to support the changing interfaces of dynamically placed modules. Low-level implementation parameters are considered at every stage to ensure that area, timing and budget constraints of the application are met. The approach advocates the regular layout of modules surrounded by a wiring harness supporting the communications for those modules, and thus provides an advanced understanding of how to implement the "fixed wiring harness" model of reconfigurable computing proposed by Brebner. Results have shown that compared to flattened netlists the regularity of the layout does not impose significant overheads on critical path delays. At high communication densities it can even result in lower delays. The core of the methodology is an infrastructure generation process that allocates modules to slots and merges configuration graphs to form wiring harnesses that support the communications for these merged configurations. This thesis suggests methods and evaluates algorithms for configuration graph merging so as to reduce run-time reconfiguration overheads. Initial experiments with a greedy merging algorithm performed on an optical flow application resulted in a substantial reduction of 64% in reconfiguration time. The effects of graph merging with the initial greedy algorithm and an improved dynamic programming algorithm were explored for a range of device sizes and architectural parameters. Results show that configuration merging using the greedy method results in significant reductions to the reconfiguration delay. The dynamic programming algorithm provides consistent improvements above and beyond the savings provided by the greedy method. In addition, a strong correlation was identified between the quality of front-end design activities such as partitioning and the effectiveness of back-end implementations. The methodology is integrated into the Xilinx commercial tool flow for partial reconfiguration, and is effective for implementing applications for module-based FPGA reconfiguration where the modules and their communications requirements are known at design time. It also allows a system designer to consider alternate device sizes and parameters until a set is found that satisfies the application constraints.

# List of Publications

1. Koh, S. and Diessel, O. COMMA: a communications methodology for dynamic module-based reconfiguration of FPGAs. *In International Conference on Architecture of Computing Systems, Dynamically Reconfigurable Systems Workshop Proceedings*, pages 173–182, Frankfurt, Germany, 2006. Gesellschaft für Informatik, Bonn, Germany.

2. Koh, S. and Diessel, O. COMMA: a communications methodology for dynamic module reconfiguration in FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 273–274, Napa Valley, California, 2006. IEEE.

3. Koh, S. and Diessel, O. Communications infrastructure generation for modular FPGA reconfiguration. In *IEEE International Conference on Field Programmable Technology*, pages 321–324, Bangkok, Thailand, 2006. IEEE.

4. Koh, S. and Diessel, O. Module graph merging and placement to reduce reconfiguration overheads in paged FPGA devices. In *International Conference on Field Programmable Logic and Applications*, pages 293–298, Amsterdam, The Netherlands, 2007. IEEE.

5. Koh, S. and Diessel, O. The effectiveness of configuration merging in point-to-point networks for module-based FPGA reconfiguration. Accepted for presentation at the *IEEE Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, California, 2008. IEEE.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AMBA** | Advanced Microcontroller Bus Architecture |
| **ASIC** | Application-Specific Integrated Circuit |
| **ASIP** | Application-Specific Instruction-set Processor |
| **BRAM** | Block Random Access Memory |
| **CCC** | Chip and Communications Configuration |
| **CDFG** | Control Data Flow Graph |
| **CLB** | Configurable Logic Block |
| **CPU** | Central Processing Unit |
| **CTG** | Conditional Task Graph |
| **DAG** | Directed Acyclic Graph |
| **DCM** | Digital Clock Manager |
| **DCT** | Discrete Cosine Transform |
| **DES** | Data Encryption Standard |
| **DSP** | Digital Signal Processing |
| **DyNoC** | Dynamic Network-on-Chip |
| **EAPR** | Early-Access Partial Reconfiguration |
| **EDIF** | Electronic Design Interchange Format |
| **ESM** | Erlangen Slot Machine |
| **FIFO** | First-In First-Out |
| **FPGA** | Field-Programmable Gate Array |
| **GRM** | General Routing Matrix |
| **HDL** | Hardware Description Language |
| **ICAP** | Internal Configuration Access Port |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **ILP** | Integer Linear Program |
| **I/O** | Input/Output |
| **IOB** | Input-Output Block |

| | |
|---|---|
| **ISE** | Integrated Synthesis Environment |
| **JPEG** | Joint Photographic Experts Group |
| **JTAG** | Joint Test Action Group |
| **JTRS** | Joint Tactical Radio System |
| **LUT** | Look-Up Table |
| **MIC** | Module-Infrastructure Connector |
| **MPEG** | Moving Picture Experts Group |
| **NCD** | Native Circuit Description |
| **NoC** | Network-on-Chip |
| **NP** | Non-deterministic Polynomial time |
| **PAR** | Place-and-Route |
| **PCB** | Printed Circuit Board |
| **PIP** | Programmable Interconnect Point |
| **PRM** | Partially Reconfigurable Module |
| **PRR** | Partially Reconfigurable Region |
| **RAM** | Random Access Memory |
| **RCM** | Reconfiguration Control Module |
| **RDP** | Reconfigurable Data Port |
| **RMB** | Reconfigurable Multiple Bus |
| **RPM** | Relatively Placed Macro |
| **SDR** | Software Defined Radio |
| **SLU** | Swappable Logic Unit |
| **SRAM** | Static Random Access Memory |
| **TRACE** | Timing Report And Circuit Evaluator |
| **VHDL** | Very-high-speed integrated circuit Hardware Description Language |
| **XAPP290** | Xilinx Application Note 290 |
| **XDL** | Xilinx Design Language |
| **XOR** | Exclusive-OR |

# Chapter 1

# Introduction

## 1.1 Background and Research Context

A Field-Programmable Gate Array (FPGA) is a silicon device that can be programmed and reprogrammed any number of times to behave like different application-specific circuits. An FPGA is primarily composed of an array of logic cells, and wiring interconnecting these cells. Each logic cell implements a very small logic function, and the wiring between the cells is routed by means of switch settings which direct the flow of data through specific wiring tracks.Programming the FPGA involves changing the contents of memory cells (the *configuration memory*) that configure the logic and the wiring on the device.

*Configuring* an FPGA refers to loading a configuration bitstream, i.e., the actual data to be loaded into configuration memory, onto the device. FPGA design tools start by synthesizing an application circuit specified in a design language such as VHDL [36] or Verilog [37] into a *netlist*, then mapping the netlist to a particular FPGA, thereby creating a *circuit description*, and finally generating the configuration bitstream for this circuit description. Once the bitstream is loaded into the configuration memory of the FPGA, it behaves like the application circuit description that was synthesized at the start of the design flow. Thereafter, loading a different bitstream to change the circuit on the device is known as a *reconfiguration*.

Newer FPGAs such as the Xilinx Virtex-4 family [87] can not only be reprogrammed multiple times, but can also be reprogrammed while the device is running. To differentiate this from reconfiguring the device while the user logic clocks are disabled, this is commonly known as *dynamic reconfiguration* or *runtime reconfiguration*. Furthermore, such FPGAs are also partially reprogrammable, i.e., only a portion of the device is reprogrammed while the rest of the configuration memory remains unchanged. Partially reprogramming an FPGA requires less configuration data, and in effect takes up less time to do so. This is termed *partial reconfiguration*. Finally, these FPGAs can accordingly be partially reprogrammed at run time, where part of the FPGA is reprogrammed while other parts continue running. All three terms: *dynamic reconfiguration*, *runtime reconfiguration* and *partial reconfiguration* are used in the literature to express this concept, with *runtime partial reconfiguration* or *partial dynamic reconfiguration* being commonly-accepted variants.

FPGAs are most commonly included as components of embedded systems, in which performance requirements demand hardware support, yet cost, time-to-market pressures, the need to provide alternative hardware components, or the need to revise hardware-based components over time preclude the use of Application-Specific Integrated Circuits (ASICs), Application-Specific Instruction-Set Processors (ASIPs), or general-purpose processors alone. More often than not, that part of the system design that is targeted at the FPGA is static, or doesn't change while the system is operating.

The ability to partially reconfigure FPGAs at run time opens up many opportunities to enhance systems using these FPGAs. To date the most prevalent use of dynamic reconfiguration is to support so-called hardware virtualization, of which several flavors can be identified. In order to make do with insufficient FPGA area, a large FPGA circuit might be temporally partitioned into components that are swapped over time in order to map a large circuit into a smaller device [81]. When the computation can be temporally partitioned, the decision to virtualize may be made without the need to do so. Instead of mapping the design to a large device, a smaller one can be used in order to reduce part cost and power consumption. This is especially advantageous since a larger device is almost twice the cost of the next smaller alternative, shown as the multiplicative factor in parentheses in Table 1.1, which lists the average prices of Virtex-4 devices from Avnet as of February 2008. Note that the price per logic cell also increases as the device size grows. Furthermore, virtualization allows application circuits to be specialized for one application instead of another, e.g., to implement a specific protocol in a 3G multi-standard wireless basestation [48].

As device sizes continue to scale, an alternative use of dynamic reconfiguration may eventually dominate. Conceivably, complex systems involving multiple subsystems will be able to share expensive (in terms of chip area and power) and underutilized (in terms of functional density) resources such

| Device | Logic Cells | Average Price | Price/Logic Cell |
|---|---|---|---|
| XC4VLX15 | 13,824 | \$206.22 | \$0.0149 |
| XC4VLX25 | 23,192 | \$420.63 (×2.04) | \$0.0181 |
| XC4VLX40 | 41,472 | \$608.13 (×1.45) | \$0.0147 |
| XC4VLX60 | 59,904 | \$928.75 (×1.53) | \$0.0155 |
| XC4VLX80 | 80,640 | \$1493.75 (×1.61) | \$0.0185 |
| XC4VLX100 | 110,592 | \$2816.25 (×1.89) | \$0.0255 |
| XC4VLX160 | 152,064 | \$4560.63 (×1.62) | \$0.0300 |
| XC4VLX200 | 200,448 | \$8320.00 (×1.82) | \$0.0415 |

**Table 1.1:** Virtex-4 LX FPGA costs in US Dollars (source: Avnet February 2008)

as FPGAs. This will give rise to the desire to multitask FPGA devices. Another factor likely to influence the emergence of multitasked FPGAs is that devices are scaling much faster than IP size, allowing more applications to fit onto a single device. Multitasking offers a means of utilizing the available resources and further reducing system part counts.

Design complexity, verification, and time-to-market pressures encourage reuse of components and designs that are tried and proven. Module-based design methodologies form a class of higher-level design methods that focus on implementing a design that is specified or described in terms of its constituent modules [17][38]. As such, dynamic reconfiguration at the module level is ideal for implementing hardware virtualization or multitasking.

However, modular dynamic reconfiguration is not yet widely accepted nor utilized in industry and this is primarily due to a lack of practical methods for designing such applications. Current vendor tools are limited, insofar as they only perform the low-level implementation processes for area-based partial reconfiguration. The tools assume that the target application is appropriately partitioned and scheduled for dynamic reconfiguration. This entails that the system designer must have intimate knowledge of the FPGA device architecture and partial reconfiguration, and how to best design an application for dynamic reconfiguration.

There is lack of a coherent toolset to support the design and implementation of dynamically reconfigurable applications in a top-down manner, i.e., from application specification to bitstream generation. Current research deals with various aspects and sub-problems in the area but does not combine to form a coherent methodology.

If a top-down methodology is employed, the low-level device parameters should be factored in at every step of the design flow in order to ensure that the application can be feasibly implemented on the target device. In order to do this, current approaches to solving a particular step must be modified to handle these low-level constraints, or new methods must be proposed if current solutions are not applicable. The parameters for actual devices should be applied and architectural exploration should be performed to demonstrate that the methods are practicable.

This thesis proposes a top-down methodology for implementing modular dynamic reconfiguration in FPGAs. In doing so, it examines the application of current research in various aspects of dynamic reconfiguration to the proposed methodology. In order to maximize implementation feasibility, the methodology is applied to a current FPGA family that supports partial dynamic reconfiguration, i.e., the Virtex-4 FPGA family [87]. Methods proposed in current research are applied to suitable aspects of the methodology. However, some aspects require specific treatment where current research cannot be directly applied, thus novel approaches and algorithms to solving specific sub-problems in the methodology are also proposed and evaluated.

## 1.2 Thesis Contributions

The specific contributions of this thesis are described in this section.

This thesis proposes a top-down methodology for implementing dynamic modular reconfiguration. At its core, such a design methodology must implement a communications infrastructure that supports the dynamically changing communications requirements of the modules placed on the device at run time. As such, one of the main foci of this thesis is the generation of such an infrastructure given an application specification.

A complete design and tool flow in line with the proposed methodology was developed to implement dynamically reconfigurable applications with the generation of appropriate communications infrastructures.

This thesis advocates the use of point-to-point wiring harnesses as the backbone of the communications infrastructures. A novel approach to developing these wiring harnesses known as *graph merging and mapping* is proposed, and appropriate algorithms and tools are developed for it. It is imperative for a design methodology to ensure that the application can be feasibly implemented on the target device. Thus, the aim of the graph merging and mapping approach is to implement wiring harnesses that do not exceed the timing and area constraints of the target application when physically implemented on the FPGA.

A framework is also proposed within which to assess dynamically reconfigurable systems. This framework allows the iterative execution of steps in the design flow to determine the best partitioning. The input to the framework is an application, which is analyzed and prepared to be implemented as a dynamically reconfigurable application on a particular device. Two metrics are used to analyze the efficacy of the methodology — the reconfiguration delays and critical path delays.

Finally, this thesis demonstrates the use of the methodology and its associated tools by using the experimental framework to assess a variety of synthetic applications mapped to the Virtex-4 FPGA family. The experiments show the effectiveness of implementing point-to-point wiring harnesses using the methodology by performing architectural exploration.

## 1.3 Thesis Organization

Chapter 2 presents a detailed background and review of related work in modular dynamic reconfiguration. This provides the necessary background for the rest of the thesis, and summarizes current research in the area.

Chapter 3 presents the overall methodology proposed in this thesis. The methodology is top-down and consists of several steps from application specification through to implementation, and results in the final FPGA configuration bitstreams are generated.

Chapter 4 describes the specific problem addressed in this thesis. The methodology proposed in Chapter 3 aims to be complete but tackling every aspect of dynamic reconfiguration is beyond the scope of this thesis. This thesis addresses the DR1 problem, defined in Chapter 3, in which the application specification, its modules and placements are known or determined at design time. Chapter 4 describes the problem and presents its detailed specification as well as the approaches taken to solving it.

The methodology proposed in Chapter 3 advocates the use of a paged layout on the FPGA, with fixed-sized slots within which modules are placed and with the wiring between the modules and external I/O pins surrounding these slots. In contrast, traditional FPGA design methods place logic freely about the device in optimal locations. Chapter 5 examines the overheads to the critical path delay that are incurred by placing the modules and wiring in the regular structure proposed in this thesis. The impact on design time is also assessed.

As introduced in Section 1.2, the generation of a communications infrastructure is at the core of the methodology. Chapter 6 describes the approach taken to generate a set of wiring harnesses to support the dynamic communications of the modules in the target application. The *graph merging and mapping* process is described in this chapter, and the initial algorithms for the three subprocesses *subgraph merging*, the *merging of two subgraphs* and *subgraph mapping* are presented.

The algorithm presented for the main subprocess *subgraph merging* in the *graph merging and mapping* process in Chapter 6 is based on a greedy method and is sub-optimal. An algorithm based on a dynamic programming approach that considers many more possibilities for subgraph merging is presented in Chapter 7.

Chapter 8 presents an evaluation of the effectiveness of point-to-point communications for module-based dynamic reconfiguration. An experimental framework for assessing dynamically reconfigurable systems is introduced. The experiments performed to judge the efficacy of the communications infrastructure generation method are described. The results of the experiments are presented and explained.

The final, low level implementation stages of the methodology are described in Chapter 9. These stages seamlessly integrate with the commercially available partial reconfiguration tool flow from Xilinx, the Early Access Partial Reconfiguration tool flow [95]. The output of these low-level stages is a set of bitstreams for partial reconfiguration.

This thesis concludes in Chapter 10 with a summary of the research and its evaluations. Proposals for further research and implementation work conclude the chapter.

# Chapter 2

# Background and Related Work

This chapter presents the background for the rest of the thesis, while highlighting relevant research in dynamic reconfiguration. The general design flow for implementing dynamically reconfigurable applications is described, while relating each step to that for implementing static designs.

This thesis is focused on implementing communications infrastructures for modular dynamic reconfiguration, but most of the low-level aspects of the design flow such as synthesis, place and route and bitstream generation are left to commercial tools. The last section of this chapter provides a detailed overview of the low-level implementation aspects and a history of dynamic reconfiguration while focusing on methodologies and communications infrastructures.

Detailed comparisons of related work with specific contributions in this thesis are addressed where appropriate in later chapters.

## 2.1 Modular Design Flow

Modular FPGA design refers to the partitioning of an application into its natural functional units, whereby each unit, a "module", can be independently developed, implemented or modified. The modules are then assembled to form the completed application design. Modular design facilitates the reuse of well-tested and proven components and encourages a hierarchical design style. Dynamic reconfiguration, at the modular level, shares the same benefits. In addition, since modules are natural functional units of an application, they are also ideal as units of reconfiguration as each module performs a specific task, e.g., a particular stage in a pipeline. Modules can be swapped in and out of the system to perform different tasks as required.

Figure 2.1 depicts an example of an application going through the higher level stages of the modular design flow. The full application design, be it a documented application specification, HDL circuit descriptions or in some other representation, is first decomposed into its constituent modules. This decomposition is normally based on each module performing a specific function of the application, e.g., a JPEG encoder application can have the discrete cosine transform and quantization functions as individual modules.

The constituent modules can be in-house HDL circuit descriptions (e.g., in VHDL or Verilog), as in modules A and B in Figure 2.1; as third party HDL circuit descriptions (module C); or as third-party pre-synthesized netlists (module D). Third-party netlists are common in the case of commercially sourced components, when intellectual property rights are of importance. Ultimately, all the HDL has to be synthesized into netlists. Thus, the five

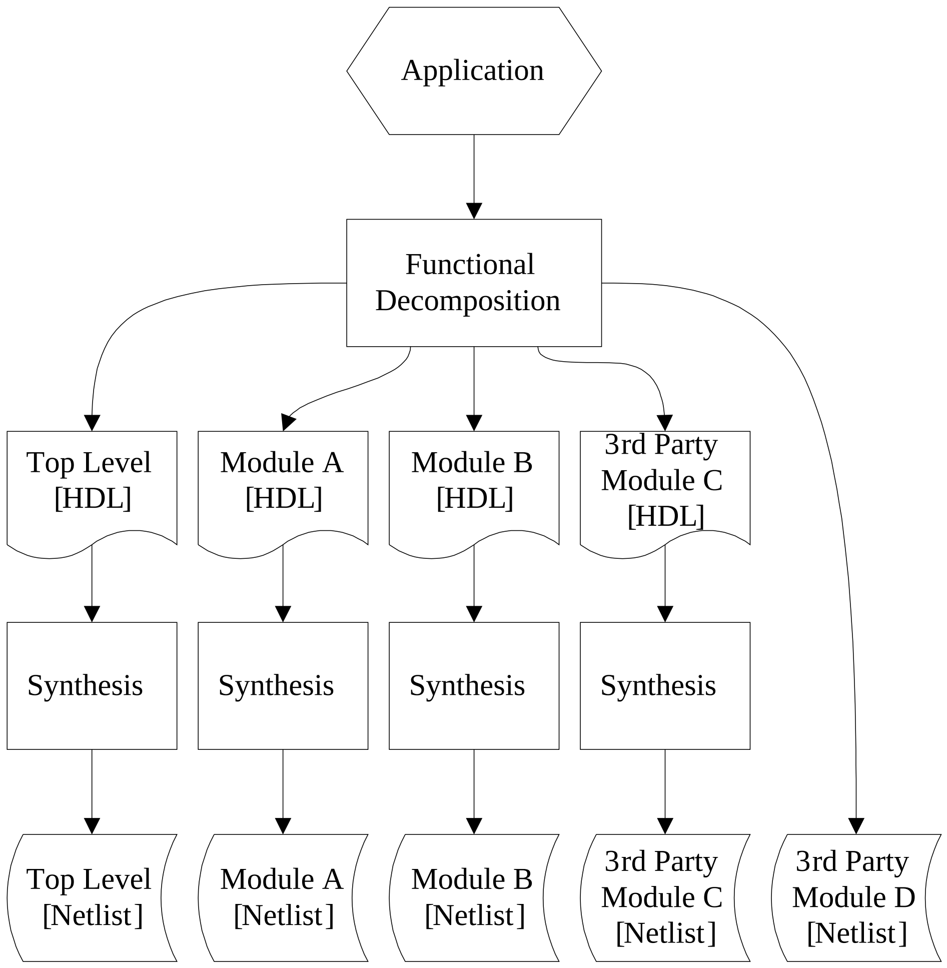


**Figure 2.1:** Higher level stages of the modular design flow

netlists shown at the bottom of Figure 2.1 are the technology independent outputs of the input stages of the design flow.

The higher level stages of the modular design flow are similar for both static and dynamically reconfigurable applications in that the top level netlist, and a set of module netlists are ultimately generated. However, as depicted in Figure 2.2, the low-level implementation stages of the modular design flow are different with respect to the static and dynamic cases.

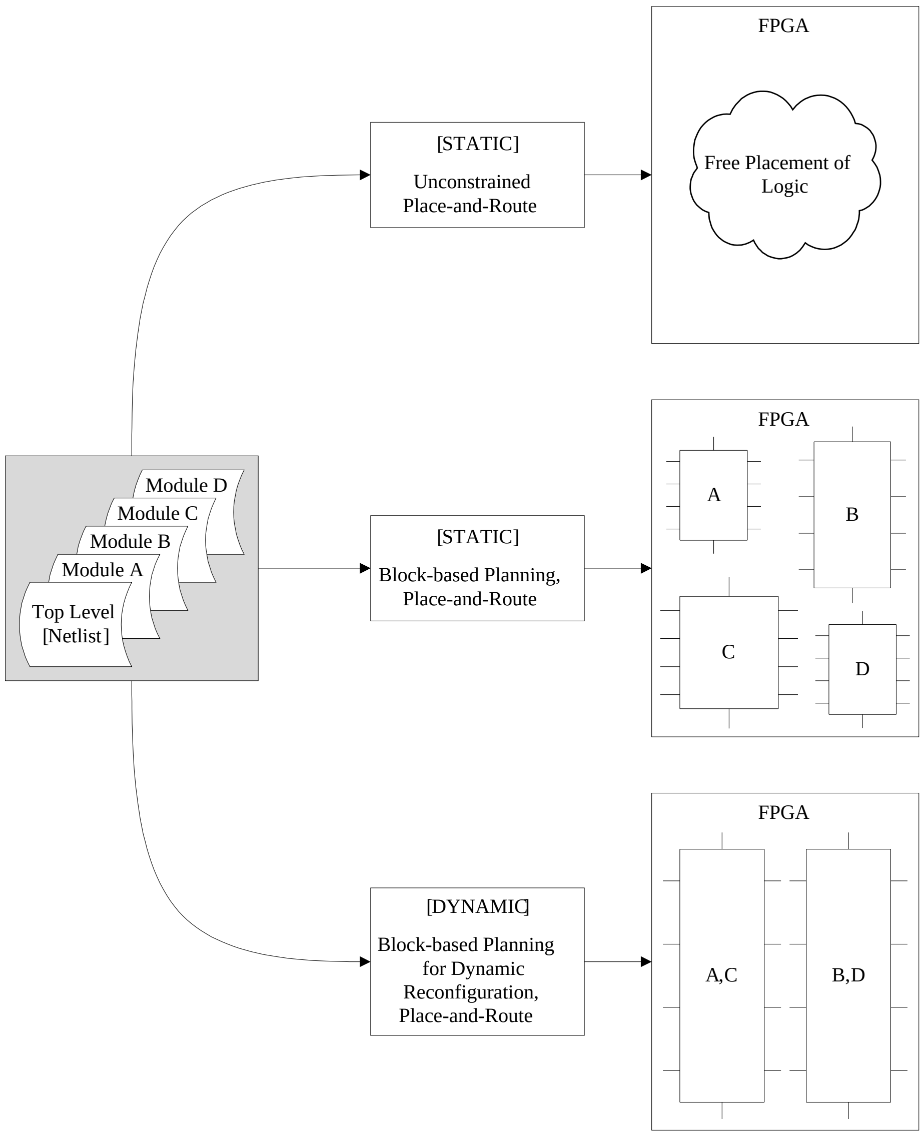

FPGA
[STATIC]
Unconstrained Place-and-Route
Free Placement of Logic
Module D
Module C
Module B
Module A
Top Level [Netlist]
[STATIC]
Block-based Planning, Place-and-Route
FPGA
A
B
C
D
[DYNAMIC]
Block-based Planning for Dynamic Reconfiguration, Place-and-Route
FPGA
A,C
B,D

**Figure 2.2:** Low-level implementation stages of the modular design flow for the static and dynamic cases

In the static case, the system designer may opt to perform an unconstrained place-and-route of the top level and module netlists, resulting in logic being freely-placed about the FPGA, as shown in the topmost example in Figure 2.2.

Alternatively, also in the static case, the system designer can also plan the placement of logic in each module in a block-based fashion, as shown in the middle example in Figure 2.2. Placing each module in a predefined area reduces place-and-route time greatly and also helps in identifying and improving critical nets. The Xilinx PlanAhead design optimization tool [96] helps the system designer to determine the best size and positions for these blocks to optimize area and timing. Note that in the static case each module must have its own physical block area in which to place logic. If two modules were to be placed in the same block, that block must be large enough for the combined logic of both modules. Thus, the FPGA must be large enough to contain all the logic for every module.

The dynamic case differs from the static case in that multiple modules can be allocated the same block, and at any one time only one of those modules is physically placed in the block. In the bottommost example in Figure 2.2, either module A *or* C can be placed in the left block of the FPGA. Consider an example where the left block currently contains module A and the right block current contains module B. At some point in time in the application, when the need arises, module A is swapped with module C. Module A is thus removed and completely replaced by module C. Depending on how the application is scheduled, the module B that is currently in the right block may continue to execute while module A is being replaced by module C. In this example, the FPGA only needs to be large enough to accommodate the combined logic of any combination of modules that can be resident at any one time, i.e., $max((A + B), (C + B), (A + D), (C + D))$.

This thesis focuses on harnessing the potential of dynamically reconfigurable FPGAs for hardware virtualization. Hardware virtualization can be used to time-multiplex an application onto an FPGA that cannot accommodate its entire logic. Alternatively, several applications could be multi-tasked onto a large FPGA. In both cases, the total amount of logic to be placed onto the FPGA is generally too large to fit onto a single device. This is the case no matter if it is a single application or a composition of several applications that is to be executed in parallel.

The strategies for block-based planning are different for hardware virtualization or multitasking. This is because the objective in hardware virtualization is to swap stages of an application in and out, whilst multitasking aims to execute multiple applications on the same device. However, there are no specific differences at the device level, e.g., the same layout of the dynamic reconfiguration example in Figure 2.2 can be used for both hardware virtualization and multitasking. A hardware virtualization scenario of a pipelined application might be where all modules A through D belong to the same application. The first stage of the application is where modules A and B are resident and module A outputs data to module B. Once module A finishes execution, module C replaces module A and module B then outputs data to module C. Thereafter, module B gets replaced by module D, then module C outputs to module D and completes execution.

In a multitasking scenario, consider two applications, one consisting of modules A and C and the other consisting of modules B and D. A and C are loaded first and the first two stages of the two applications execute in parallel. When modules A and C complete execution modules B and D are swapped in place of them and the second stage of both applications start executing.

## 2.2 General Design Flow for Implementing Dynamically Reconfigurable Applications

The design flow shown in Figure 2.3 is a more abstract view of the example flows in Figures 2.1 and 2.2. The "partitioning and scheduling" step generally only applies to dynamically reconfigurable applications and is not explicitly indicated in Figures 2.1 and 2.2. This step partitions and schedules the full application design for hardware virtualization and multitasking. Each resulting partition is then time-multiplexed onto the FPGA in the order as specified in the schedule.

The output of the final stage is a set of bitstreams for programming the device. Apart from performing place-and-route and bitstream generation, other implementation processes are included in this step such as implementing communications infrastructures (e.g., creating on-chip buses such as AMBA or CoreConnect), configuring hard cores such as the PowerPC core in Virtex-4 FX devices, and implementing reconfiguration control. These

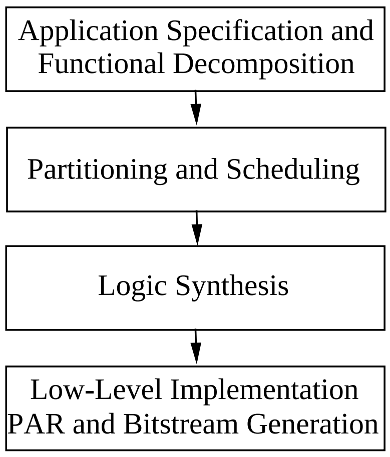

Application Specification and Functional Decomposition
Partitioning and Scheduling
Logic Synthesis
Low-Level Implementation PAR and Bitstream Generation

**Figure 2.3:** General design flow for dynamically reconfigurable applications

actions are not part of logic synthesis and mapping as they usually require intimate knowledge of the target device and may involve detailed placement and configuration of soft or hard logic blocks and communications infrastructures.

Each of these steps are described in detail in the following sections of this chapter. In addition, current and related research in the area is highlighted. However, related work in this area can be sourced from many domains, e.g., multi-processing and software/hardware co-design. Thus, instead of providing a broad review of related work, the research activity highlighted in this chapter is focused on modular dynamic reconfiguration of FPGAs and implementing communications infrastructures to support it.

### 2.2.1 Application Specification and Functional Decomposition

In this stage an application is divided into modules, each of which performs a specific function. This may be hierarchical, as each module may in turn be composed of several sub-modules. This section discusses application representations for dynamic reconfiguration after functional decomposition.

At the highest level of abstraction, an application may be expressed at a structural level as a result of extracting the key functions that are performed, and making each key function a module. This is shown in Figure 2.4 for the pre-processing stages of an optical flow algorithm [15]. At a lower level, modules can be specified in a hardware description language such as VHDL or Verilog.

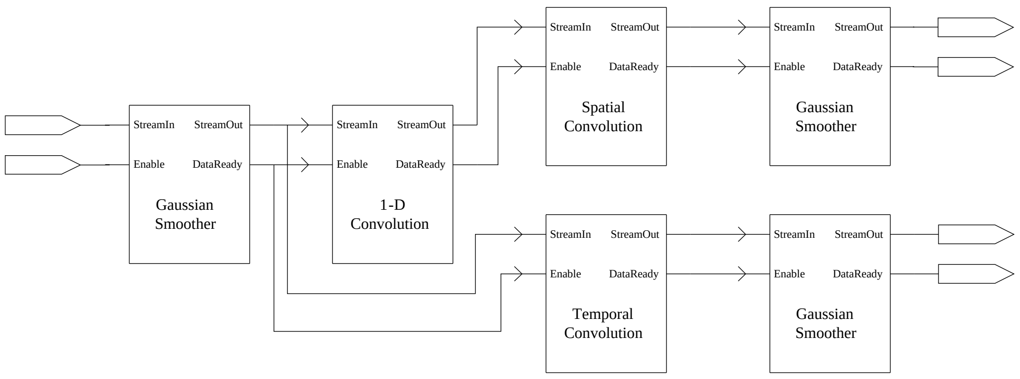


**Figure 2.4:** Optical flow tensor preparation modules

This form of representation is commonly used for static designs, but may not be suitable for dynamic reconfiguration. This is because neither the period of activity for each module, the intervals during which communications occur, nor the bandwidth needs are specified.

Dynamically reconfigurable applications are rarely specified only in this manner, unless they have some form of implied temporal representation. For example, the modules in Figure 2.4, are a part of a pipeline in which each module depends on the previous one for data. Based on this data dependency, the modules should execute from the left to the right of the diagram, and one module may not begin executing before its dependent module produces data for it to consume. Examples of dynamically reconfigurable applications specified in this manner are typically image processing pipelines [70] [22]. Since dynamically reconfigurable applications specified in this manner rely on their implied temporal representation, this is insufficient in general to obtain enough temporal information to allow an automated tool to partition and schedule applications for dynamic reconfiguration.

Another class of dynamically-reconfigurable applications are simple enough to specify with a structural module diagram such as Figure 2.4. These are those that swap one version of a module with another, e.g., a Bluetooth baseband unit swapping between the header and payload processing or between receiving and transmitting [25], and swapping waveform processing modules in software defined radio [79]. Timing is usually not specified or is user-dependent. Where a module may have multiple versions it may suffice to specify it in the application specification document such that the system designer creates these multiple versions when writing the HDL.

Another, more detailed form of application representation is a task graph. A *task* is taken to mean some computation that a module performs in some specified time. Task graphs are usually specified by directed acyclic graphs (DAGs) in which each node represents a task. Each edge indicates data dependency, i.e., the destination node requires the source node to produce some data for it to consume An edge can also encapsulate some other communications information, e.g., the number of bits of data transferred, the input/output port names, etc. Task graphs for dynamically reconfigurable applications are typically specified with nodes representing a hardware module. For example, the module diagram in Figure 2.4 is represented temporally as tasks in Figure 2.5. This task graph also consists of sub-modules not shown at the level of abstraction in Figure 2.4.

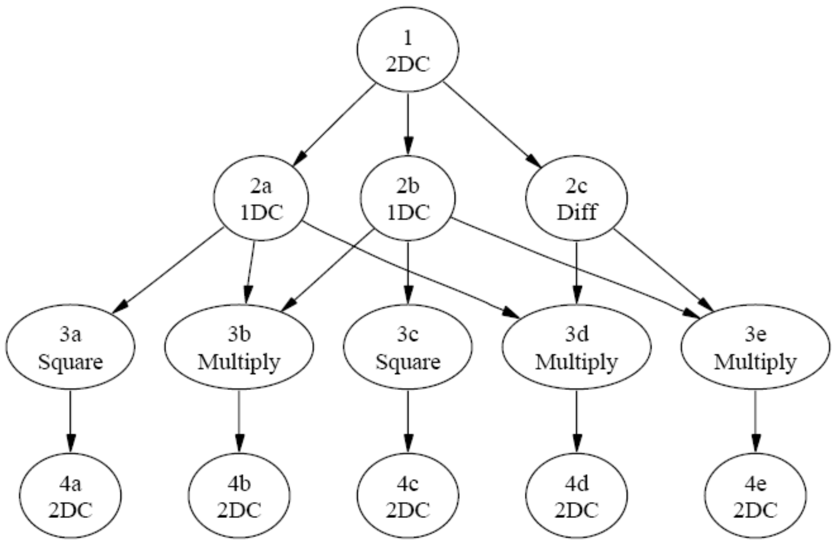


**Figure 2.5:** Optical flow tensor preparation task graph

Information about *temporal dependency* is specified as a node should only start to execute once all of its predecessors have started execution and have produced some data for it to consume. Some task graphs have nodes annotated with earliest and latest start and end times to more precisely specify the application's timing constraints.

As stated earlier, edges can also represent a communication flow, e.g., with an attribute that specifies the quantity of data that flows from one node to the other. This allows partitioning tools to try to optimize for communications density, e.g., by minimizing the total amount of communications between partitions.

Task graphs are used extensively by researchers in dynamic reconfiguration especially for research in partitioning and scheduling [2] [18] [28] [64] [66] [69] [77].

Some variants of task graphs that are not acyclic allow the specification of loops [9] and conditionals [82] in the input specification. These graphs can specify data dependent application execution where the system designer does not know *a priori* the exact sequence of tasks that will be executed.

### 2.2.2 Partitioning and Scheduling

In the execution of a module-based dynamically reconfigurable application on an FPGA, modules are being swapped in and out of the device at different points in time. Without loss of generality, the full module or task graph, i.e., the output of functional decomposition, is too large to fit onto the FPGA. This is the case for both hardware virtualization and multitasking. As such, this full graph must be *partitioned* into smaller subgraphs, each of which represent the modules resident on the FPGA at some point in time. These subgraphs should be in turn *scheduled* to specify the order and/or times each subgraph or node should be resident on the device.

Application partitioning has been studied extensively especially in the domain of multi-processor task assignment. As this thesis does not focus on partitioning this section presents a targeted review of approaches that are relevant to modular dynamic reconfiguration, as well as traditional techniques that have evolved to target dynamic reconfiguration on FPGAs.

Purna et al. [69] proposed two algorithms to partition and schedule a task graph for dynamic reconfiguration in FPGAs. As one of these algorithms is used for experiments in this thesis, an outline of their approaches are given here.

An example of the task graphs considered by Purna et al. is depicted in Figure 2.6. The numbered annotation alongside each node indicates its dependency *level* in the graph. For example, nodes of level 3 may not execute until all its dependent nodes of level 2 and above have started execution and have produced data for them to consume.

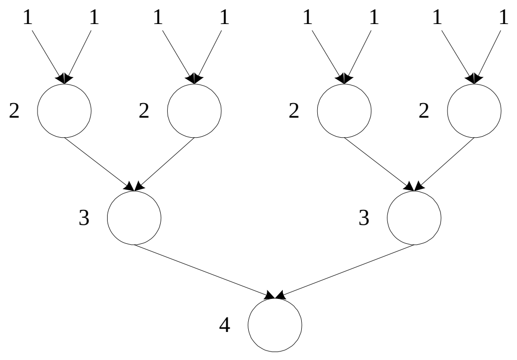


**Figure 2.6:** Task graph example considered by Purna et al. with level assignments

The first algorithm, the "level-based partitioning" approach, tends to maximize the amount of parallelism between partitions by maximizing the number of nodes that are of the same level of dependency in each partition. An example of this partitioning is shown in Figure 2.7, where the maximum size of each partition is 2 nodes. Partition P1 has two nodes at the same level of dependency, i.e., level 2, thus both tasks can run at the same time.

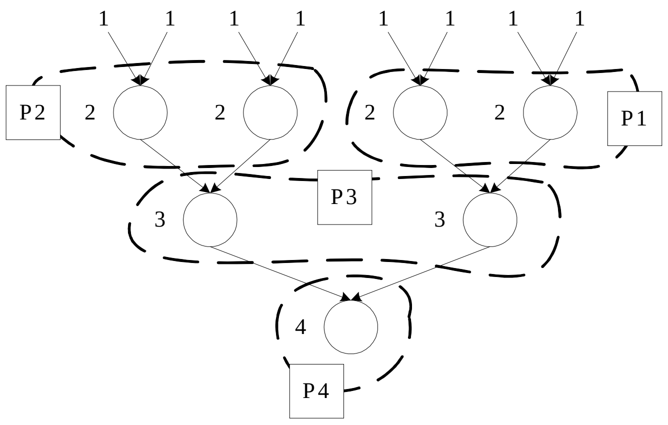


**Figure 2.7:** Task graph example with level-based partitioning

The second algorithm, the "clustering-based" approach, adds nodes into a partition as soon as its dependencies are resolved. An example of this is shown in Figure 2.8, again where the maximum size of each partition is 2 nodes. Note that partition P1 using the level-based algorithm has 4 incoming edges and 2 outgoing edges, while partition P1 using the cluster-based algorithm has 3 incoming edges and 1 outgoing edge. Adding all the external edges of every partition together, the level-based partitions have 14 external edges whilst the clustering-based partitions have 12 external edges.

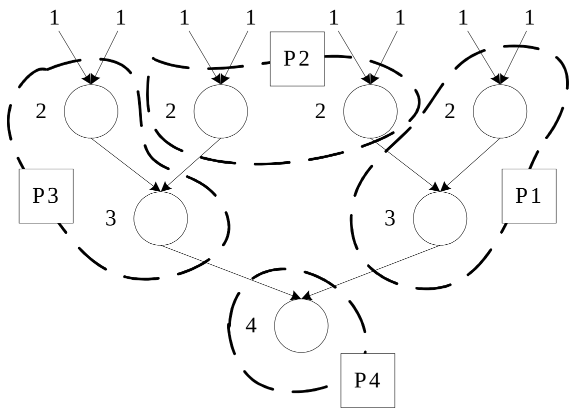


**Figure 2.8:** Task graph example with clustering-based partitioning

A concise and thorough review of other partitioning approaches is provided in Bobda's thesis [9], which also proposes improvements over classical list-scheduling approaches and an improved spectral method. List scheduling is a class of scheduling algorithms that order tasks by priorities and subsequently selects one of the highest priority tasks for scheduling.

Partitioning and scheduling algorithms have typically used input task graphs specified as DAGs. However, DAGs are not amenable to specifying applications with conditionals or loops. Ganesan et al. [27] proposes extracting Control Data Flow Graphs (CDFGs) from the behavioral specification of the application. CDFGs specify loop and conditional constructs and is the input into the partitioning and scheduling algorithm proposed by Ganesan et al. [27]. Xie et al. [82] introduced partitioning and scheduling algorithms for Conditional Task Graphs (CTGs) [23]. CTGs are task graphs with conditional edges specifying mutually exclusive operations. Bobda [9] handles cycles by modifying Kernighan and Lin's [47] iterative improvement procedure to handle directed graphs. As stated at the beginning of this section, each partition represents a configuration in the FPGA. If two partitions contain nodes in the same loop, then each of the partitions must be swapped in and out for each loop iteration, resulting in thrashing. Thus, these approaches share a common technique – to try to fit any cycles and branches into a a minimum number of partitions, thereby minimizing the amount of thrashing that may occur.

Finally, Banerjee et al. [5] highlights an important issue: approaches that ignore physical constraints can lead to implementation infeasibility even if the partitioning and scheduling is optimal. Being physically-aware at the high level is especially relevant to dynamically reconfigurable FPGAs as the reconfiguration delays are significant, possibly even exceeding the application run time. An example of an approach that considers physical constraints is by Tanougast et al. [77], in which data flow graphs are annotated with timing constraints to ensure that the partitioning algorithms take them into consideration. It is also not always consistent in the literature as to how a partitioning approach is integrated with module placement, as the partitioner must be mindful of the physical constraints as well. A methodology to implement applications on a dynamically reconfigurable FPGA must consider the area and timing constraints in order to guarantee that the application can be mapped.

### 2.2.3 Logic Synthesis

This stage consists of taking the partitions obtained from the previous stage, together with the HDL in the application specification and synthesizing them into netlists. Modern synthesis tools are also optimized for the target FPGA. This is typically performed by synthesis tools such as the Xilinx Synthesis Tool [94] in the ISE Tool Suite [92] or Synplicity's Synplify and Synplify Pro [76].

Xilinx currently offers two approaches to dynamic reconfiguration — a new module-based flow [95] in which modules are dynamically loaded at run time, and a difference-based flow in which a partial bitstream consisting of the difference to the previous bitstream is loaded at run time [86]. Either flow can be used for this step but one may be preferred over the other if the partitioning and scheduling is targeted for a particular one. Using the module-based flow simply enforces the adherence to module and interface naming guidelines in the HDL (see Chapter 9 for more details).

There are no other significant differences at this stage compared with traditional FPGA logic synthesis.

### 2.2.4 Implementation

In this stage the following is performed:

First, some low-level device-specific system architecture components are placed and/or configured. For example, it may be necessary to implement communications infrastructures such as on-chip buses (e.g., AMBA or Core-Connect), configure hard cores such as the PowerPC core in Virtex-4 FX devices, set up clock domains and clock managers, assign package pins, and in the case of dynamic reconfiguration, implement reconfiguration control through an on- or off-chip mechanism.

Next, the netlists obtained from the synthesis stage and pre-synthesized netlists are mapped to a specific FPGA and placed-and-routed. Then, the full and partial bitstreams to be loaded onto the FPGA are generated.

## 2.3 Support for Module-based Dynamic Reconfiguration — A History

The partial reconfiguration tool flows provided by Xilinx only place-and-route and generate the bitstreams for the application, and do not provide other facilities such as communications infrastructures and reconfiguration control. It is thus insufficient for effective implementation of dynamically-reconfigurable applications. This section provides a technical history of dynamically-reconfigurable Xilinx FPGAs and highlights representative research performed at the implementation level.

Note: At the time of writing of this thesis, Xilinx was the only vendor to support dynamic partial reconfiguration. Thus, all the approaches to implementing module-based dynamic reconfiguration on FPGAs to date are based on Xilinx devices.

### 2.3.1 Dawn of an Era — The XC6200

In May 1995, the first dynamically-reconfigurable FPGA family (the Xilinx XC6200 family [83]) was introduced. XC6200 FPGAs could be configured using a full parallel CPU interface known as FastMAP. With FastMAP, the entire configuration SRAM of the FPGA can be addressed by a host computer as conventional memory mapped SRAM. This therefore allows direct run time modification of the configuration SRAM at the granularity of a single byte. The format of the configuration bits is also fully specified in the datasheet [83], allowing for any degree of fine-grained logic or routing manipulation at run time.

Actual dynamically-reconfigurable applications were implemented using XC6200 FPGAs such as speeding up loop computations [13], DCT algorithms [63] and virtual control circuits [65]. Even at this early stage, attempts were made to produce synthesis tools and a design environment for dynamic reconfiguration [75] and reduce run time reconfiguration overheads [33][19].

A run time support system [22] to manage reconfiguration was built and successfully run with a wavelet image compression application. The authors, however, stated that performance results were poor and this could be due to the large amount of hardware overheads involved in implementing such fine-grained reconfiguration.

### 2.3.2 A One-Dimensional View — The Virtex Family

In 1998 Xilinx released the next generation of FPGAs — the Virtex family [84]. Although these FPGAs could still be partially reconfigured, the reconfiguration granularity was increased to that of a *frame* [88]. A Virtex frame is a one-bit-wide vertical slice of configuration data spanning an entire column of configuration resources. The amount of data in each frame thus depends on the number of CLB rows in the device, and ranges from 384 bits for the smallest XCV50 device to 1248 bits for the largest XCV1000 device. In the Virtex family of devices, the number of frames per column of CLB resources ranges from 50 to 55 [88]. The XCV50 device floorplan is depicted in Figure 2.9, with an example of a portion of the device marked for reconfiguration.

Using the fastest mode of configuration (SelectMAP at 50 MB/sec) loading a single frame takes up $0.96\mu s$ for the XCV50 (in which a column is 16 CLBs high) and $3.12\mu s$ for the XCV1000 (in which a column is 64 CLBs high). As each frame is a single bit wide, it is possible to reconfigure only a portion of a column of CLBs while keeping the rest of the configuration unchanged. Due to this Virtex devices also introduced a feature known as *glitchless reconfiguration* that guarantees that no glitches will occur in the

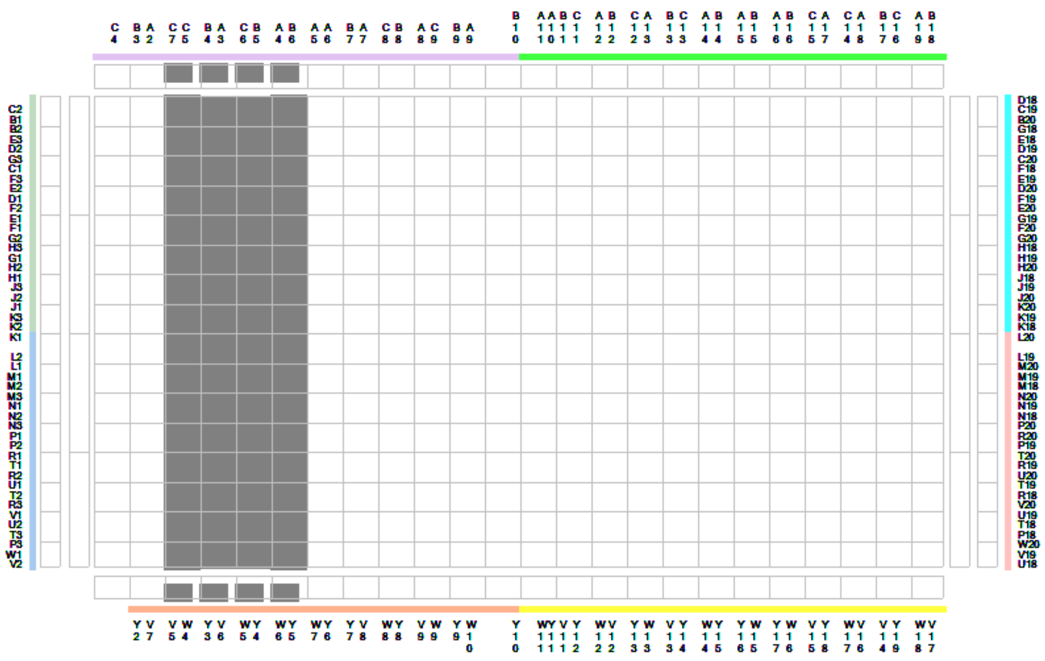


**Figure 2.9:** XCV50 device floorplan with a portion of CLBs marked for reconfiguration

operation of any portion of the FPGA that is loaded with the same configuration data. This allows part of a column of CLBs to be modified while keeping the rest of the column intact and operating without interruption. It is unclear from the literature to what extent this is implemented in Virtex FPGAs, but it is clearly stated in [8] that the next generation of FPGAs, Virtex-II (see Section 2.3.4) and beyond fully support glitchless reconfiguration.

### 2.3.3 JBits — Java Bitstream Manipulation

In 1999 Xilinx released a tool called JBits [30] that allowed the modification of bitstreams at a very fine-grained level. Individual routes, LUT contents etc. could be modified via a Java interface with a full library of the device resources in the Virtex family. Run time reconfiguration is possible by reading back the bitstream in the FPGA (or using one that is currently in the host PC memory), making modifications to it, and downloading it back into the FPGA.

The approach is limited because modifications are done at the lowest level of the design flow — direct logic and routing manipulation. Reading back bitstreams, modifying them and downloading them through a Java interface running on a host computer incurs a very large overhead as compared to pre-generated bitstreams and direct host programming methods such as SelectMAP. Another limitation is that communications between cores was not addressed apart from a suggestion of using “Stitcher” cores that connect modules together [29].

Nevertheless JBits allowed a lot of research to be performed at the implementation level, and had a device simulator that could be used by researchers to monitor and record the execution of the device for every clock cycle. However, run time reconfiguration could not be performed efficiently at that level of bitstream manipulation and it was clear that a different method was necessary.

### 2.3.4 The Second Generation — Now with PowerPCs

In 2000 the Virtex-II family was introduced, followed by the Virtex-II Pro in 2002, which also included embedded PowerPC cores. The reconfiguration mechanism for the Virtex-II/Pro families was identical to the frame-based method employed on the first generation Virtex family, and thus the reconfiguration granularity still depended on the size of the device.

A useful feature that was introduced in the Virtex-II device family was the Internal Configuration Access Port [90] (ICAP). Up until then the only way to configure an FPGA was through external configuration pins using either the serial JTAG [85] or byte-wide SelectMAP [88] interfaces. The ICAP is an internal access port to write and read back bitstreams using a similar interface to SelectMAP. This opened up the possibility of designing self-reconfiguring systems without the overhead of accessing an off-chip configuration controller. The XC6200 also supported internal configuration, but the device family was obsolete by then.

At this time there had already been a strong interest, especially in academia, in partial and dynamic reconfiguration of these devices. However, there was no tool support to implement partial reconfiguration other than to generate the difference in bitstreams between two different configurations. A "difference bitstream" only contains the frames that are different between two configurations, thus it is a partial bitstream, and the time to download it into the device is less than a full one. However, it was not officially documented as to whether loading a bitstream difference onto the FPGA was safe, because proper partial configuration requires specialized headers and control frames that may not be generated properly in a difference bitstream. Three years after the Virtex-II family was introduced, Xilinx released an application note [86] and software overlay that modified core ISE synthesis and place-and-route tools to support partial reconfiguration[1].

[1]This was actually based on a 2002 draft when partial reconfiguration capabilities were first integrated into ISE 4.2i.

### 2.3.5 XAPP290 — Long-Awaited Tool Support

Xilinx Application Note 290 (XAPP290) [86] specifies two types of partial reconfiguration — module-based and difference-based. With module-based reconfiguration the designer specifies "Partially Reconfigurable Regions" (PRRs) in which modules (i.e., a VHDL entity or Verilog module) can be placed. Different modules can then be loaded onto these PRRs at run time.

There are two major shortcomings to this method of module-based reconfiguration. The first is that each PRR must span the entire height of the device. This can force module logic to be placed and routed into a narrow slice of the device, which may result in long routing paths within the module. If a larger region than is required is allocated to place the amount of logic in the module, then the available area is underutilized. A study by Kalte et al. [40] showed that the critical path delay may be increased to as much as twice that of an optimal implementation if a module is packed into its narrowest possible region.

The second criticism is that modules can only communicate between adjacent modules, as shown in the design layout example provided in the application note, reproduced here as Figure 2.10. All intermodule communications go through so-called "Bus Macros" placed between PRRs and adjacent PRRs or static module regions. An example is depicted in Figure 2.11, which was reproduced from the application note. Bus macros are implemented using three-state[2] wires spanning the horizontal width of the device. It allows mod-

[2]This is equivalent to the term "Tri-state", which is a registered trademark of National Semiconductor.

ules on either side of the boundary to drive the three-state buffers, which can be read by any CLB attached to it. They are essentially connectors placed in fixed locations connecting reconfigurable modules to other modules. Bus macros are essential as they ensure that the I/O ports of every module that is to be placed in a particular region connect to the same location.

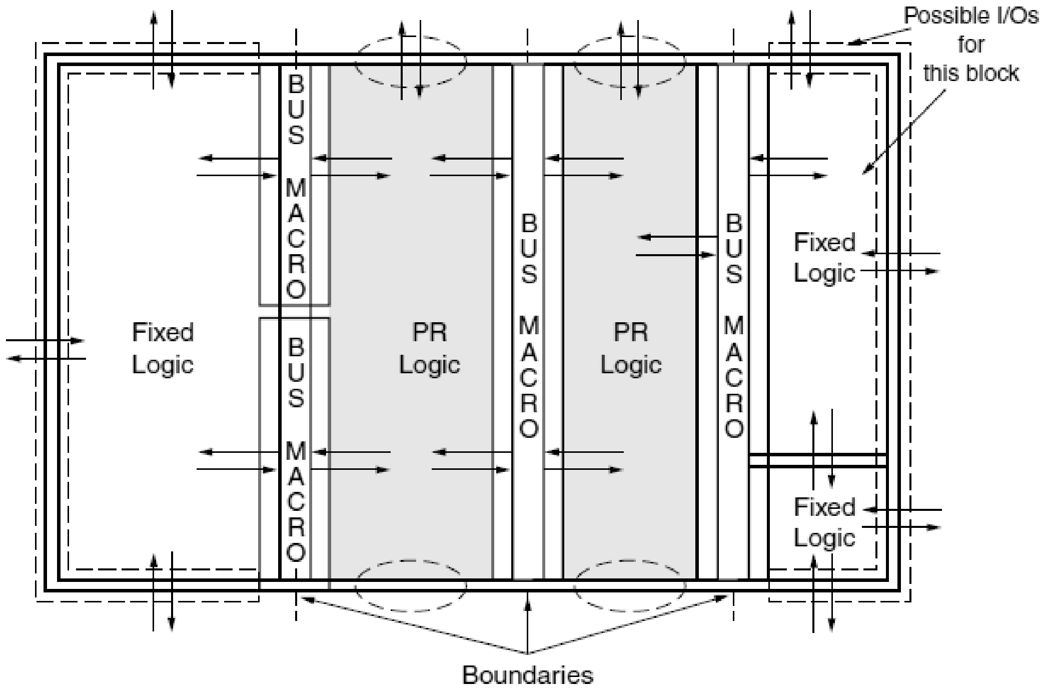


**Figure 2.10:** Xilinx application note 290 design layout with two partially reconfigurable regions (PRRs)

Difference-based reconfiguration involves generating *difference bitstreams* to partially reconfigure a device. A *difference bitstream* between bitstream $x$ and bitstream $y$ comprises the frames that contain bits of bitstream $y$ that differ from bistream $x$. *diff(x,y)* $\neq$ *diff(y,x)*, thus a difference bitstream has to be generated for each source and destination bistream. Difference-based reconfiguration reduces reconfiguration bandwidth. This thesis employs both

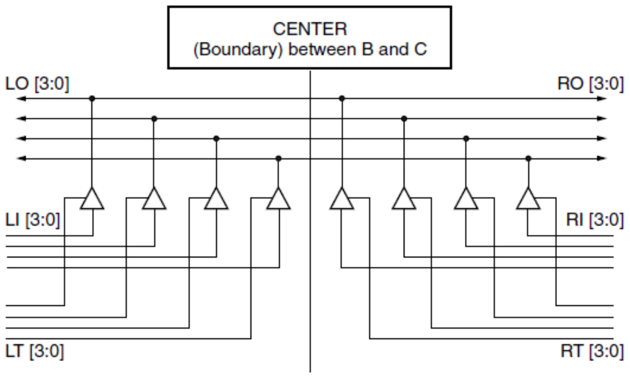


**Figure 2.11:** Bus macro used in XAPP290

module-based and difference-based reconfiguration depending on the situation.

### 2.3.6 Limitations of XAPP290

There are several limitations to XAPP290. These are mainly due to the use of three-state wires and the placement of bus macros between neighboring modules:

- **Delay:** Three-state wires are slower than wires in the general routing matrix as 3-state buffers are inherently slower than pass-transistors that are used in the switchboxes [7].
- **Density:** The same three-state wire spanning the width of the device cannot be used by more than two modules, thus the total number of bits available for bus macros is the number of three-state wires available in the device. Even though a three-state wire can be tapped at any horizontal location on the device, the use of bus macros does not allow this as the specification states that each bus macro must have a driver,

and there cannot be more than one driver per three-state wire. This is a severe limitation as there are only four three-state wires per CLB row. The smallest device has only 8 CLB rows, with a maximum of 32 bits of communication available for reconfigurable modules in the entire device. The largest device therefore provides 448 such wires.

- **Module Placement:** Even though the three-state wires span the entire width of the device, a reconfigurable module can only communicate with an adjacent module (be it static or reconfigurable). If two non-adjacent modules need to communicate with each other, the modules between them have to provide ports to allow routing to pass through.
- **Currency:** Devices in the current Virtex-4 and Virtex-5 families do not have three-state wires, thus these bus macros cannot be used.

As researchers began to use XAPP290 it became clear that a better communications infrastructure was needed in order to support dynamic modular reconfiguration. However, there was no other official method to implement partial reconfiguration at that time, and there was nothing that could be done to reduce three-state wire delay or to implement partial reconfiguration on the new Virtex-4 devices when they were first released. The problems regarding limited communications density and adjacency were thus addressed by some researchers.

### 2.3.7 Overcoming Limitations at a High Level

Marescaux et al. [60] proposed one of the earliest communications infrastructures using module-based reconfiguration in XAPP290. The approach is to implement a network-on-chip using Wormhole Routing [21], i.e., a routing infrastructure with a torus topology. The network is “folded” into a single dimension as depicted in Figure 2.12 in order to be placed onto the chip.

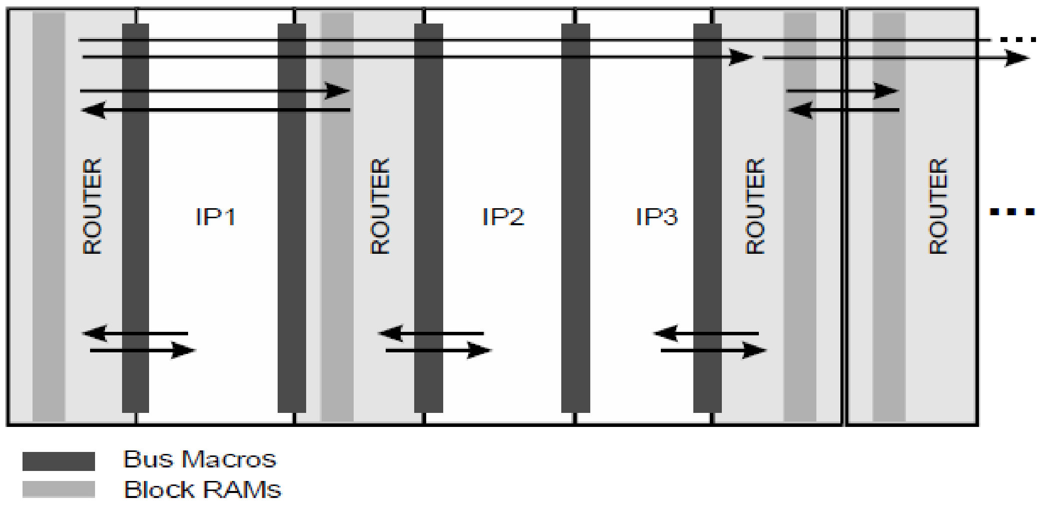


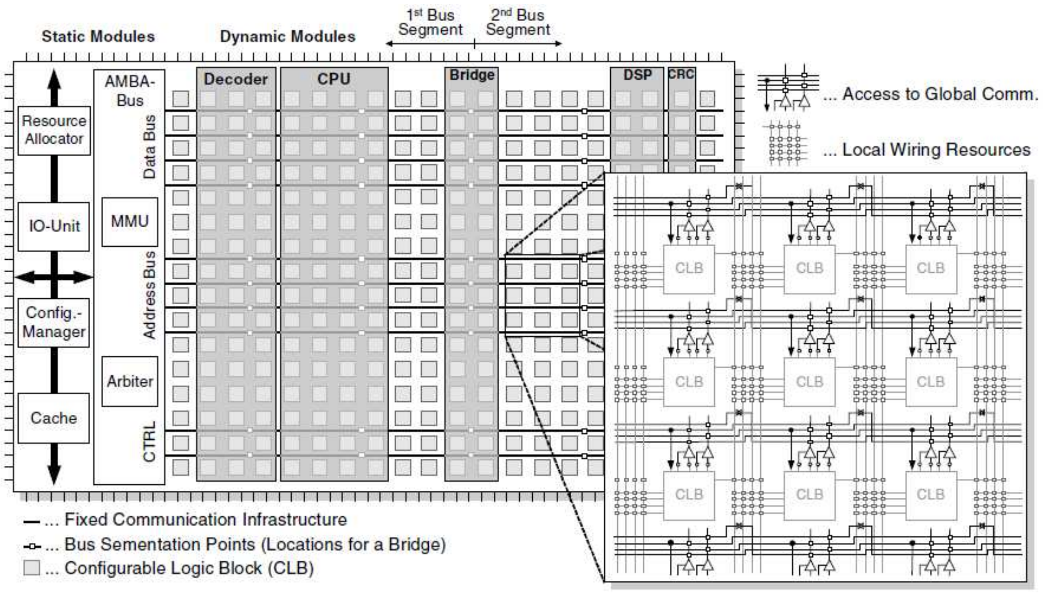
**Figure 2.12:** Wormhole routing (Marescaux et al.)

This approach inherits all the limitations of XAPP290, but attempts to overcome them at a higher-level. However, it also introduces its own limitations. For example, all modules must implement the network interface, or some form of wrapper must be synthesized for them. A routing protocol has to be used, thus incurring communications and area overheads. In addition, the maximum number of routers it can support ranges from 2 for the smallest device to 6 for the largest device. This is because the routers need to use BRAM resources.

Since using XAPP290 alone is not sufficient for implementing real applications, researchers considered other approaches. Methods that utilized the partial reconfiguration capabilities of XAPP290 but which provided sophisticated custom communications infrastructures and management modules began to emerge.

## 2.3.8 Early Extensions to XAPP290

Kalte et al. [43] proposed using the three-state wires as buses instead of connectors between adjacent modules, as illustrated in Figure 2.13.



Static Modules
Dynamic Modules
1st Bus Segment
2nd Bus Segment
Resource Allocator
IO-Unit
Config.-Manager
Cache
AMBA-Bus
MMU
Arbiter
Data Bus
Address Bus
CTRL
Decoder
CPU
Bridge
DSP
CRC
... Access to Global Comm.
... Local Wiring Resources
CLB
... Fixed Communication Infrastructure
... Bus Sementation Points (Locations for a Bridge)
... Configurable Logic Block (CLB)

**Figure 2.13:** Bus system with relocatable modules (Kalte et al.)

The approach overcame the limitations of XAPP290 in several ways.

Firstly, it allowed modules of any width to be placed at any location in the reconfigurable area. It did this by allowing the bus lines to be tapped at any location on the device. This was achieved by laying out bus macros throughout the device. As the three-state lines were used as bus lines, bus macros composed of three-state buffers could not be used. Instead, as understood through conversations with the authors, the bus macros used were composed of slices connecting the modules to the three-state lines. This was one of the first uses of slice-based bus macros, which were adopted by Xilinx about two years later.

Secondly, it allowed the same three-state wire to be used by more than two modules simultaneously by segmenting the bus bridge modules, as depicted in the middle of the reconfigurable area in Figure 2.13. Bridge modules also allowed signals to pass through them to allow inter-segment communication.

Finally, it provided communication and run-time reconfiguration management. A standard bus protocol was used for communications management. AMBA is depicted in Figure 2.13 but was later superseded by a custom bus protocol [42]. Other modules provided run-time control of the application by allocating resources, managing external I/O and loading partial bitstreams onto the device.

The approach is currently implemented on a custom board [42] with a custom bus protocol. Most recently, modules were developed to filter partial bitstreams at run time such that they could be relocated to a different area of the device [41] [39]. The authors have also proposed module placement defragmentation [54].

There is a large number of overheads involved in managing this system. These include the bus protocol communication overheads, bitstream relocation filtering, loading bridge modules, etc. In addition, three-state wires are inherently slow, especially if they have to travel across the entire device. Finally, due to the fact that modules can be placed at any location, a large number of bus macros must be laid out on the device regardless of whether they are used or not.

Allowing modules to be freely sized and relocated is expensive, and requires a lot of manipulation as Xilinx tools require that each PRR has a fixed size. Slot-based approaches that have fixed reconfiguration times per slot have also been proposed. Relocation can also be emulated by generating versions of the module bitstreams placed at each slot.

Ullmann et al. [80][34][35] proposed the "On-Demand Run-Time System" as shown in Figure 2.14. This system resembles the approach by Kalte et al. in many ways including the implementation of a custom bus system, albeit it is not co-located in the same FPGA area as the modules. It also implements bitstream decompression and self-reconfiguration via the ICAP.

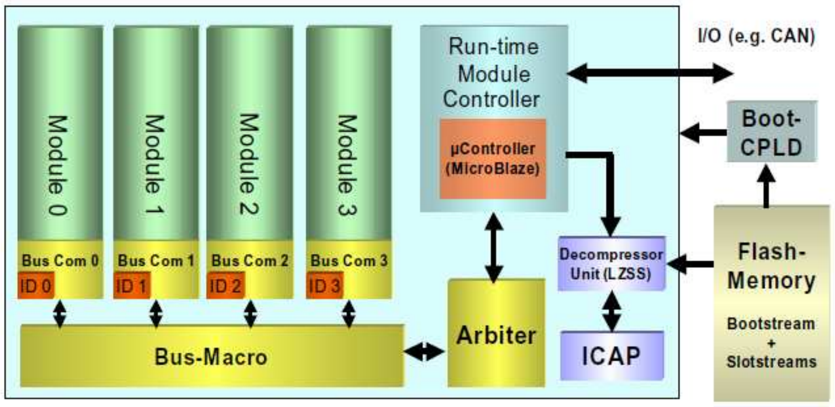


**Figure 2.14:** On-demand run-time system (Ullmann et al.)

Although this approach does not suffer from the design complexity and overheads of managing freely-sized and placed modules, bus protocol overheads still exist. In addition, there can be long routing paths from the top of the modules to the bus interfaces at the bottom and then through the bus network (labeled as "Bus-Macro" in Figure 2.14). It is also unclear as to how one designs applications targeted to this system.

A more recent one-dimensional slot-based approach is the Erlangen Slot Machine (ESM) by Majer et al. [57]. The ESM is composed of two PCBs — a BabyBoard and a MotherBoard, as depicted in 2.15. The BabyBoard has a reconfigurable Virtex-II FPGA, SRAM and a reconfiguration manager responsible for bitstream relocation and loading. Modules are loaded into fixed-sized slots on the Virtex-II FPGA. The MotherBoard is composed of a crossbar switch that links external I/O to the BabyBoard, and a PowerPC that runs software to control the application, e.g., operating system like functions.

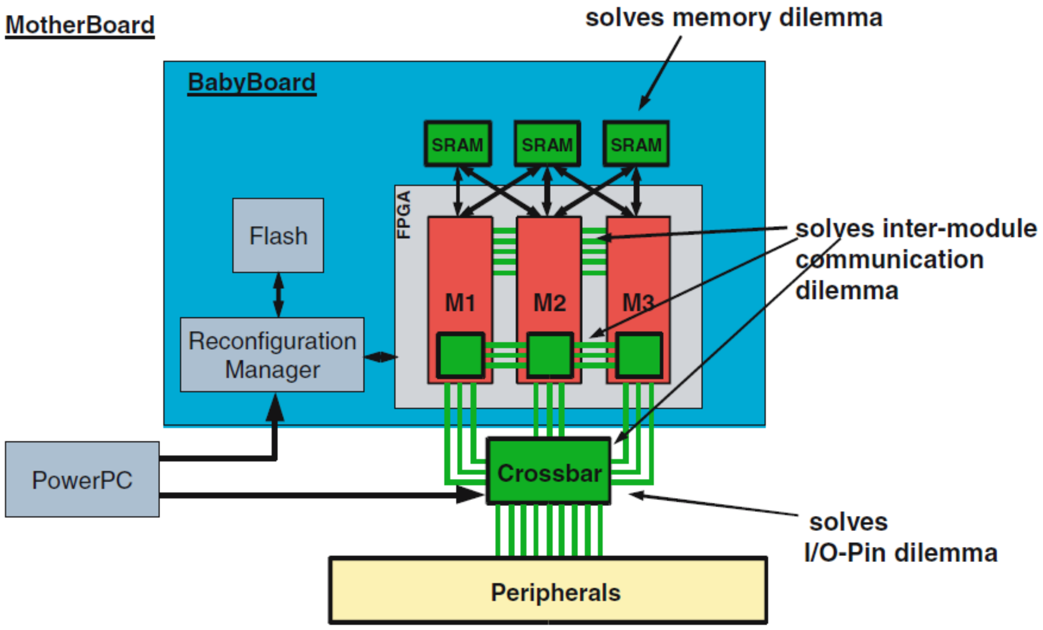


**Figure 2.15:** The Erlangen Slot Machine (Majer et al.)

There are four levels of inter-module communications provided on the ESM. The first is direct communication between adjacent modules akin to using the bus macros in XAPP290. The second level is shared SRAM access, again between neighboring modules. At the third level non-neighboring modules can communicate via a modified version [1] of a Reconfigurable Multiple Bus (RMB) [24]. Finally, the crossbar can be used for off-chip intermodule communication as all the modules are connected to it.

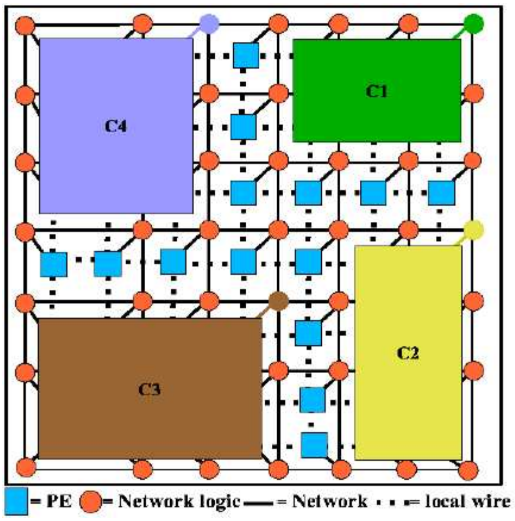
This approach also suffers from area and timing overheads required to manage the system and its communications. The RMB requires crosspoint modules consisting of a controller, FIFOs and data network. External I/O is very expensive as it has to go off-chip into another FPGA which in turn has to be routed through the crossbar and then off-chip again to the actual peripherals or off-chip logic. The delay of the crossbar itself is 15 ns with an 18 clock-cycle setup time.

Programming such a system is an extremely complex task. The multiple levels of communication and application management add a high degree of complexity at the application design level. This makes it even more difficult to ensure that area constraints are met and that timing closure is achieved.

### 2.3.9 Moving to Two Dimensions

Perhaps the main reason that most approaches seen to date are one-dimensional in nature is due to the configuration architecture of the Virtex and Virtex-II/Pro FPGAs. Two-dimensional approaches offer more flexibility in placement and thus the possibility of shorter intermodule wiring paths.

The Dynamic Network-on-Chip architecture (DyNoC) proposed by Bobda et al. [10] [11] is a two-dimensional network-on-chip with routers laid out in a grid on an FPGA (see Figure 2.16). Rectangular modules of any size can be placed onto the network. Routers are disabled if they are temporarily obscured by a module placed over them, and are re-enabled when the module is removed.



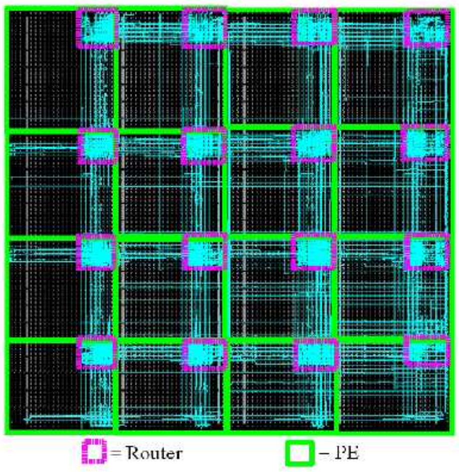


**Figure 2.16:** DyNoC (left: conceptual, right: implemented) by Bobda et al.

The flexibility of such a two-dimensional layout has to be weighed against the complexity of implementing applications on it, as well as against the overheads involved in routing communications.

It is a complex task to determine the appropriate set of shapes and placements of the modules in the application such that area and timing constraints are met. This is accentuated by the fact that it is very difficult to provide network bandwidth guarantees. The complete temporal communication patterns of the application must be known. Even then, doing so requires that multiple NP-hard problems be solved, as determined by Chan et al. [16] in addressing a similar problem in NoC topology generation and module shaping.

In addition, the uncertain bandwidth is further impacted by the overheads involved in routing packets. Not only does this include the router and interface delays, but as the NoC is packet-switched, the modules have to be designed in such a way that they can accept packets that arrive out-of-order.

Finally, some area of the chip is underutilized because a ring of unobscured routers must surround every module in order to ensure routability.

Technically speaking, a two-dimensional layout such as this is counter-intuitive to the reconfiguration mechanism of the Virtex and Virtex-II devices. Since a configuration frame spans the entire height of the device reconfiguring a module that is wider than it is high will take longer than one of the same size that is rotated 90 degrees. This tends to indicate that the best way to lay out modules would be to pack them into the narrowest possible rectangles, thereby reverting to a one-dimensional layout.

### 2.3.10 Limitations of Current Extensions to XAPP290

All the approaches to extend XAPP290 that were described in this section share a common motivation — to implement a dynamic communications infrastructure capable of supporting dynamic modular reconfiguration in general.

None of the approaches, however, indicate what the methodology should be in order to design applications targeted to the proposed systems. All the approaches introduce additional overheads in the communications protocol and system control. System designers already perceive a great amount of complexity in using partial reconfiguration [62]. The additional limitations imposed by an unclear design methodology and communications and control overheads are compounded onto those complexities. This may counter-intuitively cause system designers to perceive that using partial reconfiguration is too complex to be worth its benefits.

Researchers are now becoming more aware of this fact that the application must first be considered. A very recent proposal by Hagemeyer et al. [31] includes a design methodology for communications infrastructures. However, it does not address the core problem of how an application can be designed for the system. The resulting communications infrastructure is a bus network laid out in a homogeneous fashion across the area allocated for partial reconfiguration, as shown in Figure 2.17. At present the infrastructure is targeted for general use and as yet does not show any indication of application-level optimization. Not only do the communications and control overheads still exist, there is a severe side-effect of implementing this fine-grained homogeneous infrastructure: signals have to travel through multiple short wires over many switchbox hops, or have to use the chip-length three-state wires (which, again, are not present in Virtex-4 and Virtex-5 FPGAs). Apart from being much finer grained, the infrastructure is extremely similar to that proposed by Kalte et al. [43] and thus shares the same limitations.

### 2.3.11 The Virtex-4 FPGA Family

In 2004 Xilinx introduced a new FPGA family — the Virtex-4 Platform FPGAs. Among many improvements over its predecessors the family introduced the notion of platforms (i.e., “sub-families” with resources selectively tuned

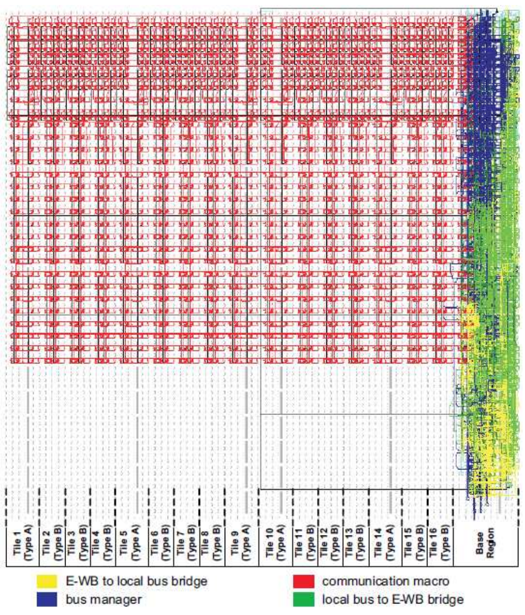


**Figure 2.17:** Homogeneous communications infrastructure (Hagemeyer et al.)

for logic, signal processing or embedded applications) and embedded DSP cores.

An important architectural improvement specific to partial reconfiguration is that Virtex-4 FPGAs are no longer configured by frames spanning the entire height of the FPGA. Instead, each frame spans 16 CLBs in height, thus every device has a height that is a multiple of 16 CLB rows. Figure 2.18 shows a layout of the 8 clock domains in the smallest device in the LX (logic) platform. Each clock domain can be reconfigured independently of the others, as can a single column of 16 CLBs anywhere in the device.

This "paged" structure can be used to lay out modules in two-dimensions. As long as no module is higher than 16 CLBs rows and the modules are aligned with the clock domain pages, each module can be reconfigured independently of any other regardless of its width. Compared to a one-

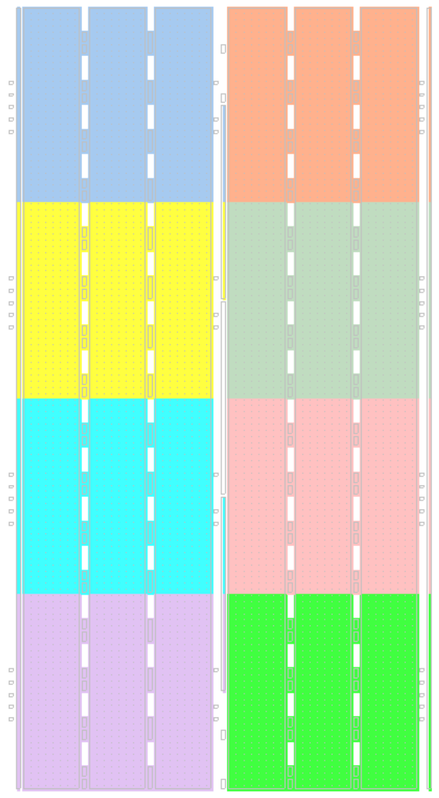


**Figure 2.18:** Pages in an XC4VLX15 device

dimensional layout of modules, shorter routing paths can be formed. To the best knowledge to the author, the work in this thesis is the first approach that targets the Virtex-4 device family and takes advantage of its two-dimensional reconfiguration architecture.

### 2.3.12 The Virtex-5 FPGA Family

The Virtex-5 FPGA family was introduced in 2006 [97]. Its many improvements to the Virtex-4 architecture included 6-input LUTs, diagonal routing and an increased maximum clock frequency of 550 MHz compared to the Virtex-4 at 500 MHz.

The reconfiguration mechanism of Virtex-5 devices is identical to that of the Virtex-4 except that each frame spans a column of 20 CLBs instead of 16. As such, with the appropriate specification of parameters, the methodology proposed in this thesis is also suited to Virtex-5 devices.

### 2.3.13 The Early-Access Partial Reconfiguration Tool Flow

Research in supporting dynamic modular reconfiguration carried out up until 2006 targeted the Virtex and Virtex-II/Pro FPGAs as XAPP290 did not allow partial reconfiguration on Virtex-4 devices. However, hints at a new tool flow with use of slice-based bus macros (rather that three-state based ones) appeared in a papers written by Xilinx — Sedcole et al. [73] and Lysaght et al. [56].

In February 2006 Xilinx, in collaboration with the U.S. Department of Defense Joint Tactical Radio System (JTRS) program, announced the availability of a prototyping-to-production kit that accelerates the implementation of software defined radio (SDR) modems [93]. The JTRS SDR kit uses partial reconfiguration with a Virtex-4 device to dynamically swap waveform processing units while other waveforms continue to be processed on the device. To date this is the first true industrial application to utilize partial reconfiguration.

In March 2006 Xilinx released a new tool flow — the Early-Access Partial Reconfiguration (EAPR) Tool Flow [95], available to a select group of customers and academic institutions.

The EAPR tool flow is an extension to the module-based XAPP290 flow, with several improvements. Firstly, bus macros are now composed of slices and can be placed anywhere on the boundary of a PRR. Three-state bus macros are no longer needed nor supported. Secondly, PRRs can be placed anywhere in the device and can be of any rectangular size and shape, with some device-specific constraints. Finally, communications via the bus macros are no longer restricted to adjacent modules. Intermodule wiring routes are now permitted to pass through PRR and non-PRR areas. This was motivated by the incentive that routing through module areas results in shorter wiring paths rather than routing around them.

Slice macros are bus macros that are composed of slices rather than three-state buffers. An example of a slice macro is depicted in Figure 2.19, which was reproduced from [95]. Each slice macro handles 8 bits of communication, and is directional. Figure 2.19 depicts a right-to-left slice macro where the module to the right of the macro, i.e., in the white area, outputs 8 bits of data to the module to the left of the macro, i.e., in the gray area.

Another form of slice macro, the wide slice macro, is depicted in Figure 2.20, also reproduced from [95]. A wide slice macro spans 3 CLBs, thus 3 wide slice macros can be nested within each other to provide a total of 24 bits of communication at any CLB on the module boundary.

The steps in the EAPR tool flow are outlined in Figure 2.21. The flow is described in detail in Section 9.2, but a short outline will be given here. The flow starts from a modular design description in HDL. At this stage the system designer must have already planned how many PRRs there will be in the system, and which modules are to be placed in which PRR. The actual area

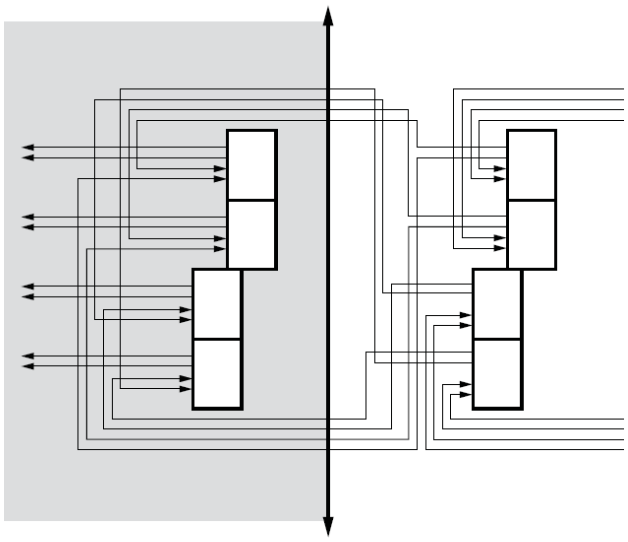


**Figure 2.19:** Right-to-left slice macro

and timing constraints are input into a constraints file in Step 2. Steps 3 and 4 implement non-partially-reconfigurable versions of the application, where each version consists of one of the possible configurations of modules in the PRRs. This is to ensure that the design meets timing and area constraints. In Step 5, the "base design", i.e., the static portions of the application, is implemented (placed-and-routed). Each PR module is implemented in Step 6, and finally the base design and the PR modules are merged into complete bitstreams representing initial configurations in Step 7. Partial bitstreams for each of the PR modules are also generated during this merge step.

Although the EAPR tool flow is documented only as a module-based tool flow, the difference-based flow is still operational, as confirmed by Xilinx representatives during a partial reconfiguration workshop [98].

These improvements were very welcome to research and development communities. However, the developer is still responsible for ensure that an

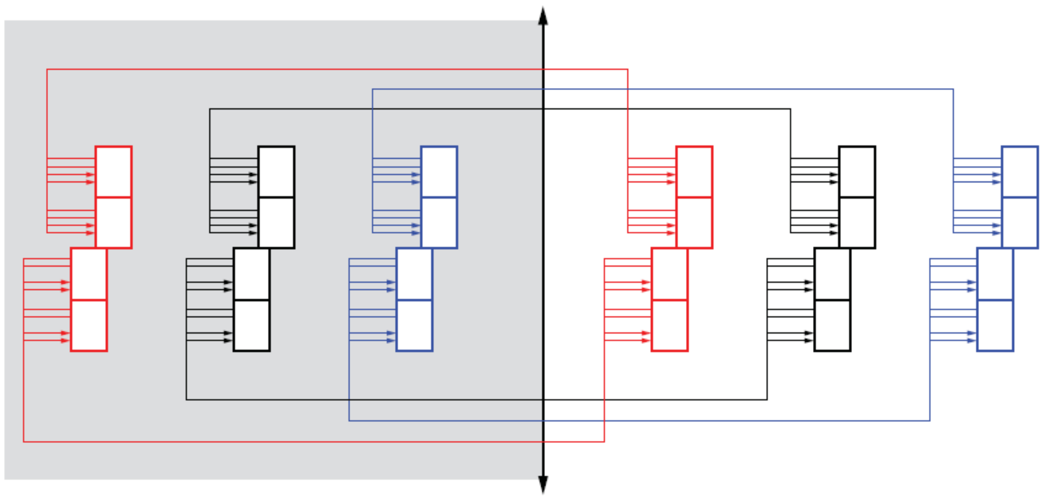


**Figure 2.20:** Nested wide slice macros

application is properly mapped to the FPGA and that the appropriate intermodule communications are provided. As such, researchers still need to resort to custom, low-level methods in combination with the EAPR tool flow in order to implement applications.

## 2.4 Conclusion

This chapter provided a background on implementing dynamically reconfigurable applications in the hardware virtualization paradigm. The review of current approaches given in this chapter raises several important points.

At the higher levels of the general methodology, current approaches to partitioning and scheduling do not generally take into consideration how the application will be mapped at the lower levels. This may cause critical path delays and reconfiguration overheads to render the application infeasible for

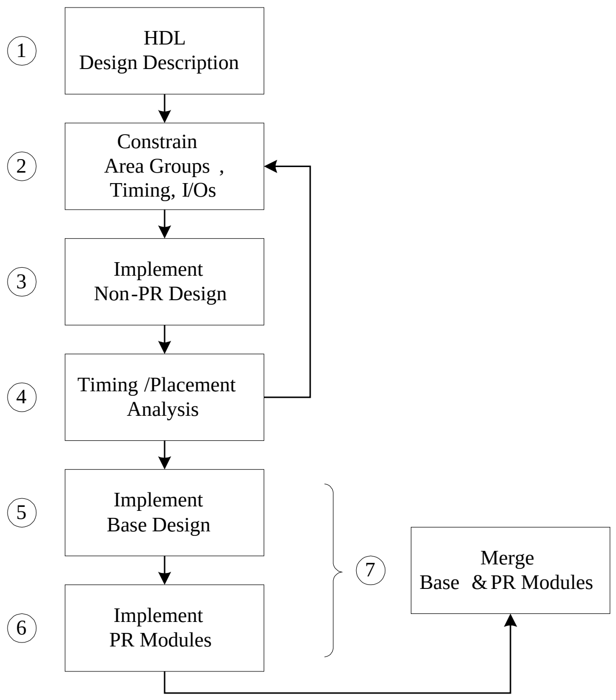


**Figure 2.21:** Steps in the EAPR tool flow, reproduced from [95]

implementation. However, these approaches can be modified to cater for a particular target device and implementation.

At the low-level implementation stages, the methods provided by Xilinx have many limitations and are insufficient to effectively support intermodule communications as is. This is expected as the aim of XAPP290 and the EAPR tool flow is to provide tool support for partial reconfiguration at the lowest FPGA implementation level, i.e., slice macros, ability to specify partial reconfiguration regions and bitstream generation. System designers will have to manually design communications infrastructures on top of that for their applications.

Current approaches to improve on XAPP290 and the EAPR tool flow have been tackled from the bottom-up. This compounds the existing overheads of partial reconfiguration on FPGAs together with communication protocol overheads, reconfiguration delay due to relocation, free-sizing and fragmentation, as well as reconfiguration control. As such, these approaches provide an entire system or platform for partial reconfiguration without demonstrating applications that can be implemented on them.

In conclusion there is a pressing need for an integrated methodology that takes the low-level implementation aspects into account throughout the design flow in order to ensure that the application can be feasibly implemented on the FPGA. As current methods have not targeted the new Virtex-4 and Virtex-5 FPGAs, it is advantageous to consider exploiting the potential advantages provided by the new reconfiguration mechanism, logic density and speed.

# Chapter 3

# COMMA: A Communications Methodology for Module-Based Dynamic Reconfiguration of FPGAs

## 3.1 Introduction

Allowing system designers to exploit dynamic reconfiguration raises many challenges. Of foremost concern is that system designers need an integrated toolset based on a top-down methodology to design and implement dynamically reconfigurable applications on an FPGA. At this time there is not yet any such integrated methodology nor toolset available that is widely accepted and in use.

Module-based design encourages reuse of proven components and encourages good hierarchical design. The module is also a natural atomic unit of reconfiguration as an application is usually decomposed into modules based on its key functions. At the highest level of abstraction, a toolset for a top-down methodology should allow a designer to decompose an application into modules that are potentially reconfigured or swapped while the system in which the FPGA is embedded is active. The toolset must also allow for the definition of module interfaces and their timing characteristics. The partitioning of the system into modules cannot be done without reference to the resources available to implement the modules, including, necessarily, the communications infrastructure available to connect the modules together.

The modules themselves may be sourced externally from IP vendors or developed in-house. This raises issues about the consistency of definitions and descriptions of interfaces and behaviors. It is difficult to adapt each module interface to a standard or proprietary protocol.

Some means of simulation must also be provided. A simulation framework suitable for dynamic reconfiguration needs to include communication and reconfiguration timing and the ability to simulate logic replacement.

Place and route tools need to operate at the module level, and assuming the required communications can be provided or codesigned, must not perform global design or optimization steps. These capabilities are implemented to some degree with module-based flows [92] and manual optimization tools [96].

The designer is also likely to want some form of run time or operating system to be generated or provided that can hide the burden of managing the device. The OS defines policies and strategies for utilizing and sharing resources, which must be fed back to the design system for it to be able to partition and adapt the implementation accordingly.

Design iteration may be needed to accommodate the constraints of the device and its run time management. Overheads should be minimized so as not to lose the benefit of hardware implementation. Moreover, the penalty for designing at a higher level of abstraction and supporting virtualization should not be large.

With respect to dynamic module placement, it is desirable that modules be relocatable in order to support schedules that are not known at design time. This places a significant burden on the communications infrastructure to connect, when required, specific IO pins with module ports whose locations are not known until run time. Inter-module connections involving dynamically placed modules may also need to be provided at run time. A conflict thus arises between the need to provide fast routes of adequate width for the sake of performance and the constraint of finding and setting these at run time, conceivably with little laxity in the task.

Another challenge researchers and designers currently face is a lack of vendor support for dynamic reconfiguration. Lack of support takes two forms: sub-optimal device architectures and inadequate tool support. While the concepts underlying support for dynamic modules in a general manner have been investigated since Brebner's Swappable Logic Units [14], little real progress has been made in developing widely applicable solutions with sufficient design tool support. With the release of new architectures [87], which give more recognition to the scalability of reconfiguration mechanisms, the time seems right to move forwards on practical and effective methods for harnessing reconfiguration at the module level.

A main concept advocated in this thesis is that effective module-based reconfiguration of FPGAs requires at its core a communications infrastructure that can support the varying communications needs of the run time configured modules. Invariably, these dynamically placed modules will need to connect with other modules on the FPGA and to the I/O pins of the device. Since the interfaces of the modules and their run time placement will in general vary, the communications infrastructure needs to support run time reconfiguration of the routing, or provide an indirect mechanism for connecting ports with pins.

This chapter presents a methodology for the deployment of a communications infrastructure that efficiently supports the communications needs of a collection of dynamic modules when these are known at design time. The methodology also provides a degree of flexibility to allow a range of unknown requirements to be met at run time.

An important feature of the methodology is that it does not constrain the module interfaces or communication protocols. The methodology provides the wiring for the raw module interfaces, i.e., the connections between the input and output ports of the modules and external I/O. A major advantage of doing so is that system designers do not have to design modules to adapt to a particular interface or communications protocol, or, viewed differently, they are free to choose whatever interface or protocol they wish. In addition, off-the-shelf components can be easily used without engineering the appropriate wrapper modules to adapt them to a particular interface or protocol.

Overheads that are incurred using a specific communications protocol such as data serialization, bus arbitration, etc., can also be avoided.

It is a much tougher problem to provide effective intermodule wiring when interfaces are unconstrained instead of conforming to a standard, fixed interface. When the module interfaces are fixed, it is simply a matter of creating a wiring infrastructure with enough wiring to connect each of these interfaces between the modules that communicate. Instead, with the COMMA methodology, the interfaces are unconstrained and the wiring infrastructure supports modules with different interfaces arriving and leaving.

The methodology described in this chapter was presented at the Dynamically Reconfigurable Systems workshop of the 2006 International Conference on Architecture of Computing Systems [49], and at the 2006 IEEE International Symposium on Field-Programmable Custom Computing Machines [50].

## 3.2 A Hierarchy of Dynamically Reconfigurable Systems

System designs that target FPGAs may consist of a static collection of subcomponents or modules. That is, the design process results in a static circuit that is configured onto the FPGA until the system needs to be reconfigured to perform an entirely different function. Such an FPGA design can be defined as static since it is not altered while the system is active, each configuration is completely defined at design time, and there is sufficient time between reconfigurations to carry out the design to completion. Design methodologies for this type of FPGA use are relatively well understood and supported by FPGA tools, although a commonly accepted module-based orientation to the design of large-scale static designs may still be lacking.

By virtue of their reconfigurability, systems including FPGAs may exhibit a degree of flexibility and dynamism. As this dynamism may take a number of forms, a hierarchy of dynamically reconfigurable systems is proposed based on the constraints imposed on said systems. This hierarchy provides a framework for subsequent discussion.

It is important to first distinguish between complete and partial reconfiguration. This thesis is concerned with supporting partial reconfiguration, in which part of an FPGA device is reconfigured to support new functionality. Moreover, dynamic reconfiguration should be supported as much as it may be permitted with commercial devices. This means allowing part of a device to be reconfigured while the system in which it is embedded, or indeed, the parts of the FPGA device that are not being reconfigured, remain active. A complete reconfiguration of the device is considered to be an incarnation of the static design case described above. Static designs are supported as a special case within the methodology.

Second, the thrust of this work is to support module-based or core-based reconfiguration, such as the replacement of one video codec with another, rather than fine-grained reconfiguration, such as the specialization of a constant coefficient multiplier. The approach is not intended to preclude use of fine-grained reconfiguration, rather the focus is on how to support modular design techniques and in particular the communications issues raised by dynamically reconfiguring design modules.

Apart from static designs, three types of dynamically reconfigurable systems, herein denoted DR1, DR2 and DR3 respectively, can be distinguished. DR1 is the most constrained of these and applies to systems in which it is known at design time which sub-component of a design may be swapped with a given (set of) alternative(s) and, crucially, which region of the placed and routed circuit is to be reconfigured with a given dynamic module (see Figure 3.1 for a diagrammatic comparison of this situation with the static case). Part of the FPGA circuit design is static and another part of the circuit is to be swapped at run time. Many examples of such systems have been reported. For example a Bluetooth baseband bitstream processor is reconfigured after processing the header packet to process the payload in [25]. In this class, the communication needs (bit- and band-widths, communication patterns, protocol (if any), etc.), and thus the interface of the module alternatives are known in advance. Connecting a dynamically placed module to the surrounding circuitry and device pins therefore involves selecting from a number of preconfigured channels. It is worth mentioning that for DR1, the triggers for reconfiguration and all possible schedules are known at design time.

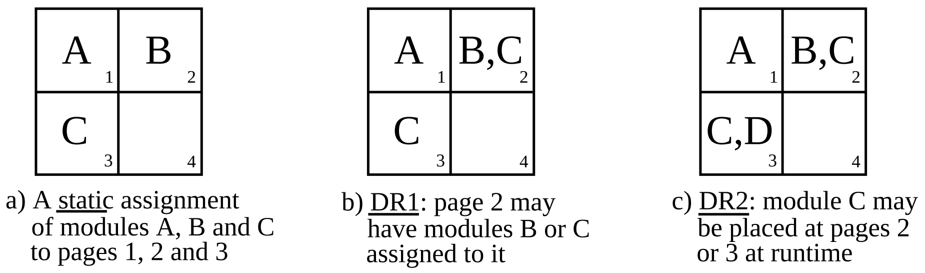

A
B
C
1
2
3
4
a) A static assignment of modules A, B and C to pages 1, 2 and 3
A
B,C
C
1
2
3
4
b) DR1: page 2 may have modules B or C assigned to it
A
B,C
C,D
1
2
3
4
c) DR2: module C may be placed at pages 2 or 3 at runtime

**Figure 3.1:** Classification of module-based reconfigurable systems

The class DR2 includes those systems for which the static components and the set of possible dynamic modules is known at design time, but their placement onto the FPGA fabric at run time is not known. A representative example is depicted in Figure 3.1(c). It is possible for this situation to occur when the order in which dynamic modules are called for at run time is not known. A possible scenario exemplifying this case is a robotic explorer that dynamically partitions real-time tasks, perhaps including vision, navigation, scientific sampling, and communications onto hardware and software subsystems according to time, area and power constraints. This situation gives rise to the need to support flexibility in the communications infrastructure in order to support the necessary interconnection between dynamic modules and other on- and off-chip components. However, given the interfaces of the dynamic components are known at design time, it is possible to engineer an inter-module communications infrastructure that supports expected requirements. Since a degree of flexibility in placing each module is required in such a communications infrastructure, this can be viewed as a step towards creating an infrastructure that supports modules with interfaces that are not known at design time. Since the methodology proposed here works with raw module interfaces, creating such an infrastructure is considerably harder than if a standard, fixed module interface is enforced.

For the sake of a complete classification, scenarios are possible in which the set of modules for which the communications needs to be supported is not known at design time. Such systems are classified as DR3 in this hierarchy. An example of such a system may be one in which modules arrive dynamically at run time, perhaps in response to a user choosing to run an application downloaded from the Internet or some other repository. It is anticipated that such a system can be supported by providing a communications infrastructure that is parameterized on the maximum expected bit- and band-widths.

## 3.3 The COMMA Approach

The COMMA approach is founded on some core principles. First, there is a need to provide practical and efficient methodologies that assist designers in effectively exploiting dynamic reconfiguration. Such methodologies should be top-down, allowing applications to be specified, partitioned and implemented. The entire flow should target a current FPGA device family and physical constraints must be taken into account at every step.

Second, such a methodology must be supported with tools that provide as much transparency as possible for designers to harness dynamic reconfiguration without having to focus on low-level details. The current EAPR tool flow [95] requires expertise with implementing dynamic reconfiguration at the device architecture level. In order for such tools to be developed the low-level device constraints must be factored into the methodology starting with the application specification.

Third, an effective communications infrastructure is paramount to efficiently exploiting dynamic reconfiguration. The COMMA approach aims to provide an infrastructure capable of efficiently supporting the communications needs of dynamic modules, and at the same time is optimized for the application requirements, the hardware, and available design-time knowledge.

The rest of this chapter presents the target device and layout, an approach for pin virtualization to allow access from any module pin to any other module pin or external I/O pin, and a description of the COMMA design flow.

## 3.4 Reference Target Device

The reference device family that is used in order to describe and demonstrate the COMMA approach is the Xilinx Virtex-4 family [87] of FPGAs. The approach takes advantage of the "paged" configuration layout of this family but is also applicable to any device that is configured in a similar way, including the newer Virtex-5 family [97].

The minimum unit of configuration is a bitstream *frame*. The frames in Virtex-4 devices are unique in that they are of a fixed-length of 41 quad-byte words, each spanning 16 CLB rows. In predecessor devices the frames span the entire height of the device. The frames are tiled about the device into "pages" that span half the width of the device and 16 CLB rows, as shown in Figure 3.2, which depicts a Virtex-4 FX12 device with 64 rows and 24 columns of CLBs (and includes one hard PowerPC core, depicted as a white rectangle in the figure). This is the smallest device available in the FX family.

The fastest configuration interface for the Virtex-4 FPGA family is the SelectMAP interface running at 100MHz in 32-bit mode [89]. Each frame thus takes about $0.41\mu s$ to be loaded onto the device. Each column of 16 CLBs consists of between 22 and 23 frames and takes about $9\mu s$ to load.

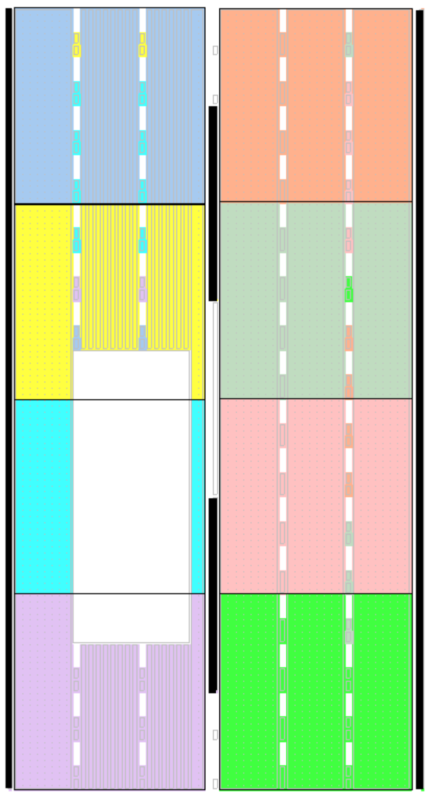


**Figure 3.2:** XC4VFX12 device with pages

This is a considerable improvement over the previous Virtex-II Pro device family, which can take from $20\mu s$ for the smallest device to $135\mu s$ for the largest device to configure a column of CLBs. Normalized, each CLB[1] takes $0.56\mu s$ to be reconfigured in a Virtex-4 device as compared to $2.5\mu s$ for a Virtex-II Pro device.

The external I/O banks are located on the left, in the middle and on the right of the device (shown in Figure 3.2 as black bars). This contrasts with predecessor families, in which I/O pins are distributed around the periphery

[1]In terms of logic capacity, CLBs in Virtex-4 and Virtex-II Pro devices both consist of four slices each, and each slice consists of two 4-input LUTs and two flip-flops.

of the device. The locations of external I/O blocks are important when considering how the communications infrastructure should connect to them.

The next section introduces the module and wiring infrastructure layout for the COMMA approach. The layout is targeted to the characteristics of the Virtex-4 architecture described above.

## 3.5 Module Placement Strategy

### 3.5.1 Paged Module Placement

Since each frame spans 16 CLBs vertically and the minimum unit of reconfiguration is one frame, it can be seen that any of the 8 pages shown in Figure 3.2 can be reconfigured independently of every other. Were modules mapped to these pages, they could be dynamically swapped while other modules are left running.

The COMMA approach proposes that each page should accommodate one reconfigurable hardware module. While it is possible to reconfigure less than a page, the overheads (such as placement and defragmentation) of micro-managing modules (e.g. arbitrary placement akin to [43]) are likely to be unacceptable and unnecessary. However, there may be reconfiguration delay savings if the module size can be reduced.

Another advantage to this approach is that each clock region on the device corresponds to the pages shown in Figure 3.2, thus each module can be clocked independently, and with the new support for dynamic reconfiguration of functional blocks [89], Digital Clock Managers (DCMs) [87] may be dynamically reconfigured to adjust the clock frequency in each page independently.

An important characteristic to note about Virtex-4 devices is that the resources in each page may be different. For example, the XC4VFX12 device shown in Figure 3.2 has a large PowerPC processor block in three of the pages on the left side of the device. In addition, the XtremeDSP signal processor elements [91] are located only on the right half of an XC4VFX12 device. The implications of this heterogeneous layout is that it may not be possible to relocate a module bitstream, placed-and-routed for a particular page, to another using address modification techniques such as REPLICA by Kalte et al. [41][39]. It may theoretically be possible to relocate modules by modifying the bitstream if non-similar resources are not used. However, the modification does still incur a run time overhead. This effect also extends to hard placed-and-route cores, i.e., hard macros. Hard macros can only be placed in pages where all its constituent resources present and are located in the appropriate relative positions.

### 3.5.2 Page Aggregation

In order to allow for better flexibility in terms of module sizing, this placement strategy also opens up possibilities of placing a module in two or more adjacent pages such that larger modules can be accommodated.

Since the center column effectively divides the FPGA resources into two halves it is preferable to aggregate pages vertically instead of horizontally. The ratio of the number of rows to the columns in Virtex-4 devices is always at least 2 and thus it is natural to do so. This arrangement also allows the vertical carry chains implemented between CLBs to be expanded.

### 3.5.3 Horizontal Page Division

Larger Virtex-4 devices can be very wide, with a horizontal width of up to 116 CLBs in the largest LX device. As the device size scales it is conceivable that each page may be overly large for a single module. Since the minimum unit of reconfiguration is one frame it is possible to divide each page into sub-pages at the granularity of 1 CLB. This subdivides wide pages in large devices and may be desired if it is known that there will be very small modules available and that some space savings are necessary.

The sample module placement in Figure 3.3 shows modules M1 and M2 being placed in a divided page, M3 in three aggregated pages and M4 in a page of its own.

It should be noted that division and aggregation does not need to be permanent. It is possible to place a module utilizing the entire page that M1 and M2 takes up when they are removed. Similarly it is possible to place a smaller module into any of the three pages that M3 occupies when it is removed.

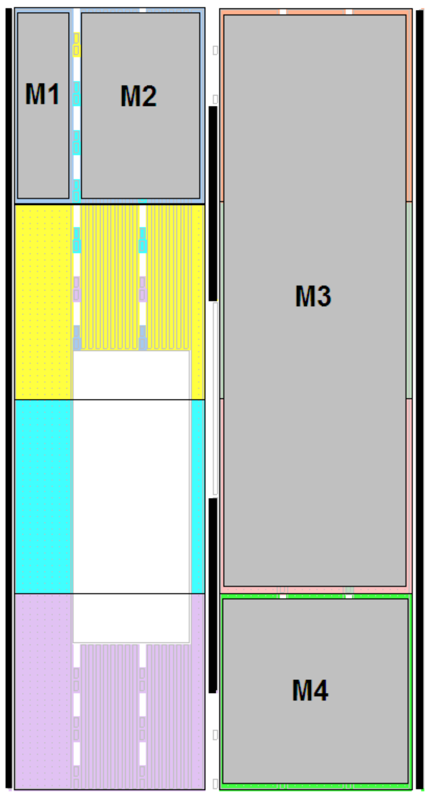


**Figure 3.3:** Page aggregation and division

## 3.6 Pin Virtualization

### 3.6.1 Physical Communications Infrastructure

As discussed in Section 2.2.4, bus-based and NoC-based approaches for implementing intermodule communications limit access to module and and external I/O pins. This limits bandwidth, placement flexibility, and restricts implementation to particular devices and/or platforms. In contrast, the COMMA approach allows the point-to-point virtualization of any module and external I/O pin in the fabric. This allows any module pin to access any other module's pins and any external I/O pin.

The physical communications infrastructure consists of module *slots*, a *wiring harness* connecting the modules to other modules and IOBs, and *slice macros* [95] [73] connecting the slots to the wiring harness. Modules are dynamically placed in these *slots*, and are surrounded by the *wiring harness* which provides the connections to any other slot and to external I/O. The area for the wiring harness envelops the external I/O pins used by the system for communication.

This is an implementation of Brebner's "parallel harness with wiring" SLU-based virtual hardware model [14]. Brebner's original depiction is reproduced in Figure 3.4.

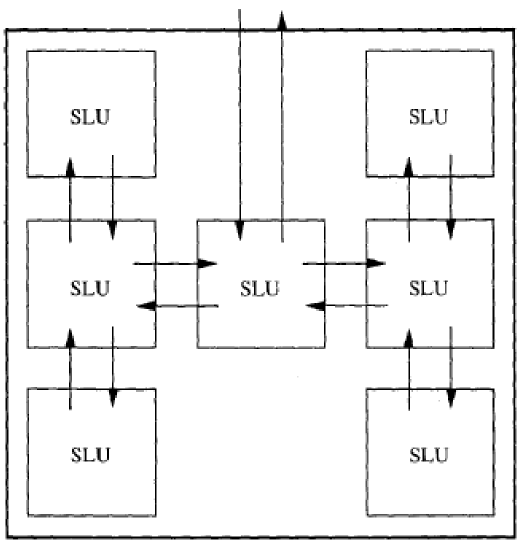


**Figure 3.4:** "Parallel harness with extra H-tree wiring supplied" from Brebner [14]

The stylized layout of a possible maximal configuration of a COMMA communications infrastructure is shown in Figure 3.5. The grey area, where the wiring harness resides, envelops every external IOB. Each of the six white module "slots" shown in the figure occupies a little less than one page and is connected to the infrastructure via LUT-based slice macros [73] such as those used in the EAPR tool flow (see Figure 3.6 below). A module can be placed in one or more contiguous slots. Note that the number of and the size of the slice macros in the figures in this subsection are not representative of actual implementation scenarios, and are for illustrative purposes only.

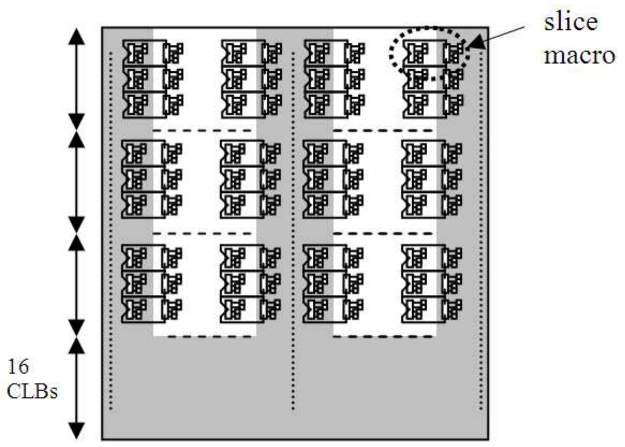


**Figure 3.5:** Trident layout

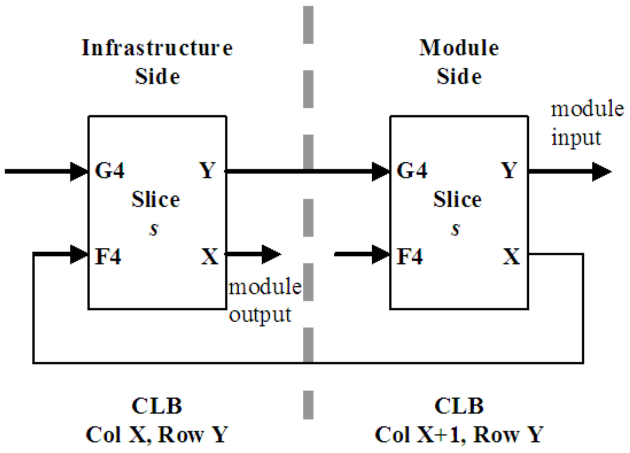


**Figure 3.6:** Slice macro connector

The "trident" layout, so-called because of the appearance of the area reserved for the wiring harness, has homogeneous slots, thus allowing for simple means of relocation utilizing major-address modification techniques such as REPLICA [41]. However, the critical path length from a pin at the top left of the layout to the top right corner of the arrangement is long, i.e., from the top left corner, down to the bottom where the large wiring area exists, then moving to the right of the device, and up again to the top right corner.

Virtex-4 slice macros similar to those provided in the EAPR toolkit (see Figure 3.6) are used in order to provide connection to the communications channels that are routed via the wiring harness. However, the COMMA methodology generates slice macros that have any combination of inputs and outputs (up to 8 bits) for two vertically or horizontally adjoining CLBs, and are of arbitrary width. The EAPR toolkit provides 8-bit (single) macros and 24-bit (triple-wide) macros. The COMMA methodology generates macros of any width, which implies that any bitwidth is possible subject to space constraints.

The “double-ring” layout depicted in Figure 3.7 overcomes the critical delay of the “trident” layout at the cost of slightly greater relocation complexity.

This layout provides better routing opportunities but three different types of module placement slots now exist (the two at the top, the four in the middle and the two at the bottom). As reconfiguration in Virtex-II (Pro) and Virtex-4 devices is glitchless [73], reconfiguring modules in the top and bottom module areas where part of the page is the wiring infrastructure should not cause device or circuit malfunction. If reconfiguration is said to be glitchless, this indicates that if a particular portion of the configuration

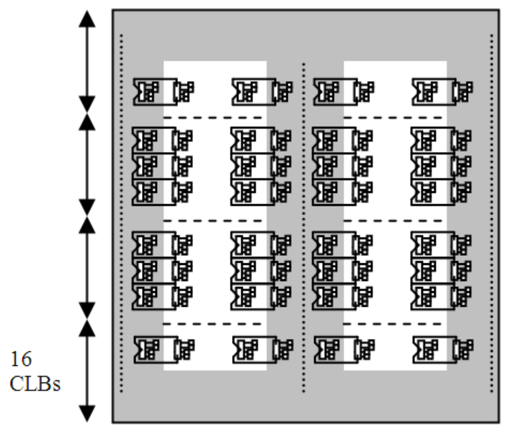


**Figure 3.7:** Double-ring layout

data is overwritten with the same configuration data as it currently holds, and if that portion is currently running, no signal glitches will occur and the behavior of that portion of the device will be as if the reconfiguration never occurred. The EAPR tool flow [95] ensures that if the module boundaries are properly defined, the configuration data for the portion of the frames outside the module boundaries will be the same.

However, modules can only be relocated to slots of the same type for which they are synthesized for, e.g., modules synthesized for any of the middle two slots can be relocated to any other of the middle two slots, but not to the top or bottom slots. If it is necessary to "relocate" modules to slots of different types, one module bitstream needs to be synthesized for each different type of slot.

Another alternative, the "ribbed" layout, is depicted in Figure 3.8. Additional horizontal routing channels are available in this layout, thereby having better routability and possibly lower critical path delays. However, each page now has a single, separated slot, which inhibits vertical page aggrega-

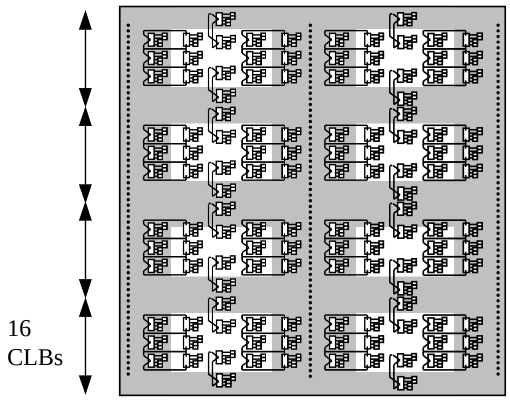


**Figure 3.8:** Ribbed layout

tion. This disadvantage can be ameliorated by judicious module sizing at the higher levels of the design flow, e.g., at the behavioral specification and partitioning levels. Larger modules can be split into smaller ones that fit into each slot. On the other hand, smaller modules can be aggregated to fully utilize the resources in each slot.

The layouts presented here illustrate some possibilities and their relative merits. However, it is important to note that these observations simply imply that the COMMA methodology does not specify a fixed layout for the communications infrastructure. The systems designer can optimize the layout for the needs of the application. If not all the external I/O pins are required, the communications infrastructure need not envelop the pins that are not required. If it is known that the modules in the application are generally of a particular size, the page layout can be catered to that size to reduce the necessity to aggregate or divide pages. The amount of resources allocated for the wiring infrastructure can be judiciously determined based on the sizes of the modules and their communication needs. The principles guiding these layouts are to take advantage of the paged reconfiguration mechanism and thus should be exploited by the system designer as much as possible. Also note that the layout alternatives presented here are targeted at the LX15/FX12 devices in the Virtex-4 family. These are the smallest devices in the Virtex-4 range and only support 8 modules. The number of slots supported by the range of devices in the Virtex-4 LX range and their slots sizes, assuming a ribbed layout with a wiring infrastructure border of 2 CLBs surrounding each slot, are shown in Table 3.1. If a different amount of resources are allocated for the wiring infrastructure, the slot sizes will be different but the number of slots should remain the same.

| Device Code | CLB Array | Number of Slots | Slot Size (CLBs) | Reserved for Wiring (%) |
|---|---|---|---|---|
| XC4VLX15 | 64 × 24 | 8 | 96 | 50.0% |
| XC4VLX25 | 96 × 28 | 12 | 120 | 46.4% |
| XC4VLX40 | 128 × 36 | 16 | 168 | 41.7% |
| XC4VLX60 | 128 × 52 | 16 | 264 | 36.5% |
| XC4VLX80 | 160 × 56 | 20 | 288 | 35.7% |
| XC4VLX100 | 192 × 64 | 24 | 336 | 34.4% |
| XC4VLX160 | 192 × 88 | 24 | 480 | 31.8% |
| XC4VLX200 | 192 × 116 | 24 | 648 | 30.2% |

**Table 3.1:** Number of slots and slot sizes for a ribbed layout with a border of 2 CLBs per slot

The percentages of CLB resources reserved for wiring are shown in the rightmost column of Table 3.1. The relative amount of reserved wiring resources may seem high in smaller devices but this is because the ribbed layout is optimized for short routing paths. The system designer has the freedom to tailor the wiring area to their required optimization goals.

### 3.6.2 Reconfigurable Data Ports

Each module placement slot is capable of connecting a large number of bits to the wiring harness. Each slice macro occupying one CLB on the boundary of the module area can connect 8 module I/O pins to the wiring area. As one of the aims of dynamic reconfiguration is to allow different modules to be placed in the same slot, a method must be set in place to map the dynamically-changing number and type of module I/O pins to the communications infrastructure.

The COMMA approach introduces Reconfigurable Data Ports (RDPs) as a means of mapping module ports to slice macros. In Figure 3.9 the two output ports destined for module M2 are mapped to the one 8-bit RDP (because they can be routed together) whilst the 3-bit port has its own 3-bit RDP.

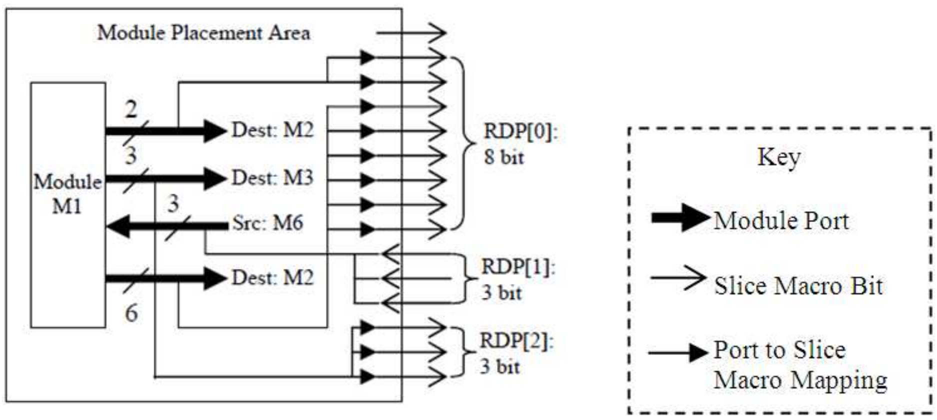


**Figure 3.9:** Reconfigurable data ports

The definition of RDPs allows communication channels to be set up in the infrastructure. Channels of varying bitwidths can be defined to ease routing complexity (it is less complex to route a group of bits because there are fewer possible destinations than if each bit were individually routed).

RDPs should be defined after the placement of each module is known. At the implementation level, wrappers can be generated for the modules to direct the module ports to the appropriate slice macros. As these wrappers simply map the original bits of module communication to slice macro inputs or outputs, they should not consume any logic area.

### 3.6.3 Implementation and Optimization

The communications infrastructure can be optimized depending on how much knowledge is known a priori about the modules, and any user-imposed constraints.

For applications in the *DR3* classification, the systems designer should specify the expected amount of communications that the application requires. An appropriate number of slice macros and external I/O pins can then be allocated so that the infrastructure can support these requirements. The toolchain can then determine the optimal locations of these macros and I/O pins in order to minimize the critical path delay and/or to maximize module slot size by minimizing the wiring channel width.

Since more information is available, applications in the *DR2* class allow further optimization. Since the module interfaces are available *a priori*, the communications infrastructure can be tailored for optimal use of the module set rather than allocating more than is necessary as may occur in the *DR3* case. The size and location of the module areas and slice macros can be optimized to the needs of the modules.

In the *DR1* case, further optimizations are possible since the module placement and communications requirements are fixed. The toolset can aim to minimize the total reconfiguration time and/or critical path delay. The entire wiring infrastructure can be customized to the application.

### 3.6.4 Management Issues

In DR2 and DR3 applications, the management of modules arriving and leaving and the setup of channels and routes may be performed by on- or off-chip agents. DR1 applications do not require explicit module management as all the communications requirements are known *a priori*. However, some communications buffering may be necessary.

Essential requirements to be fulfilled by management agents, with respect to communication, are module registration and on-line placement. Modules should be registered with a central control unit as they arrive and leave. This is to allow modules to locate peer modules and external I/O. An on-line module placement and relocation unit is necessary to place modules in optimal locations as they arrive.

These agent functions may be performed by a host controller or by an embedded CPU core such as the PowerPC block in Virtex-4 FX devices.

## 3.7 The COMMA Design Flow

In order for this infrastructure to be readily implemented a design flow has to be set in place. In addition to standard tools for module logic synthesis, place-and-route and bitstream generation, tools are needed to perform two tasks — the automated generation of the communications infrastructure, and the preparation of modules (e.g. RDP definition) for placement. Figure 3.10 depicts the overall COMMA design and tool flow for communications synthesis. There are three main tools in the COMMA flow — the Configurator, the Infrastructure Generation tool and the Partial Reconfiguration Tool Flow.

### 3.7.1 Configurator

The "Configurator" uses Xilinx-supplied device information with user-supplied parameters to create a Chip and Communications Configuration (CCC) file. The user may specify a layout of the communications infrastructure and channel, slice macro and slot preferences to assist the infrastructure generation tool in optimization. The IO pad parameters are mandatory and are used to place and optimize the infrastructure. This essentially provides a high-level interface to the system designer. A screenshot of such a tool created for prototyping and demonstration purposes is shown in Figure 3.11.

### 3.7.2 Infrastructure Generation

The "Infrastructure Generation" tool analyzes the CCC file created by the Configurator and generates an optimized communications infrastructure implemented with HDL, constraints, slice macros and accessory macros. The

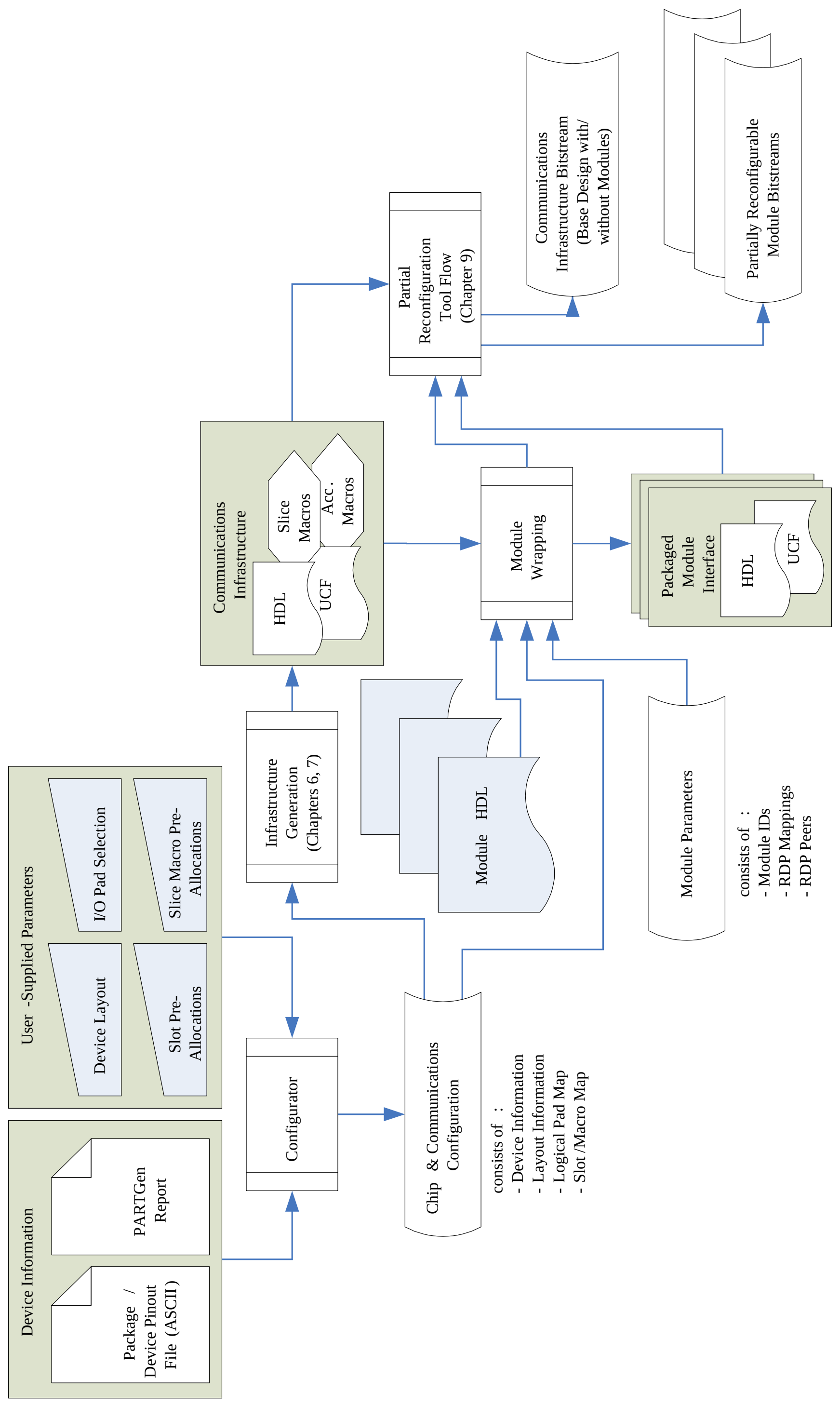
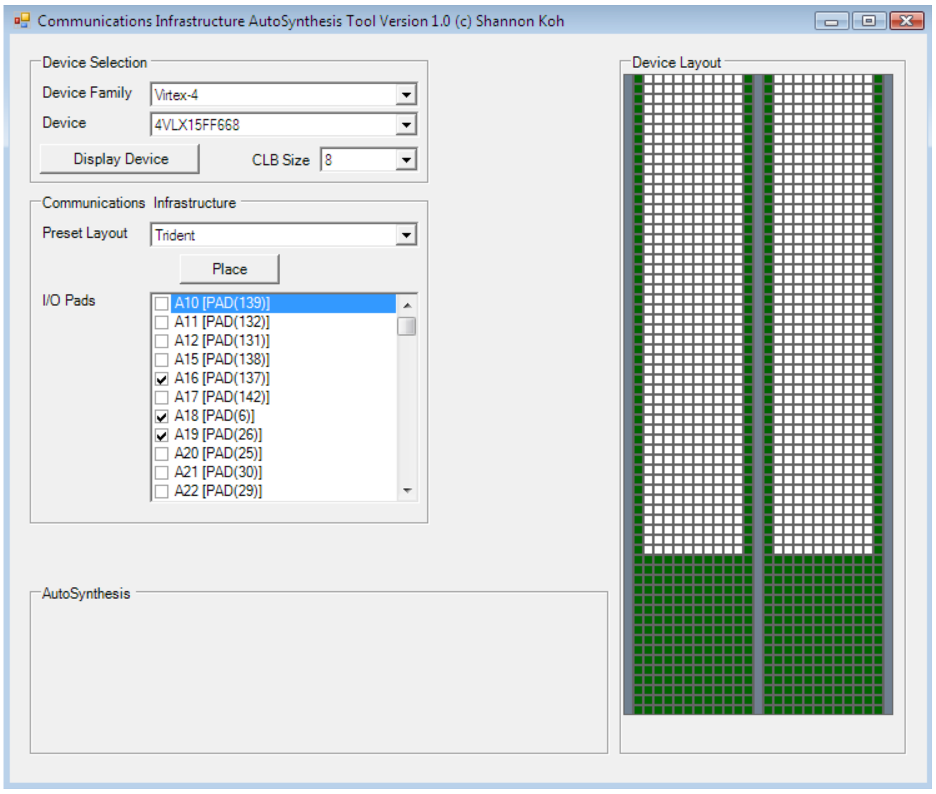

Device Information
Package / Device Pinout File (ASCII)
PARTGen Report
User -Supplied Parameters
Device Layout
I/O Pad Selection
Slot Pre-Allocations
Slice Macro Pre-Allocations
Configurator
Chip & Communications Configuration
consists of :
- Device Information
- Layout Information
- Logical Pad Map
- Slot /Macro Map
Infrastructure Generation (Chapters 6, 7)
Module HDL
Module Parameters
consists of :
- Module IDs
- RDP Mappings
- RDP Peers
Communications Infrastructure
HDL
UCF
Slice Macros
Acc. Macros
Module Wrapping
Packaged Module Interface
HDL
UCF
Partial Reconfiguration Tool Flow (Chapter 9)
Communications Infrastructure Bitstream (Base Design with/ without Modules)
Partially Reconfigurable Module Bitstreams

**Figure 3.10:** COMMA design flow


Communications Infrastructure AutoSynthesis Tool Version 1.0 (c) Shannon Koh

Device Selection

Device Family: Virtex-4

Device: 4VLX15FF668

Display Device

CLB Size: 8

Communications Infrastructure

Preset Layout: Trident

Place

I/O Pads

- [ ] A10 [PAD(139)]
- [ ] A11 [PAD(132)]
- [ ] A12 [PAD(131)]
- [ ] A15 [PAD(138)]
- [x] A16 [PAD(137)]
- [ ] A17 [PAD(142)]
- [x] A18 [PAD(6)]
- [x] A19 [PAD(26)]
- [ ] A20 [PAD(25)]
- [ ] A21 [PAD(30)]
- [ ] A22 [PAD(29)]

AutoSynthesis

Device Layout

**Figure 3.11:** Configurator screenshot

algorithmic aspects of how the communications infrastructure is generated is discussed in Chapters 6, 7 and 8. Low-level technical implementation details are discussed in Chapter 9.

### 3.7.3 Module Wrapping

Once the infrastructure has been generated, modules can be wrapped to enable placement into the infrastructure at any time thereafter (even after deployment) by using the wrapper tool to generate RDP interfaces for the modules. Details about this tool are presented in Chapter 9.

### 3.7.4 Partial Reconfiguration Tool Flow

When the infrastructure and an initial set of modules (which may be blank fillers) are available, they should be synthesized using an available partial reconfiguration flow. A custom partial reconfiguration tool flow based on XAPP290 was developed when this methodology was first proposed in December 2005 [49]. This can now be directly replaced by the EAPR tool flow that was made available in 2006 [95].

The partial module bitstreams and full infrastructure bitstreams are then generated and the system can then be deployed. The communications infrastructure bitstreams may contain an initial configuration of modules.

## 3.8 Summary

This chapter presented a module-oriented methodology to cope with design pressures and expected device scaling. Requirements for supporting dynamic reconfiguration at the module level were discussed. This includes the provision of a communications infrastructure that affords a degree of pin virtualization. The COMMA approach was presented to support communications for new tiled FPGA architectures and an overall design flow was outlined.

The rest of this thesis presents the problem that is focused on, an assessment of the overheads that are incurred in implementing the layout proposed by the COMMA methodology and details of how the communications infrastructure should be generated.

# Chapter 4

# Models and Problem Formulation

## 4.1 Introduction

This chapter details the models used and the core problem, "DR1/0", that is dealt with in this thesis, thus providing the background to the rest of the document. The DR1/0 problem is an attempt to implement applications in the DR1 class with specific assumptions.

Applications in the DR1 class have the following characteristics: the modules, their placements, and communication requirements are all known at design time. See Section 3.2 for a detailed description of the application classes. Such applications are good candidates for exploiting dynamic reconfiguration for hardware virtualization. As it is still unknown how applications in the DR2 and DR3 classes might benefit from dynamic reconfiguration, the DR1 class has been chosen. This is because even for this most constrained class of applications there is a lack of tools that help a designer input an application specification, prepare it for dynamic reconfiguration, and temporally map it to an FPGA. This thesis contributes to the overall goal of creating such tools by analyzing the effects of mapping applications to a concrete device model.

A high-level problem definition is first presented, followed by a description of the system and device model that was considered. Finally, a detailed formulation of the specific problem that is focused on is presented.

## 4.2 High-Level Problem Definition

The problem considered in this thesis is where a partially- or dynamically-reconfigurable application targeted to an FPGA is realized as a sequence of configurations. Each configuration can be represented as a graph comprising the active modules and their interconnections. Given a particular sequence, the goal is to map the graphs to the device so as to meet area and timing constraints. When area and timing constraints have been met, it is desirable to find mappings that will minimize the total time needed to reconfigure the device during an application run.

Traditional placement methods [78] encourage the compaction of module logic to maximize the utilization of available area and to minimize routing path lengths. However, due to the nature of reconfiguration mechanisms in FPGAs, doing so for reconfigurable applications may inadvertently require loading more data than is necessary during reconfiguration. Current devices have a “paged” configuration architecture, which this thesis proposes exploiting by allocating a single reconfigurable module slot to each page. Modules can thus be reconfigured independently. As for communication, complex infrastructures such as on-chip networks and bus systems impose area overheads so as to accommodate arbitration logic and reserved wiring that may not be fully utilized. This can be very inefficient in terms of bandwidth, area and frequency, and restrictive in that they require specific protocols to be implemented in every module's logic. Thus this thesis proposes to customize the inter-slot communications by building a wiring harness surrounding these slots that will support the communications needs for a sequence of configurations.

It would be most desirable to implement a single wiring harness supporting the communications needs of all application graphs, but the target device may not be large enough to accommodate the module logic with the required amount of wiring. It may also be the case that even if the wiring could be accommodated in the available device area, congestion might cause timing constraints to be exceeded. The aim is then to build harnesses that support the communications for subsequences of the communications graphs that are as large as possible without exceeding these constraints, perhaps using techniques such as maximizing wire reuse between graphs. The reconfiguration delay of a (whole) sequence of graphs will thus be split into two parts — the time to reconfigure individual modules, and the time to reconfigure the wiring harness when it is necessary to.

The DR1 problem is to decide the best module-slot placements for each graph, and to generate wiring harnesses with these module placements so as to reduce the overall reconfiguration delay (which is the sum of the reconfiguration delays for module swaps and wiring harnesses) subject to area and timing constraints. It is anticipated that judicious module placement will minimize the number of module swaps, and good placement will support wiring harnesses that will cater for longer sequences of configurations before needing reconfiguration.

Instances of DR1 in the real world include large application circuits that are mapped to small devices through the use of hardware virtualization, such as the optical flow application analyzed in Section 6.7. An application that swaps versions of modules at run time is also in this problem class since the modules are known and their placements are determined at design time, e.g., software-defined radio [79]. This problem class also includes multitasking applications on a device where the full application schedule is known, e.g., running two different codec pipelines at the same time for a handheld mobile device to decode an MPEG stream and send an encrypted message at the same time.

## 4.3 System Model

No assumptions are made about the type of reconfigurable system platform used, other than that it includes a tile-reconfigurable FPGA such as the Xilinx Virtex-4 or Virtex-5. The approach is to provide on-chip communications to and from modules and external I/O pins. As such no assumption is made on what external components are available, except some off-chip memory to contain the partial and full bitstreams to reconfigure the FPGA. The on-chip memory, i.e., Block RAM (BRAM), in current FPGAs is not sufficient to contain bitstreams of any reasonable length, thus off-chip memory or a separate host controller is required for any FPGA system that utilizes partial reconfiguration.

This thesis proposes creating a wiring harness for modules placed onto a single FPGA, but the ideas can be extended to cater to multi-FPGA systems.

It is assumed that data buffering between configurations is handled by the partitioning and scheduling process. Some examples of this are as follows:

- Modules present in a configuration at time $t$ can contain intermediate data to be used in a subsequent configuration at time $t + \delta$. As such, these modules remain in configuration $t + \delta$ as well. For example, an application with H.264 decoding performs entropy decoding and inverse quantization in a configuration at time $x$. Some intermediate data is stored in the inverse quantization module and the system reconfigures itself to a configuration at time $x + 1$ still containing the same inverse quantization module, but now with the inverse transform module instead of the entropy decoding module. The inverse quantization module feeds the intermediate data to the inverse transform module in this configuration. This approach can only be applied when there is sufficient memory available for buffering within the module logic.
- Modules communicate directly via external I/O blocks to off-chip components that buffer the data.
- I/O interface modules are added to either:
    1. Buffer any necessary data on-chip; or

2. To interface with off-chip memory to store intermediate data. Evaluative experiments described in Chapter 8 assume this approach is taken.

It is also assumed that no communication occurs to or from any slots that are in the process of being reconfigured. In addition, if the wiring harness is being reconfigured, no communication occurs at all while the reconfiguration is in progress. System control must buffer external data streams accordingly.

No particular communications protocol is assumed. However, it is assumed that the communications requirements for the application can be specified in terms of the number of directed bits required between each module.

## 4.4 Device Model

Figure 4.1 depicts an example of the FPGA layout that is advocated in this thesis. This is an instance of the "ribbed" layout introduced in Section 3.6.1. The example in Figure 4.1 is for an XC4VLX15 device, i.e., the smallest of the LX family of devices.

Two slots are allocated per 16 CLB rows, one per side (left or right half) of the device. This means that the number of slots for a device of $r$ rows is $\frac{2r}{16}$. Every slot is of the same size. The wiring harness (the white area in Figure 4.1) surrounds the slots and envelops the external I/O blocks.

The following parameters govern the size and shape of the slots and the wiring area:

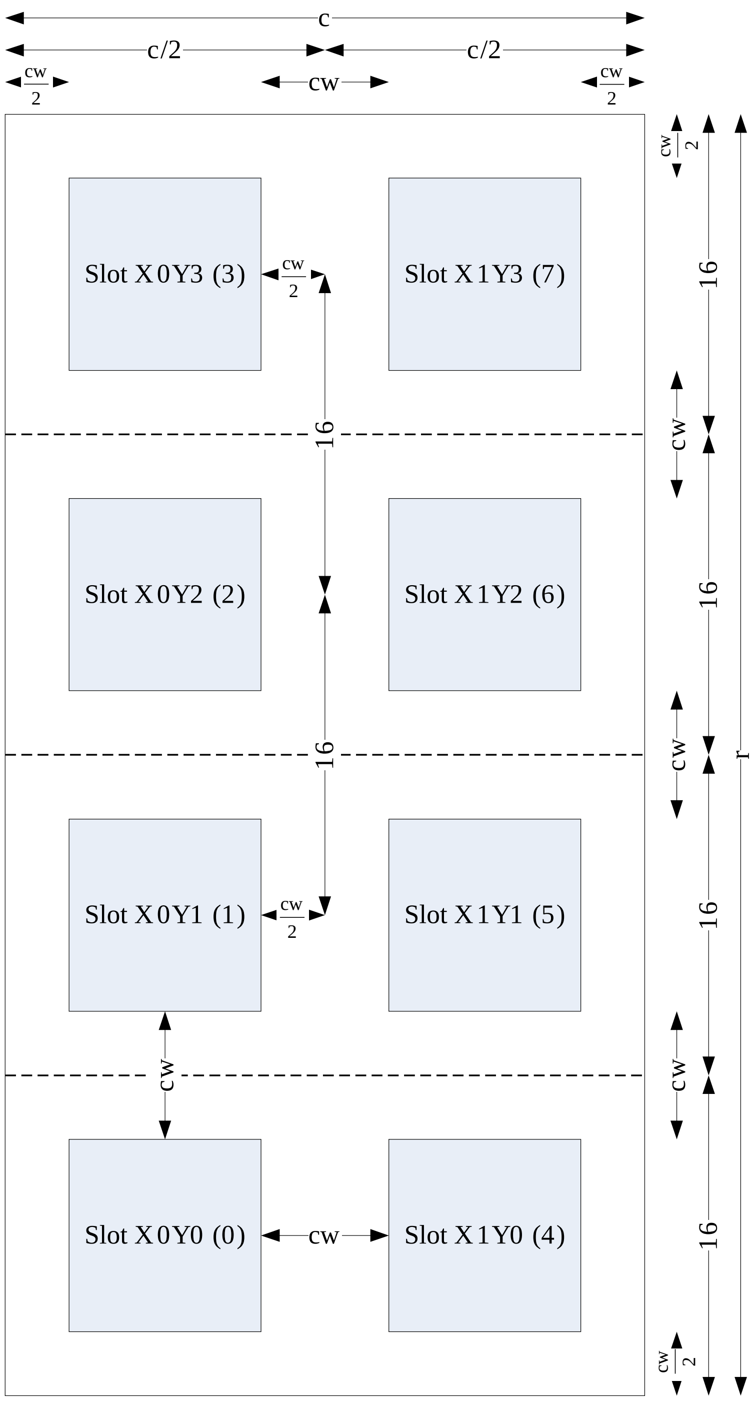


**Figure 4.1:** Device model layout

- $r$: The number of CLB rows in the device — obtained from FPGA device specifications.
- $c$: The number of CLB columns in the device — obtained from FPGA device specifications.
- $cw$: The channel width in CLBs that is reserved for wiring in the center column of the device. Viewed from another perspective a wiring channel ring of $\frac{cw}{2}$ CLB width surrounds every slot.

As shown in Figure 4.1, each slot has a fixed size. If modules are initially too large to fit into the slots, they should be divided into smaller ones. In order to maximize area utilization, smaller modules should also be clustered together up to the size of the slots. The experiments described in Chapter 8 use the clustering algorithm by Purna et al. (see Section 2.2.2) to perform this clustering process.

## 4.5 Input Specification

This section defines how the input graphs should be specified for the DR1/0 problem.

An application is represented as a sequence of graphs $\Gamma$, as depicted in Figure 4.2. Each graph is linked to another by arrows indicating their progression. Each graph $G_i$ represents an intermodule communications graph corresponding to an execution phase, or equivalently, a configuration of the application. The following timing constraints can be applied, but are optional:

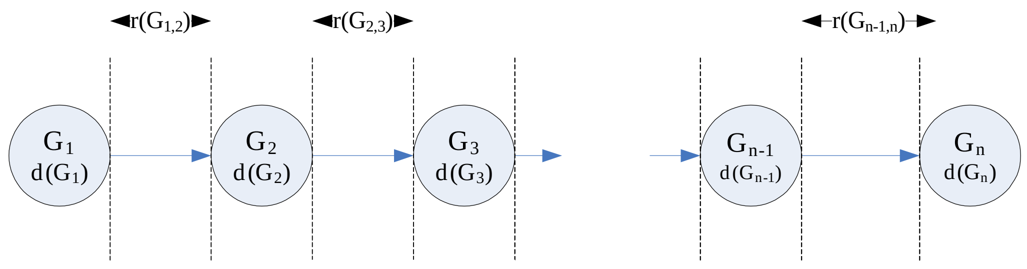


**Figure 4.2:** Sequence and timing of graphs

- $d(G_i)$: The desired maximum critical path delay, or the inverse of the minimum operating frequency of this graph. This may also be specified at the module level thus $d(G_i) = min\{d(m_j : m_j \in G_i))\}$.
- $r(G_{i,i+1})$: The maximum time to reconfigure $G_i$ to $G_{i+1}$. This can be included in the maximum total reconfiguration time $r(\Gamma) = \sum_{i=1}^{n-1} r(G_{i,i+1})$.

It should be noted that a sequence comprising a single graph is equivalent to a statically configured application.

## 4.6 Communications Infrastructure

The output of the DR1 problem is a communications infrastructure. This actually consists of a set of wiring harnesses and module-infrastructure connectors to support the communications for all the modules in the application.

The low-level communications infrastructure consists of point-to-point wiring connecting module slots to other slots and to external I/O blocks. This is implemented using two types of basic routing components, namely slice macros and wires in the general routing infrastructure of the FPGA.

This section describes the communications infrastructure from a high-level timing perspective, then provides reasons for choosing point-to-point wiring. Finally, the process of implementing the low-level communications infrastructure is presented.

### 4.6.1 Epochs and Periods

As introduced in the high-level problem specification in Section 4.2, it is best that a single wiring harness supports the communications for the entire application. However, as doing so may cause area and timing constraints to be exceeded, a sequence of wiring harnesses may have to be implemented to provide the connections needed for a single application. It is important to define the time scales over which wiring harnesses are valid on the device.

> An **epoch** is defined as a span of time within the total run time of the application for which a particular set of modules is active and communicating. The active modules and their communication requirements do not vary within an epoch. An epoch thus corresponds to a configuration of the device.

If an application description is specified as a full communications graph that is too large to fit onto the target device, the graph is partitioned into a sequence of smaller *subgraphs*, each of which do fit onto the device. The derivation of such a sequence is described in Chapter 6. Another possibility is that the application might already be modeled as a sequence of $n$ temporal configurations, in which each configuration is a *subgraph*. An *epoch* is analogous to a single communications subgraph in such a sequence.

A **period** is defined as a non-overlapping span of time encompassing one or more epochs for which a communications infrastructure in the form of a wiring harness can be built such that it satisfies the communication requirements of all the epochs it spans.

According to the definitions above the wiring harness has to reconfigured every time a period ends. As an example, the sequence depicted in Figure 4.3 is divided into two periods containing three epochs each.

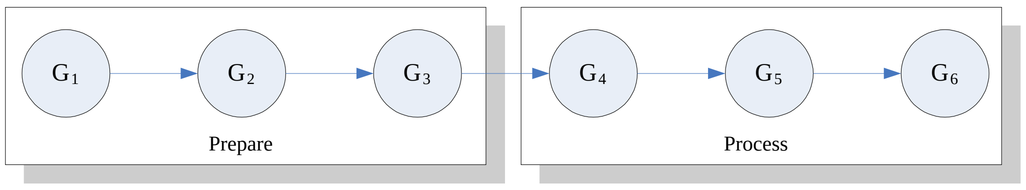


**Figure 4.3:** Epochs and periods

When the modules in $G_1$ are loaded onto the device, the wiring harness for the first period is also placed in an initial, full configuration of the device.

When the modules in $G_2$ are to be loaded, there is no necessity for the wiring harness to be reconfigured because it shares the same wiring harness as $G_1$. This applies to $G_3$ as well.

However, when the modules in $G_4$ are to be loaded, the wiring harness has to be reconfigured to the one needed for the second period.

### 4.6.2 Motivations for using Point-to-Point Connections

There are three "obvious" methods to implement a communications infrastructure that supports dynamic modular reconfiguration on an FPGA — point-to-point connections, bus systems, and networks-on-chip (NoC).

This thesis focuses on generating a customized point-to-point wiring harness for a given application specification. There are several important reasons as to why the decision was made to use point-to-point wiring rather than the other two methods. These reasons are discussed below.

#### Delay Overheads

Point-to-point connections have the minimum amount of delay overhead as compared to the other two methods. The critical path delay of the wiring only consists of the actual wiring delay and slice macros. Not only do buses and NoCs also have this delay, but introduce additional overheads as they require a specific protocol for communication. This introduces protocol, latency, and arbitration overheads. The bus width or network packet size also limit the data throughput. Finally, packet-switched NoCs cannot guarantee throughput as it is difficult to estimate exactly which routes data will take through the network as the patterns of intermodule communication are dynamic.

### Area Overheads

Due to their fine-grained reprogrammability, FPGA resources implement less logic than ASICs for the same amount of silicon area. Thus, these resources are limited and should be maximally utilized for application logic. Point-to-point connections only consume the wiring resources that are necessary. Buses require additional logic to perform arbitration and control while NoCs require logic-heavy routers spread about the device. Modules also have the overhead of additional logic required to implement the bus or NoC interfaces.

Buses and NoCs also impose a limitation on the amount of communication resources that are provided. The wiring resources are pre-allocated whether the application requires them or not. If the application requires less resources the unnecessary wires consume resources that could be used for application logic or intramodule wiring. If one module requires more resources or higher bandwidth, the entire bus system or NoC needs to be expanded to cater for it. This requires additional resources, and may require different interfaces.

### Area and Timing Constraints

Point-to-point connections have the greatest probability of meeting area and timing constraints of the application due to the lowest possible area usage and delay overheads. This is because point-to-point connections can be fully customized to the application rather than enforcing adherence to a particular protocol and communications infrastructure. If buses or NoCs are used, the application might not be implementable if the bus system or NoC do not meet the area and timing constraints. In addition, NoCs cannot guarantee timing constraints unless all the communication patterns are known *a priori*. If this is the case, a fully-customized point-to-point infrastructure will still result in lower overheads.

### Protocol

Point-to-point wiring does not enforce the adherence to any particular communications protocol. This allows more applications to be mapped as any protocol can be implemented, even bus-like or NoC-like ones. This reinforces the COMMA principle of allowing off-the-shelf cores to be used rather than restricting the set of usable cores to those that implement the interface needed for an alternate approach. Buses and NoCs have differing interface standards. Wrappers can be implemented to adhere to a protocol, but these may be difficult or impossible to implement. Modifying pre-existing cores to adapt to a particular interface is not trivial, and may perhaps be impossible if internal modifications need to be performed and the HDL is not available. Even if modification or adaptation were possible, this introduces additional undesirable area and timing overheads.

### Application Knowledge

When the application is known *a priori*, point-to-point communications are very well-suited as they can be fully customized, resulting in minimal overheads while having the highest probability of meeting area and timing constraints. With buses and NoCs the opportunities for customization are less. With buses this is limited to choosing which bus system to use and the bus-widths. With NoCs a customization is difficult because it may not be possible to provide timing guarantees.

When applications are not known *a priori* it may seem difficult to provide a point-to-point communications infrastructure that supports any type of dynamic module. Buses and NoCs have the advantage of standard interfaces where the onus is put on the module designer to adhere to the protocol rather than the communications infrastructure supporting the needs of the modules.

However, buses and NoCs allow different types of modules by simply pre-allocating the predicted amount of resources in advance. The system designer can also equivalently use the COMMA methodology with predicted communications requirements to generate an optimal point-to-point communications infrastructure. In this respect, point-to-point wiring can even be fine-tuned to the expected requirements, rather than simply using what the bus or NoC designers expect. Ultimately, if a bus system or NoC is required for some specific reason, the COMMA methodology can still be used to allocate the wiring required to implement an optimal bus-like or NoC-like infrastructure.

### 4.6.3 Drawbacks of Point-to-Point Communications

This subsection outlines the drawbacks of using point-to-point communications, and identifies siuations in which point-to-point communications may not be applicable.

#### Known Standard Interface

If the target application is known to only use modules that adhere to a specific standard interface, e.g., the AMBA bus system or a particular network-on-chip, better results may be obtained by utilizing pre-existing implementations of that particular infrastructure.

#### Less Generality

Point-to-point communications are custom-built for the target application. Thus, the resulting infrastructure may not be general enough to support modules and interfaces that are not known at design time. However, since the problem considered in this thesis is DR1, where all the modules and communications are known *a priori*, in this case this is not a significant drawback.

#### Wire Utilization

Buses and NoCs are shared resources. Assuming the temporal behavior of modules doesn't place undue demands on such shared resources, they make more efficient use of available wire resources.

## 4.7 Classes of Problems

The main identifying characteristic of DR1 are that the application, its modules and their communication requirements are known *a priori*. As this thesis deals with the problem of generating a communications infrastructure after the partitioning and scheduling stages, the application schedule is also assumed to be known. That is, we know the exact sequence of module communications graphs that we wish to implement. This section presents the classes of problems that are applicable to DR1.

The ideal graph sequence for DR1 should be a simple chain in which one graph follows another until the final graph is reached as shown in Figure 4.4.

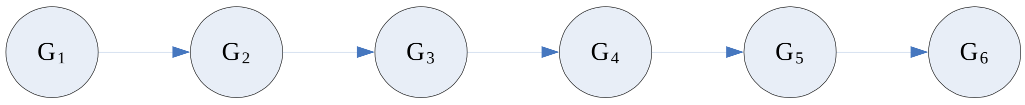


**Figure 4.4:** Standard sequence

Loops such as that in Figure 4.5 may be present in sequences where some form of iteration is required.

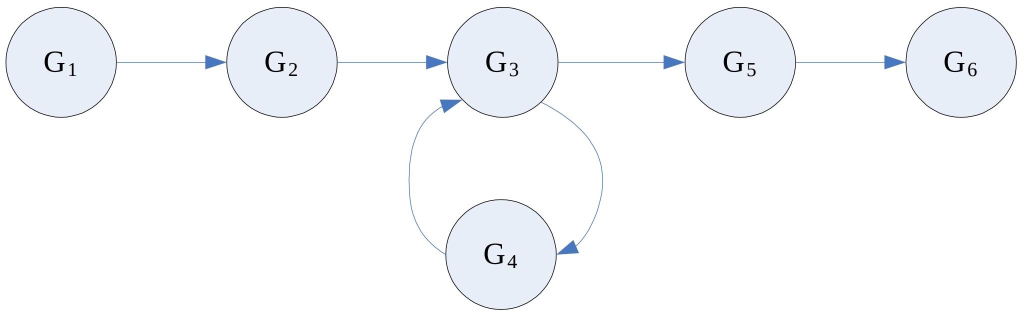


**Figure 4.5:** Sequences with loops

Significant reconfiguration overheads can occur if a period transition occurs within a loop, e.g., if $G_3$ and $G_4$ of Figure 4.5 were in different periods. This results in thrashing between wiring harnesses and thus very slow loop iterations.

Loops can be handled in two obvious ways. The first method is to unroll all the loops so the sequence will be a single chain such that it resembles the arrangement of Figure 4.4. Bondalapati has examined the mapping of loops for dynamic reconfiguration [12][13]. Although the approaches were targeted to hybrid architectures, the mapping techniques involve loop unrolling and can be applied to applications considered in this thesis. This is effective provided the exact number of iterations is known at design time.

If the application schedule is input-dependent, the number of iterations may not be known. An alternate approach can be tried in this case: place all the graphs in the loop into the same *period* such that no reconfiguration is necessary within an iteration. This approach is adopted by partitioning and scheduling algorithms that handle loops and branches, e.g., [82][9][47].

For example, the graphs $G_3$ and $G_4$ in Figure 4.5 can be targeted to be in the same period in order to try to ensure that the wiring harness does not reconfigure during the loop. Several possibilities can emerge from this. To illustrate this further, a sequence with a larger loop of 3 graphs is shown in Figure 4.6.

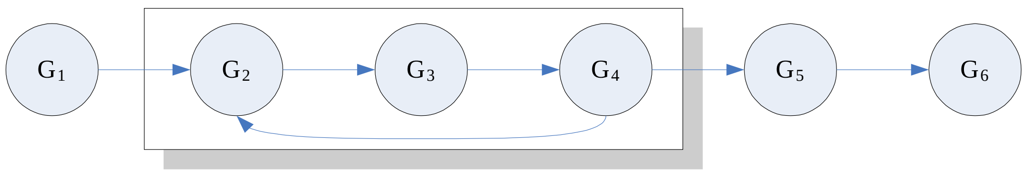


**Figure 4.6:** Sequences with larger loops

The aim here is to try to place $G_2$, $G_3$ and $G_4$ in the same period. If this were possible, then the next target should be to try to merge $G_1$ into the same period as well in order to reduce the initial overhead of reconfiguring the wiring harness when entering the loop from $G_1$. If this is not possible, $G_1$ will exist in a period of its own. The same applies for $G_5$ and $G_6$. There is also the possibility that placing $G_2$, $G_3$ and $G_4$ in the same period exceeds timing and/or area constraints. In this case, the optimization goal should be to try to ensure that the reconfiguration time between the periods encompassing $G_2$, $G_3$ and $G_4$ is minimized.

### 4.7.1 Alternatives

A sequence can include graphs where two or more sequence alternatives can exist that cannot be resolved at design time, e.g., in Figure 4.7. These can also be handled by targeting the alternatives to a single period.

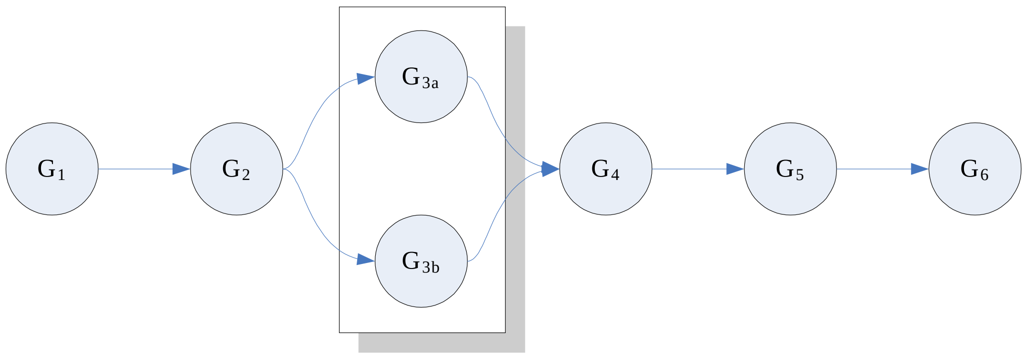


**Figure 4.7:** Sequences with alternatives

The system designer should specify a constraint to try to place $G_2$, $G_{3a}$, $G_{3b}$ and $G_4$ all in the same period in order to reduce the reconfiguration time if a "misprediction" occurs. A "misprediction" in this sense may be if $G_2$, $G_{3a}$ and $G_4$ are placed in the same period and $G_{3b}$ is in a different period. If the branch to $G_{3b}$ is more commonly taken, then the total reconfiguration time is higher compared to if $G_{3b}$ were in the same period as $G_2$ and $G_4$. The system designer has this option to apply the constraints and observe the effects as compared to allowing the infrastructure generation process to determine the best groupings. See Section 4.7.3 for a further description of these constraints.

### 4.7.2 Branching

It is possible that a user input or run time condition results in a branch to one of two entirely different sequences to be run, as depicted in Figure 4.8.

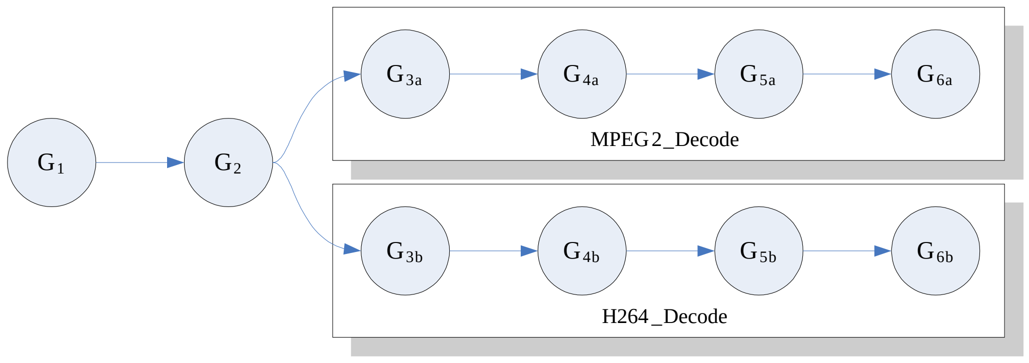


**Figure 4.8:** Branching sequences

In this case, it is best to keep graphs of each branch separate from each other. A constraint can be applied such that none of the graphs in the "MPEG2_Decode" sequence is to be placed into the same period as any graph in the "H264_Decode" sequence. It may be advantageous as well to consider preventing $G_2$ from being placed into the same period as $G_{3a}$ or $G_{3b}$ such that each of the branched sequences can be independently optimized.

An alternative is depicted in Figure 4.9, in which the wiring harness for the first few graphs in each branch is "pre-loaded" to offset the overheads of continuing execution on the correct branch and to reduce the size of the periods on either branch path.

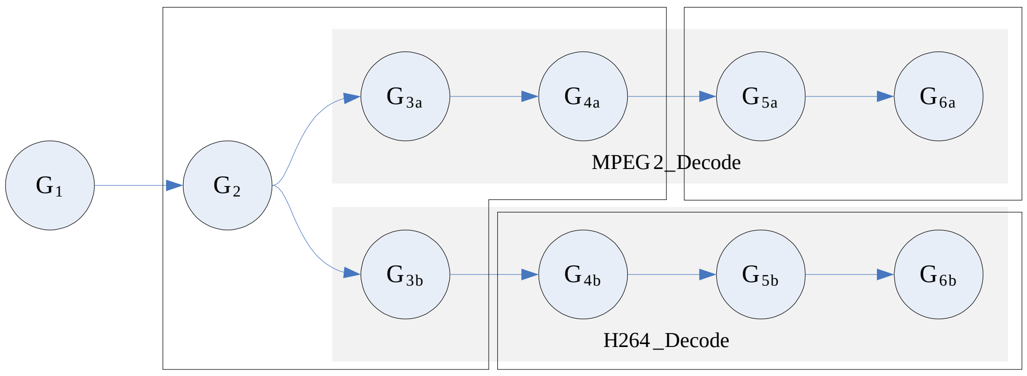


**Figure 4.9:** Preloading

### 4.7.3 Final Note

The problem classes described above suggest the use two optimization constraints to control the formation of application periods and the communications infrastructure generation process.

Firstly, the specification of a "SamePeriod" constraint can direct the infrastructure generator to try to place a specified set of graphs in the same period. This can help to prevent thrashing in loops.

Secondly, a hard "No-Merge" constraint can be applied to ensure that certain graphs should not be placed into the same period. This may help to optimize branching sequences.

As a final note regarding these problem classes, the period specifications provided in the examples above are suggestions and it is ultimately the task of the partitioning and scheduling stage to specify the constraints that best suit an application. This is similar to specifying area and timing constraints for the synthesis and place-and-route tools in a standard FPGA design flow.

If constraints are not specified, the module placement and wire harness routing stages may produce a less desirable result that satisfies the timing and area constraints. An example of this is depicted in Figure 4.10. The latter two periods consist of subgraphs where only half of the subgraphs will actually be used at any one time, i.e., only the "a" or the "b" branch will be executed. This is less optimal and involves more reconfigurations than Figure 4.8 in which the periods are optimized for each branch.

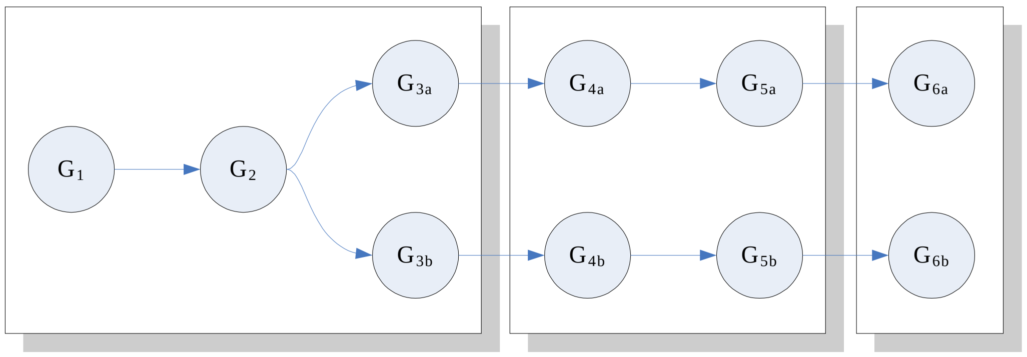


**Figure 4.10:** Unconstrained period selection

At the same time, specifying period constraints may also help in directing the module placement and wire harness routing stages to better produce solutions that satisfy timing and area constraints. An advantageous side-effect to this can also be to reduce processing time as less alternatives need to be considered.

## 4.8 Detailed Problem Specification

This section formally specifies the input and output parameters of the DR1 problem and is included here for completeness. The rest of the thesis is presented at a higher level, thus the reader may wish to skip this section and return to it if more detail is desired.

Note: The abbreviations *N*, *E*, *S*, *W*, *NE*, *NW*, *SE* and *SW* are used in the following problem specification as a set of enumerated values referring to the corresponding compass directions: north, east, south, west, north-east, north-west, south-east and south-west.

The inputs to the problem are:

- A sequence of communications subgraphs $\Gamma = \{G_1, G_2, \cdots, G_n\}$.
  - Each subgraph $G_i \in \Gamma = (V, A, P, D)$. Each subgraph represents a configuration on the device.
  - Each vertex $v \in V$ represents a clustered module that is optimized for the size of a COMMA slot, i.e., one that may be composed of smaller modules. See Sections 4.4, 6.4 and 8.2.3 for more details about clustering.
    - *Canonical Form:* $(\{graphnum \in (g \in \mathbb{N})\}, \{modnum \in (V \in \mathbb{N})\})$. E.g., (2,3) represents Module 3 in Graph $G_2$.
  - Each connection $c \in A$ the following attributes are assigned:
    - The source port number $s$ that specifies the port number of the output module. This port corresponds to the same port definitions in a VHDL entity or Verilog module although it should be enumerated.
    - The destination port number $d$ that specifies the port number of the input module.

    * The bitwidth $b$ that specifies the number of bits of this connection.
    * *Canonical Form:* $(\{graphnum \in (g \in \mathbb{N})\}$, $\{v_{src} \in (V \in \mathbb{N})\}$, $\{p_{src} \in (ports(v_{src}) \in \mathbb{N})\}$, $\{v_{dst} \in (V \in \mathbb{N})\}$, $\{p_{dst} \in (ports(v_{dst}) \in \mathbb{N})\}$, $\{b \in \mathbb{N}\})$ E.g., (2,3,1,4,6,8) refers to a connection in Graph $G_2$ from module 3 (port 1) to module 4 (port 6) that is 8-bits wide.
  - The period name $P$ indicates that all graphs with the same period name should be in the same period. $P$ may be $\emptyset$, indicating that the graph can be in any period.
  - The set of distinct graphs $D$ indicates all the graphs that should not be put into the same period as this graph. If $D = \emptyset$ this indicates that this graph can be in the same period as any other. If $D = \{i : 1 \leq i \leq n\} - P$ this specifies that this graph can only be placed in periods with the graphs in $P$.

- The device layout specified by the following:
  - The device and package information, that will provide $r$ the number of rows of CLBs, $c$ the number of columns of CLBs and all IOB (external I/O) information (names and locations).
  - The channel width $cw$ as the number of CLBs of width for the center wiring channel, or rings of $\frac{cw}{2}$ CLBs wide surrounding each slot.

The **outputs** of the problem are:

- A module placement $P_i$ for each graph $G_i$ where each $v \in V$ is given a placement $p(v) = (p_r, p_c)$. $p_r$ is the row index and $p_c$ is the column index of the module slot where the module $v$ is placed.
- Wiring harness(es), each consisting of:
  - Module-Infrastructure Connectors (MICs): The positions, macro width (a minimum of 2 for the version of the tools used in this thesis) and number of bits per direction of the slice macros on the boundaries of each slot.
    - *Canonical Form:* $(\{dir \in \{NS, EW\}\}$, $\{NEbits \in \mathbb{N}\}$, $\{SWbits \in \mathbb{N}\}$, $\{span \in \mathbb{N}\}$, $\{(slot_c, slot_r) \in (\mathbb{N}_0, \mathbb{N}_0)\}$, $\{(border,\ offset) \in (\{N, E, S, W\}, \mathbb{N}_0)\})$ e.g. (NS, 3, 2, 2, (0, 1), (N, 2)) is a 3 bit north, 2 bit south macro with a height of 2 implanted on the northern border of the slot (0, 1), at the third column from the left.
  - Wires: These are specified simply by stating the source and destination module-infrastructure connectors (MICs). If detailed routing is to be performed for the wiring harness, then each wire will also include its detailed route. Each connection $c$ in the input specification will be allocated a set of wires implementing that connection.
    - *Canonical Form:* Given $T = \{MICs \cup EIO\}$ (the set of the union of all MICs and external IOs used in this wiring harness, each wire is specified as: $(src \in T, dst \in T)$ where $src$ is the source MIC/EIO and $dst$ is the destination MIC/EIO. EIOs

(external I/Os) are specified as $e \in \mathbb{N}$ where $e$ is a pad number listed in the device package and pinout specification table for a particular device and package.

    * *Canonical form for detailed routes:* A detailed route is specified as a sequence of switchbox hops. Each routing point on a switch box is known as a Programmable Interconnect Point (PIP). COMMA uses the Xilinx Design Language PIP naming terminology database. The Xilinx Design Language (XDL) is an undocumented text-based format for describing a circuit mapped onto an FPGA. A detailed route with $r$ hops is specified in COMMA as:
      $[(src\_pip_1, dst\_pip_1) \in (XDLPips), \cdots, (src\_pip_r, dst\_pip_r)]$.

  - PortAlloc: These specify which module ports are connected to which MICs.

    * *Canonical Form:* For each graph $G_i$ and $\forall c \in A$ and for each bit in $c$, the following must be specified:
      $(\{src\_mic \in MICs,$
      $src\_dir \in \{N, S, E, W\},$
      $src\_micbit \in \mathbb{N}_0\}),$
      $(\{dst\_mic \in MICs,$
      $dst\_dir \in \{N, S, E, W\},$
      $dst\_micbit \in \mathbb{N}_0\}).$

- The module placement and wiring harness(es) will be used to derive the following:

- RDPs: Reconfigurable Data Ports implemented as wrappers around each module $v \in V$ for each graph $G_i$. These consist of:
  - **Straight connections** from module I/O bit to MIC bits if one module I/O bit is mapped to exactly one MIC bit.
  - When a module exists in the same location in different configurations (different epochs) but with the same wiring harness (same period), the following may be added:
    - **Multiplexers** from several module *output* bits to a single MIC bit. When two or more *output* bits of a module share the same physical MIC bit in different configurations, a multiplexer is placed to direct the correct module output bit to the MIC bit. The multiplexer selection inputs are changed as part of the reconfiguration process between epochs.
    - **Multiplexers** from several MIC bits to a single module *input* bit. When two or more MIC bits feed data into the same module input bit in different configurations, a multiplexer is placed to direct the MIC bit to the correct module input bit. Again, the multiplexer selection inputs are changed as part of the reconfiguration process.

# Chapter 5

# Assessing the Fixed Wiring Harness

## 5.1 Introduction

The concept of using a fixed wiring harness to implement a communications infrastructure for modular FPGA reconfiguration was introduced in the previous chapter. It was proposed that this wiring harness is to be laid out within a regular module placement structure. Since traditional FPGA design flows encourage the compaction of logic to minimize routing delays, imposing this regular wiring structure may incur some critical path delay overheads. This chapter describes an analysis performed to assess the penalties that are incurred by such a harness. The contents of this chapter were presented at the 2006 IEEE International Conference of Field-Programmable Technology [51].

## 5.2 Experimental Method

The aim of this analysis is to assess the critical path delay overheads of using a regular module and wiring layout relative to those incurred when the placer is not constrained. The experiments performed involved mapping synthetic communications graphs generated with varying parameters (such as communications density and number of modules) to a Virtex-4 FPGA. The placement of the modules and wiring were constrained according to COMMA principles (see Chapter 3) and the device model used in this thesis (see Chapter 4). Each graph mapped to the device could be viewed as a single epoch or one of the representative graphs in a period, since all graphs in a period use the one wiring harness. This section defines the problem of mapping a single graph to the device and details the experimental method used.

### 5.2.1 Problem Definition

The **inputs** to the problem are a communications graph and the device layout as shown in Figure 5.1. Each node in the communications graph represents a module or external I/O and each arc an intermodule or external connection. Arcs can be weighted according to the bitwidth of the connection. The device layout is specified using $c$ and $r$ as the number of CLB columns and rows in the device, and the channel width $cw$, representing the number of CLB columns/rows occupied by the routing channel surrounding the module slots. This is the same layout as specified in Section 4.4, reproduced here for convenience.

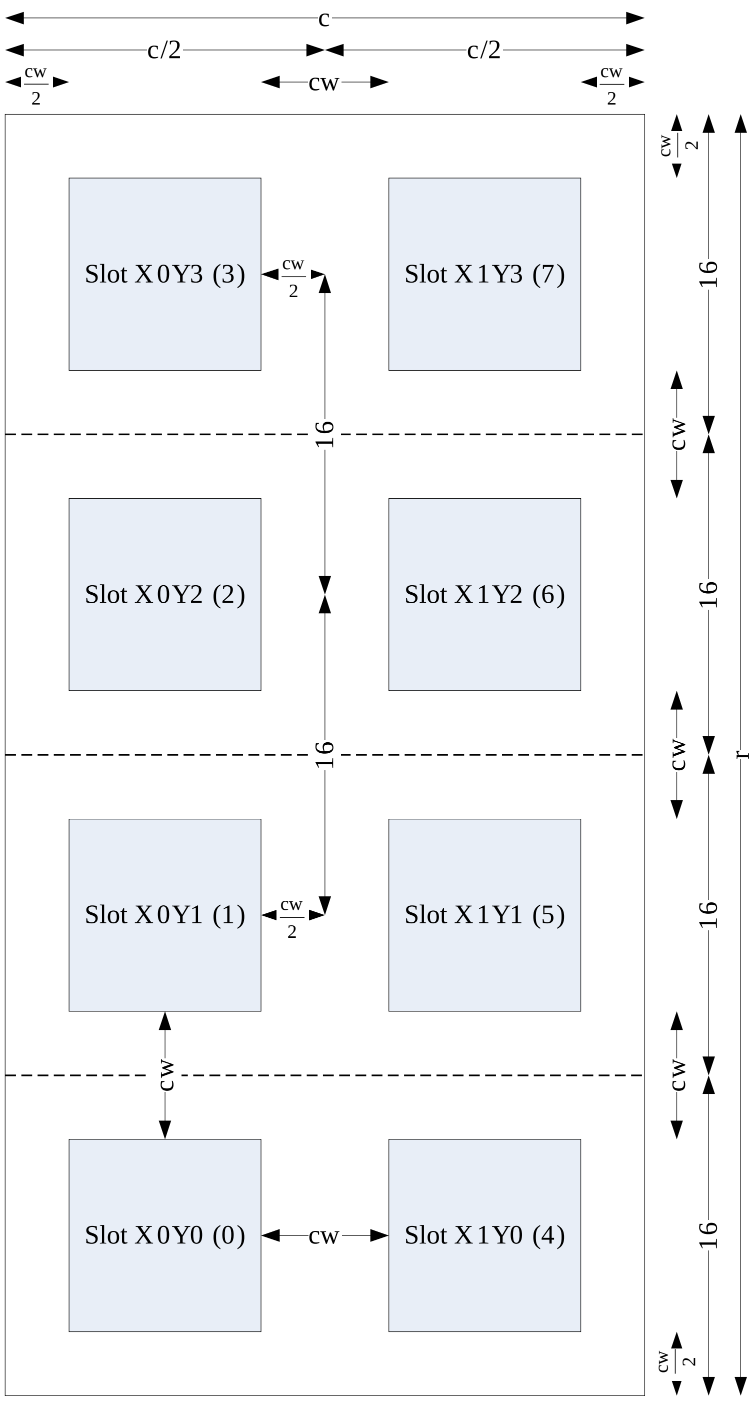


**Figure 5.1:** Device model layout

The **output** is a placement of the modules into slots such that:

- The wire availability given by the device layout and wire resource model is not exceeded.
- The distances between communicating modules are minimized in order to minimize critical path delay.
- The maximum width of any cut in wiring is minimized so as to reduce wiring channel congestion and to assist in reducing the critical path delay. Reducing congestion reduces the critical path delay as shorter wire segments can be avoided, thus reducing the number of switchbox hops necessary to implement connections.

The output is represented as a mapping from modules to slots, and a mapping from each arc in the aggregated communications graph to a set of wire resources.

### 5.2.2 Implementation Flow

The implementation flow for the experiments performed is depicted in Figure 5.2.

The first step uses an integer linear program (ILP) formulation of a problem instance to assign modules to slots while trying to minimize the total channel cutwidth and wire lengths using approximations of channel capacity. The output of this process is a heuristically-determined allocation of modules to slots, and indications of where slice macros are best placed.

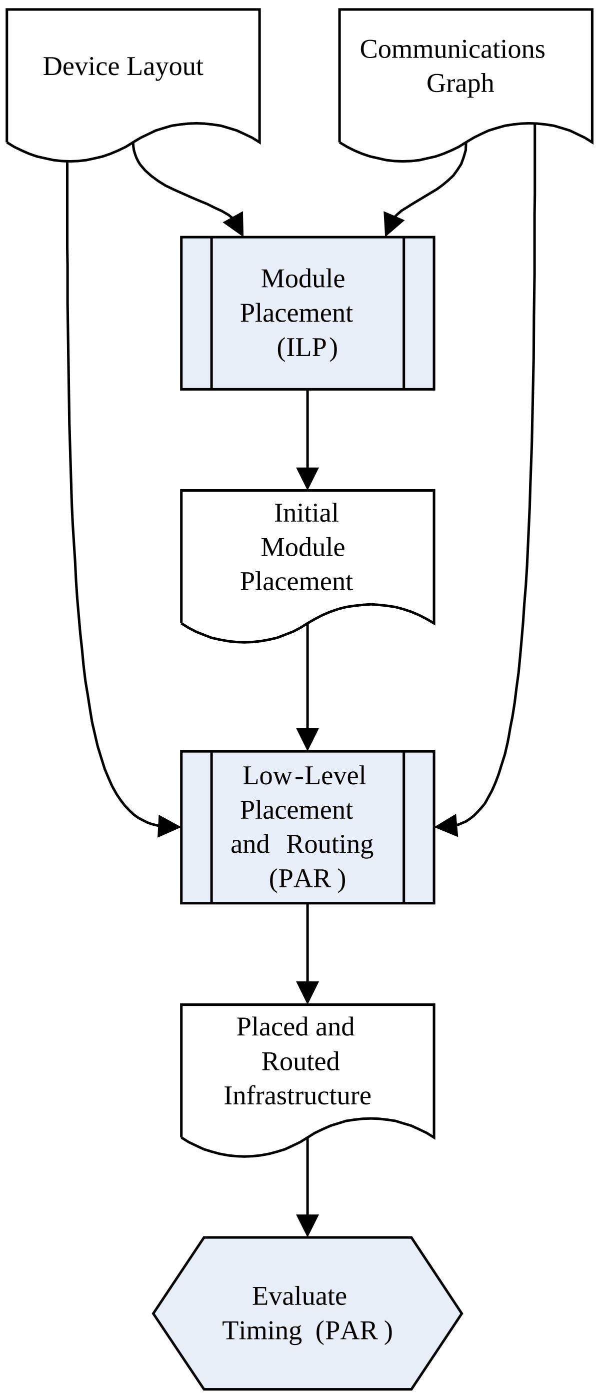


**Figure 5.2:** Experimental method flow

The next step, routing, involves selecting a suitable set of wires to implement the connections between the placed modules. At this point, the slice macros that are on the module boundaries will be placed. The output of this process is a placed and routed wiring harness in circuit description format. Xilinx uses the Native Circuit Description (NCD) and Xilinx Design Language (XDL) formats. Both of these formats are not publicly documented.

Finally, the critical path delay of the wiring harness is determined. This is obtained from the detailed timing results gathered from the place-and-route tool.

Each of the steps in this flow are detailed in the following subsections below.

### 5.2.3 Module Placement

The placement of the modules is performed by an ILP applied twice as suggested by Fekete et al. in [26]. The goal of the first ILP is to minimize the maximum number $B$ of wires interconnecting nonadjacent slots crossing each horizontal dashed line in Figure 5.1. In the second step, a placement is sought that minimizes the maximum wire distance subject to the constraint that no more than $B$ wires in total occupy the vertical channels between adjacent rows of module slots.

A heuristic module placement is thus obtained without regard to the exact wiring resources present within the channels. The subsequent routing phase of the implementation is therefore relied upon to optimize the choice of wiring resources to implement each connection.

The channel width bound $B$ determined during the first step is used to verify that the aggregated communications graph can be mapped to the device given the wiring layout constraint $cw$. Alternatively, the magnitude of $cw$ can be fixed following this step, but that may involve repartitioning and re-clustering the modules higher up in the flow.

There are several differences between the formulation adopted here and that of [26], which is targetted at a one-dimensional array of slots interconnected by a bus. With reference to Figure 5.1, wires in this model leave from or arrive at the midpoints along the edges of module boundaries. In contrast to [26], external I/O is also modeled using IOB pin locations obtained from Xilinx package files. Connections to vertically or horizontally adjacent modules are not included in the cost of a solution as these are assumed to make use of resources directly spanning the gap between the modules. If these are vertically adjacent, they do not therefore contribute to the wires running along the vertical channels on either side of, or in the center of the device. On the other hand, if they are horizontally adjacent, then these wires use resources that run orthogonally to the vertical wires, which are in short supply. The distance between nonadjacent slots $(X_i, Y_j)$ and $(X_k, Y_l)$ is taken to be $cw + 16\,|l - j|$ irrespective of the values for $i$ and $k$. These distances are then weighted by the arc weights. Each connection to an IOB is measured by the Manhattan distance between the center of each slot and the actual IOB grid location.

This placement produces two outputs that are essential for the routing stage to be performed. The first output is the set of slot-module allocations. In performing the placement the heuristic also determines which border (i.e., north, east, south or west of the slot) each wire should exit from and enter into. This second output, an exit and entry border per wire, is used to determine the slice macro positions during routing.

### 5.2.4 Routing

The flowchart of Figure 5.3 depicts the flow of processes in the routing phase, commencing with the analysis of the communications graph and module placement and finishing with circuit implementation. In relation to Figure 3.10, all these steps are carried out within the “Partial Reconfiguration Tool Flow” process. See Chapter 9 for a detailed description of the low-level processes in the methodology. The steps are discussed in more detail below.

#### Slice Macro Generation and Placement

In this step, custom slice macros are generated and placed on the slot boundaries based on the connections existing between the module slots and the wiring harness. These macros are akin to those used in the Early Access Partial Reconfiguration flow [95] but with several additional features. First, they may be of unlimited width (macros in [95] can only span 1 or 3 CLBs). They may also be vertical (to take advantage of vertically-adjacent slots), or horizontal, and may be of any combination of 1-8 northbound/eastbound and 1-8 southbound/westbound bits.

Macro placement is performed in three steps. First, a suitable edge for placing the slice macro is chosen based on the approximation obtained from the placement stage. When the placement stage estimates the wire length

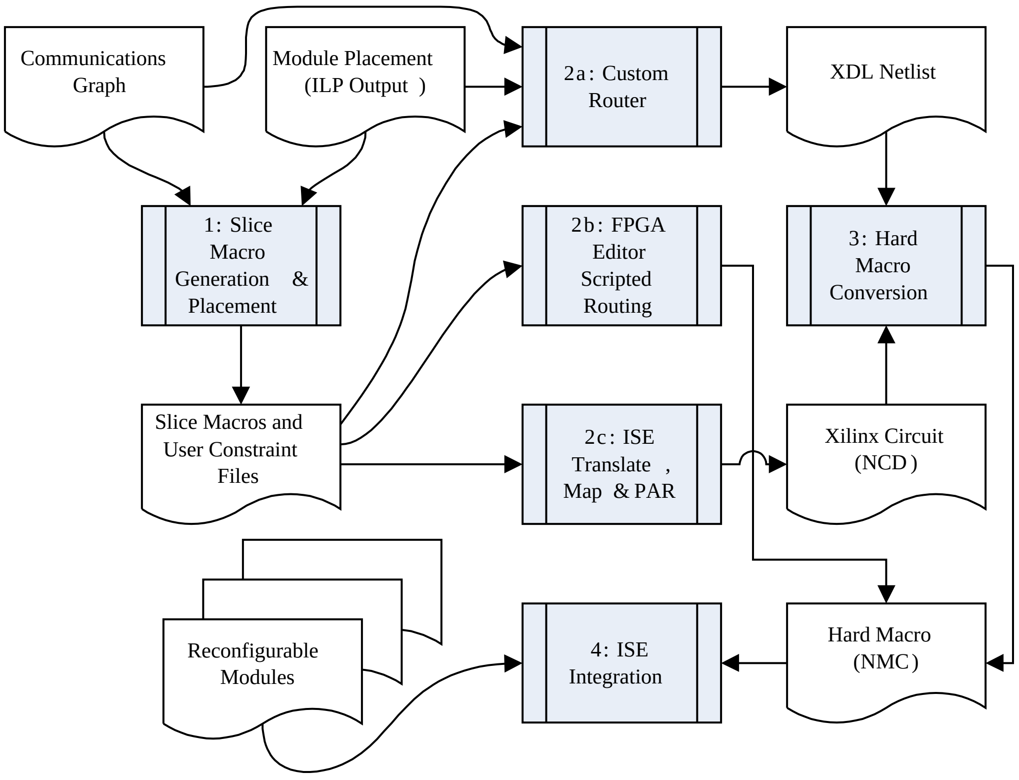

Communications Graph
Module Placement (ILP Output )
2a: Custom Router
XDL Netlist
1: Slice Macro Generation & Placement
2b: FPGA Editor Scripted Routing
3: Hard Macro Conversion
Slice Macros and User Constraint Files
2c: ISE Translate , Map & PAR
Xilinx Circuit (NCD)
Reconfigurable Modules
4: ISE Integration
Hard Macro (NMC)

**Figure 5.3:** Steps to routing the wiring harness

the initial direction of the wire is the assumed exit boundary, i.e., north, east, south or west. If there is still room on the assumed boundary for the slice macro to be placed, the slice macro will be placed there. Otherwise, the next best boundary based on the distance from the sink of the wire is chosen.

Second, an approximate location on the edge is chosen. If the communicating slots are adjacent, then it is located towards the middle of the edge, otherwise it is located towards the corner closest to its target.

Finally, the macro widths are determined by their approximate locations. To minimize interference with other macros, macros located towards the middle of an edge will have larger widths, whereas the widths of macros placed closer towards the slot corners taper off. This concept is depicted in Figure 5.4.

A limit must be imposed on the number of slice macros that can be placed around a slot. The available slot area for module logic is thus decreased by the number of slices taken up by slice macros within the slot.

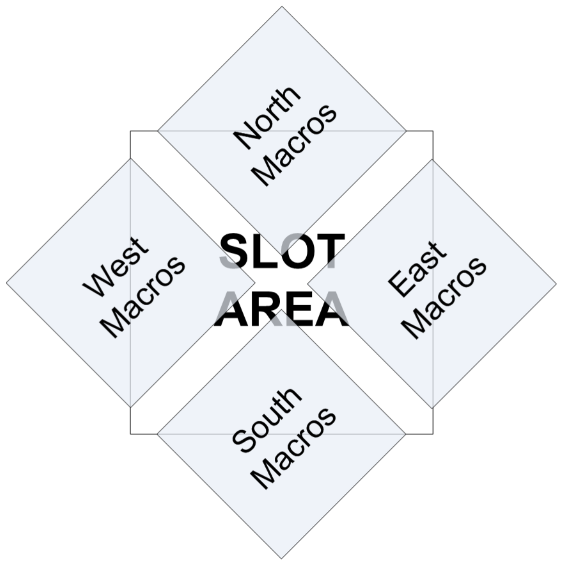


**Figure 5.4:** Macro placement

### Wire Placement

The flowchart in Figure 5.3 refers to three alternative methods for selecting and placing the wires to create a wiring harness. Method **2a** involves using a custom router to determine the best selection of wires, then using a wiring database and tool package such as COMMAPath (a tool that is currently being developed) to place the wires and to configure the harness as a hard macro. The efficacy of this method is solely dependent upon the quality of the custom routing algorithm.

Method **2b** involves automatically generating an FPGA editor script to route each individual pair of slice macro pins. This is performed by the "route" command in the FPGA editor of the Xilinx ISE toolset. This command selects the best route based on the minimum Manhattan distance, and is a good balance between full custom routing and routing that is completely automated. However, this may not scale well as the resulting overall delay may not be as good as when an iterative routing algorithm is used.

Method **2c** uses the ISE place-and-route tool with an automatically-generated HDL wiring harness using slice macro locations as specified in the user constraints file. Using the EAPR tool flow, this method does not prevent wires from passing through slots, but this only sacrifices some routing resources in the slots and limits module relocation. Module relocation is not supported by Xilinx nor has the research community developed tools for the Virtex-4 family to relocate modules by modifying bitstreams. This method guarantees that the delay of the wiring harness is as good as any Xilinx PAR output without having to design and use a custom router. However, up until and including the EAPR implementation for ISE 8.2i it was also possible to enforce a constraint that restricted wires from passing through the slots. Given the option to enforce this constraint, a system designer can evaluate the results obtained with and without wires passing through slots and choose the one that produces the best results, e.g., for area, timing, or reconfiguration delay.

#### Hard Macro Conversion and ISE Integration

The generated wiring harness from the wire placement step is integrated into ISE as a hard macro. If Method **2a** is used for wire placement, the XDL description is converted to a hard macro (using "xdl -xdl2ncd"). If Method **2b** is used, the FPGA editor circuit is saved as a hard macro. This is integrated into ISE as a Relatively Placed Macro (RPM) placed in the "base" module together with the modules. If Method **2c** is used, the circuit does not need to be converted to a hard macro as it forms part of the base design in the EAPR tool flow [95]. See Chapter 9 for a description of the integration of this process into the EAPR tool flow.

## 5.3 Results

Placing modules according to COMMA principles may result in long and unavoidable paths to IOBs and other modules, whereas running a standard ISE flow to generate a flattened netlist allows for logic replication and freedom of placement that leads to shorter paths.

In order to assess the potential overhead of the COMMA methodology an automated testbench was created. This was targeted to the Virtex-4 LX-15 with 8 slots organized as in Figure 5.1 to place-and-route synthetic task graphs with the following parameters:

- **Number of Modules:** 2, 4, 6 and 8.
- **Number of Arcs:** 10%, 20%, 40% and 80% of the full module interconnection pattern ($n \times (n-1)$ directed edges (arcs)).

- **External I/O:** 25%, 50%, 75% and 100% of modules having an external input from IOBs, and likewise for external outputs.
- **Arc Weights:** Arc weights were randomly chosen from a uniform distribution of the set { 2, 4, 8, 16 }

The channel width was not varied in these experiments because the ILP method attempts to minimize the cut width of the wires. If a design does not fit into the channel width, it is too complex to be mapped onto the device and it is rejected. On the other hand, reducing the channel width may lead to higher congestion, and thus longer critical path delays.

A co-located set of IOBs was chosen for each external connection, but the sets were randomly distributed about the IOB space. 10 graphs were generated per parameter combination and the results were averaged. Without considering the parameter combinations that did not generate feasible graphs (e.g., 10% arcs for 2 modules, which implies that there should be "0.2" edges), over 2,500 graph implementations were assessed.

Note: Comparisons are made in this section between the results of using the standard ISE tool flow (henceforth referred to as "ISE") and using the COMMA flow (henceforth referred to as "COMMA"). In the standard ISE tool flow the communications graphs are implemented as flattened netlists, i.e., without area constraints. In the COMMA tool flow the modules and slice macros are placed according to the COMMA approach as described in Section 5.2.

As per Section 5.2.3, each task graph was processed by the ILP with an exhaustive solver built in-house. The graph and placement was then input to a VHDL generator to generate and place the slice macros as per Section 5.2.4, and to produce VHDL code corresponding to the module structure with thin modules composed of a bank of XOR gates. The VHDL code was then synthesized and placed-and-routed using method **2c**. Method **2c** was chosen as a fair comparison since the ISE PAR tool is used for both the flattened and the COMMA implementations. The maximum pin-to-pin delay, which corresponds to the maximum delay of the wiring harness and the reciprocal of the maximum circuit operating frequency was then obtained for both ISE and COMMA.

### 5.3.1 Critical Path Delay

A noticeable feature of the raw data is that ISE usually obtained a lower critical path delay, but that sometimes COMMA did better. This is to be expected with heuristic placement and routing tools, but makes it difficult to find reliable predictive indicators of relative performance. It was found that the best predictive indicator was the total number of wires (interconnection density) and thus the results are generally presented relative to this metric.

For the range of module numbers explored, Figure 5.5 illustrates the average increase in the critical path delay of COMMA relative to ISE with respect to percentage internal connectivity between the modules. The plots suggest that the overheads when mapping graphs to the COMMA device layout, as compared to being unconstrained, are relatively high for low interconnectivity and decrease as interconnectivity increases.

Figure 5.6 plots contour maps of the average critical path delay for both ISE and COMMA as the number of wires in the graph and the COMMA

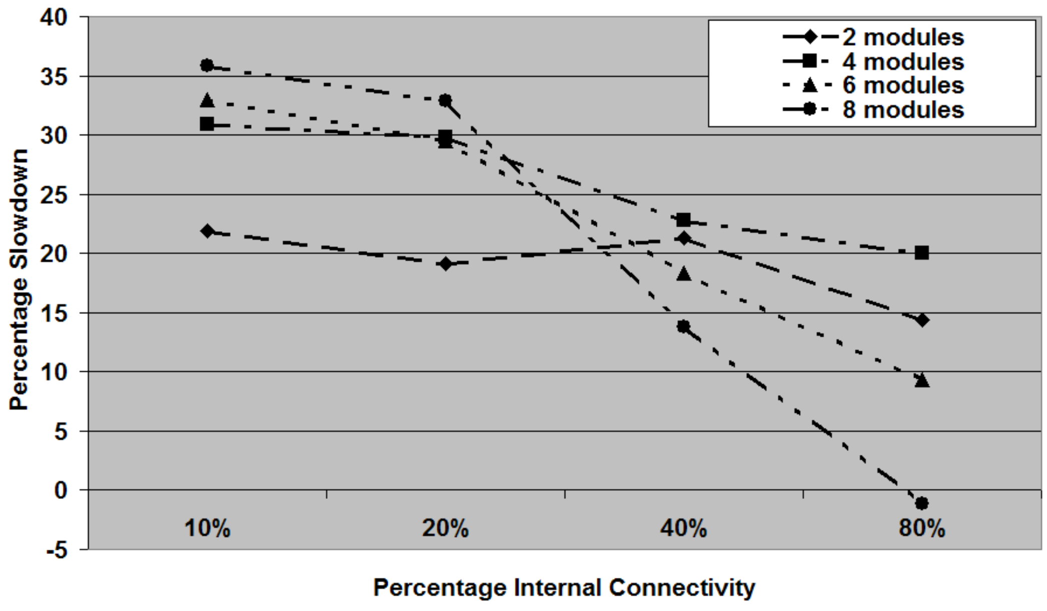
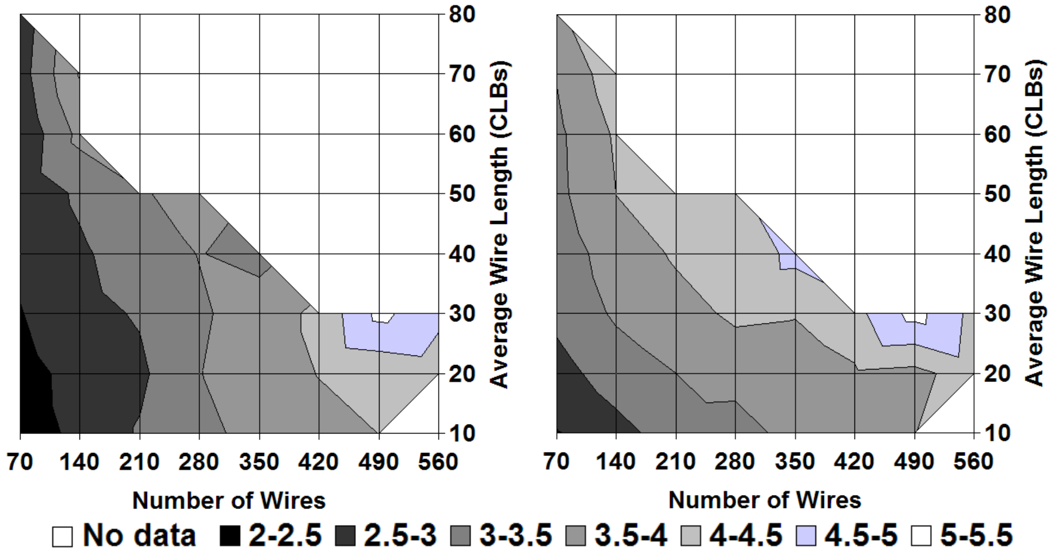


**Figure 5.5:** COMMA vs ISE critical path slowdown

average wire length varied. The contour plots were obtained by averaging the critical path delays for all graphs lying within half a grid interval of each grid intersection and setting the intersection point to that value. For example, the intersection point at 210 wires and an average wire length of 30 contains the averaged critical path delay for graphs with 175 to 245 wires and an average wire length of 25 to 35. The white region in the top-right half of each plot corresponds to a region for which data was not obtained. The contour maps indicate that the peak delay in both cases occurs for an average wire length of $30 \pm 5$ CLBs and $490 \pm 35$ wires (5.134 ns average critical path delay for ISE vs 5.174 ns for COMMA – $<$ 1% overhead). The largest difference in delays was found at $40 \pm 5$ CLBs and $350 \pm 35$ wires, where the average ISE delay of 3.344 ns compared with 4.651 ns for COMMA, i.e., an overhead of 39%. COMMA performed best at $20 \pm 5$ CLBs and $490 \pm 35$ wires, where ISE incurs a 4.132 ns average delay and COMMA a 3.856 ns average delay, i.e., 7% improvement over ISE. Averaging the overhead over the regions of the contour map the data was obtained for, an average slowdown of 12.5% for COMMA was determined. This is a remarkably good result considering the benefit of the COMMA approach is that modules are easily replaced, whereas in ISE module logic is distributed by virtue of the flattening of the netlist.



(a) ISE critical path delays (ns) (b) COMMA critical path delays (ns)

**Figure 5.6:** Critical path delay contours for ISE & COMMA.

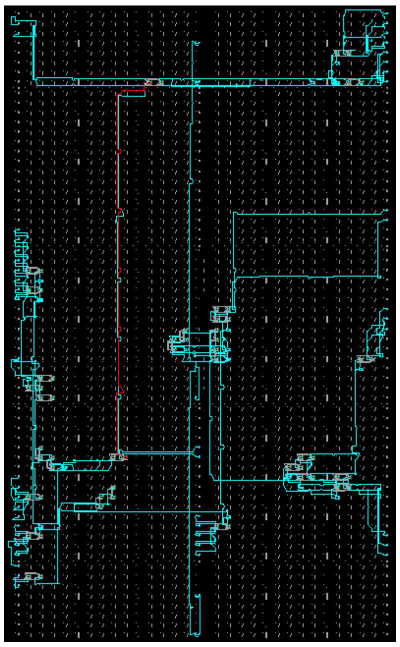
The maps also indicate that the COMMA wire delays increase more gradually than those obtained with ISE as the interconnection density was increased. This explains the relative improvement with higher interconnectivity observed in Figure 5.5. The reason for ISE's generally better performance is due to ISE distributing the module logic over the entire device and placing slice macros close to IOBs. On the other hand, COMMA is constrained to place slice macros at module boundaries and may need to connect these to distant IOBs. While COMMA has relatively fixed overheads due to its structural constraints, ISE finds it increasingly difficult to replicate logic, find good placements and route around their locations as the interconnection density rises. As circuits get dense, the COMMA overheads are absorbed.

The overheads from laying out the modules using the COMMA methodology are due to two reasons. First, the randomly distributed IOB sets for each external connection occasionally forced COMMA to implement long wiring paths irrespective of its slot location while ISE was free to place module pins at any location within the device, and in particular, close to the connecting IOBs. Figure 5.7 illustrates such an example. Figure 5.7(a) shows the ISE circuit and Figure 5.7(b) highlights the critical path for ISE. Likewise, Figure 5.7(c) shows the COMMA circuit and Figure 5.7(d) the critical path for COMMA. The overhead here was significant (2.235 ns critical path delay for ISE vs 4.992 ns for COMMA).

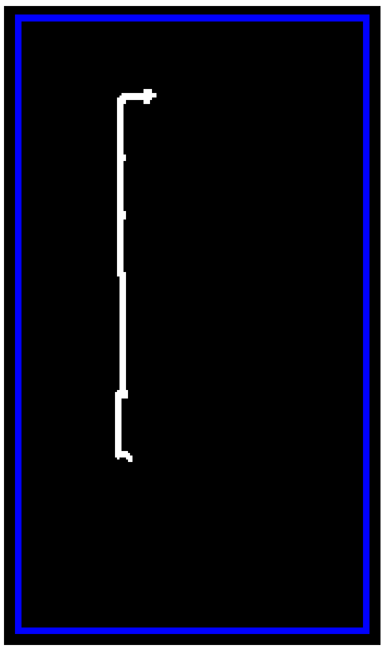
As shown in Figure 5.7(d), the longest path for COMMA connects a module at slot X1Y0 to an IOB at the top left of the device. This is a graph with 8 modules, low internal connectivity (10%) and high total external connectivity[1] (150%). All the slots were used and the module had no choice but to be placed at slot $X_1, Y_0$.

The second reason for the overheads is that COMMA was unduly penalized by the choice of module logic, which was comprised of XOR gates connecting inputs to outputs. For such simple modules, ISE was relatively free to use the entire device to replicate logic, to place it where profitable, and to route the required connections. More complex module logic would have

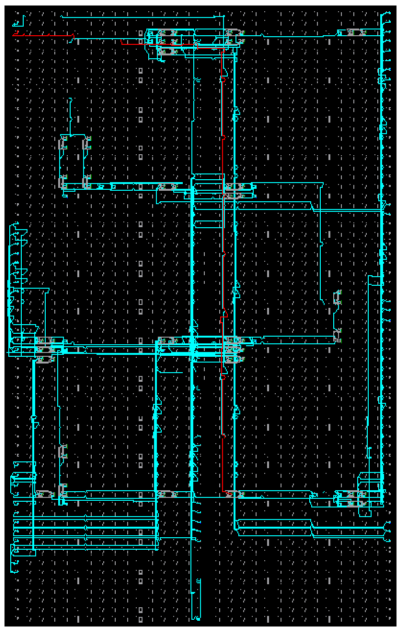
[1]The total external connectivity refers to the sum of the percentages of modules that receive external inputs and produce external outputs. External input or output connectivity ranges from 25% to 100%, thus total external connectivity ranges between 50% and 200%.



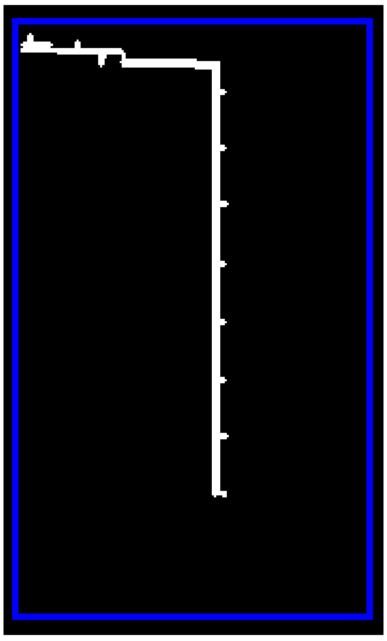
(a) ISE Circuit



(b) ISE Critical Path Highlighted



(c) COMMA Circuit



(d) COMMA Critical Path Highlighted

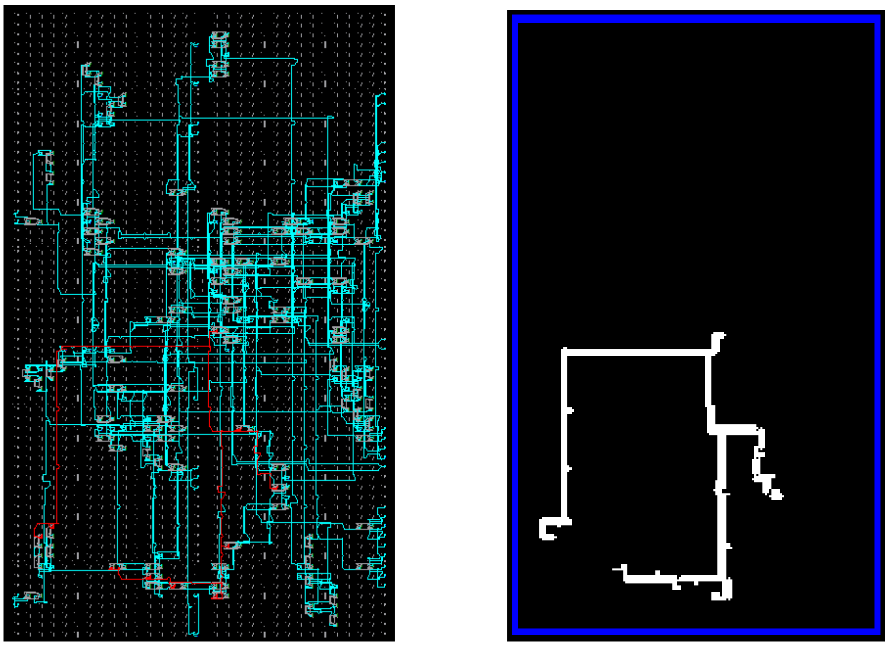
**Figure 5.7:** Circuits and highlighted critical paths where COMMA exhibited high overhead

reduced ISE's opportunities to optimize placement and replicate logic by limiting the availability of resources. ISE's performance relative to COMMA would thus diminish in practice.

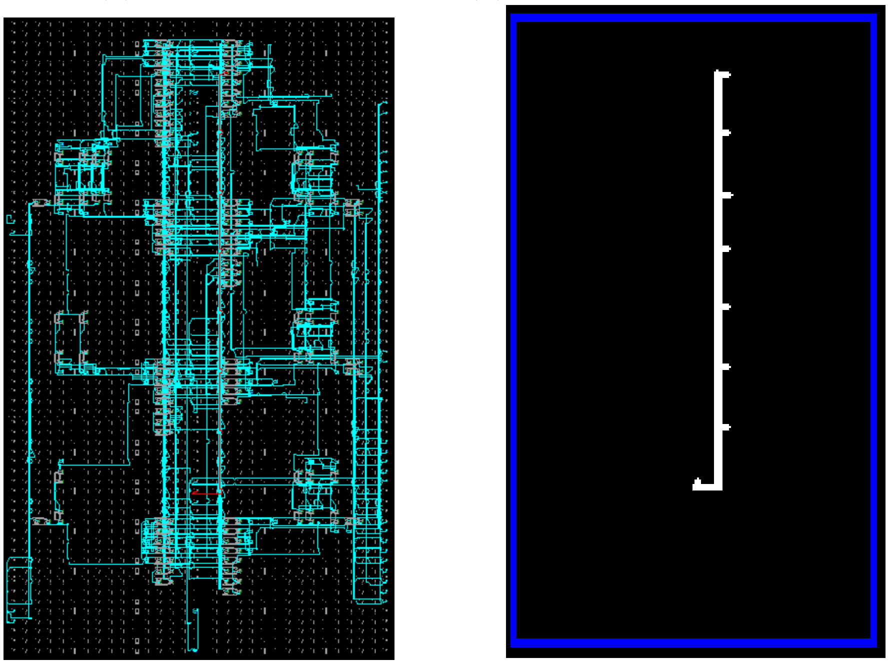
Figure 5.8 depicts an example where the COMMA layout (Figure 5.8(c)) was significantly better (5.243 ns delay for ISE vs 3.339 ns for COMMA). The longest path in the COMMA circuit was again an external I/O connection (this time to a middle I/O bank) and that for the ISE circuit was again a module-to-module connection. In this case, which illustrates a design comprised of 8 modules with 80% internal connectivity and total external connectivity of 125%, the imposition of the COMMA structure allowed longer wires to be used to good effect, whereas ISE needed to employ several shorter wires to connect heuristically placed modules.

In conclusion, the longest path for COMMA, given sufficient channel width, is the sum of the full height and width of the FPGA. This delay may be lower than a local minimum found in ISE, and is usually due to the fixed locations of IOBs or relative module placements. This overhead should become insignificant as routing algorithms struggle to find shorter paths when the number of nets increase and opportunities for logic replication are diminished.

Of primary concern are the implications of the above as the device size scales. With the organization of Figure 5.1, the slot sizes grow in width as larger devices are considered. This means more logic can be packed into a single module, thereby reducing intermodule communication. Larger devices also offer a greater number of configuration "pages" and thus more slots. Larger applications with more modules and a greater number of intermodule



(a) ISE Circuit (b) ISE Critical Path Highlighted



(c) COMMA Circuit (d) COMMA Critical Path Highlighted

**Figure 5.8:** Circuits and highlighted critical paths where COMMA surpassed ISE

wires can thus be accommodated. The results suggest COMMA will perform well under such conditions. There may be longer paths to external IOBs leading to higher overheads, but these may be mitigated by co-locating a module's external connections when the designer is free to do so, and by trying to pack modules with closely located IOBs into the same slot. Alternative slot/channel organizations may also help. The additional resource availability provides more opportunities for ISE to replicate and place logic in good locations, while longer wiring paths in COMMA will lead to greater wire delays, but this effect could be mitigated with suitable buffering of COMMA interconnections.

### 5.3.2 Design Time

The time taken to implement the circuit for each graph was also examined. The execution time to solve the ILP was negligible (below a second) for the relatively small 8-module examples. Instead, the time recorded was to place the slice macros and module logic and to route the interconnections when these were unconstrained, compared to routing the interconnections in the COMMA design, for which the slice macro and module locations were determined by the ILP. Figure 5.9 plots the design time of the PAR tool against the number of wires in each design. It can be seen that ISE takes far longer to complete the design since it has many choices for placing the slice macros which have already been placed by COMMA. As observed from the PAR reports, the bimodal nature of the plot was due to the algorithm entering additional routing phases when it fails some stage. This is a common feature of iterative routing algorithms [61].

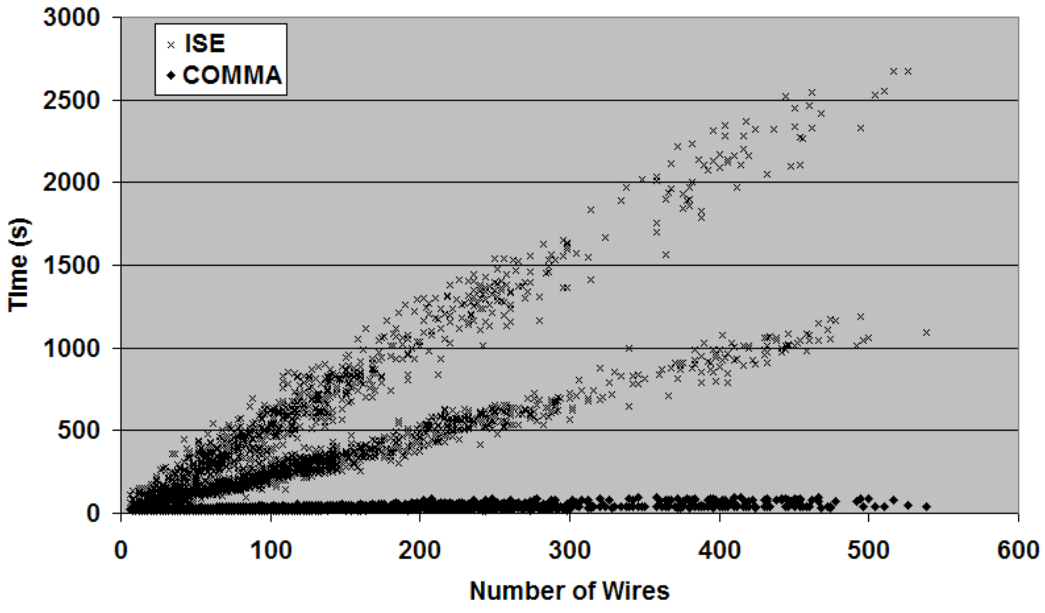


**Figure 5.9:** ISE PAR timing with and without COMMA constraints

## 5.4 Conclusions

This chapter described experiments to analyze the impact on critical path delay of using a regular routing structure as imposed by the COMMA methodology. The experimental method was described, and methods for routing a wiring harness were introduced.

An additional critical path delay was observed that is significant at low utilization and decreases as module and interconnection densities rise. These overheads are expected to diminish when realistic module logic is used. There is also significant PAR speedup due to the constrained placement of module logic and slice macros. Similar benefits should be obtained by using the COMMA principles of focusing on the independent implementation of individual modules rather than on monolithic flattened netlists. Further architectural exploration is required to assess how performance scales to larger devices. This is described in Chapter 8 with the assessment of the method to implement dynamism.

# Chapter 6

# Wiring Harness Generation for Module-based Dynamic Reconfiguration

## 6.1 Introduction

So as to minimize reconfiguration overheads, it is desirable to have a single wiring harness supporting all the communications for a particular application as the wiring never needs to be reconfigured.[1] However, this may be infeasible as implementing such a harness may exceed area and/or timing

[1]The only case where having multiple harnesses might result in lower reconfiguration overheads is when the placement of the modules with multiple harnesses results in less modules being reconfigured, and the savings in reconfiguration overhead exceed that of reconfiguring the wiring harness. However, this is unlikely for two reasons. First, the wire harness takes up a significant amount of area on the device. Second, having a single wiring harness implies that modules of the same type tend to be placed in the same slots in order to maximize reuse of wires. Thus, it is generally more desirable to have a single wiring harness.

constraints. Instead, an application may require several wiring harnesses that are reconfigured at certain stages in the execution.

This chapter presents a method to derive a set of wiring harnesses for a given DR1 application that attempts to minimize the total reconfiguration delay. A technique known as *graph merging* is described, which merges subsequences of communications subgraphs in an application schedule into *periods.* A single wiring harness is implemented for each period. This harness supports all the communications for the subgraphs within it. The optimization goal of subsequence merging is to reduce the total amount of reconfiguration delay. The goal is pursued through judicious module placement and by incrementally building wiring harnesses that satisfy area and timing constraints.

Two subprocesses of graph merging are described. The first is the *merging of two subgraphs*, which merges nodes in two subgraphs to reduce the reconfiguration delay and total communications bandwidth required. The second is *subgraph mapping*, which allocates slots to modules and estimates the critical path and reconfiguration delay of a subgraph.

A case study of an optical flow computation is presented to assess the method. The experimental results show that reconfiguration times are significantly reduced.

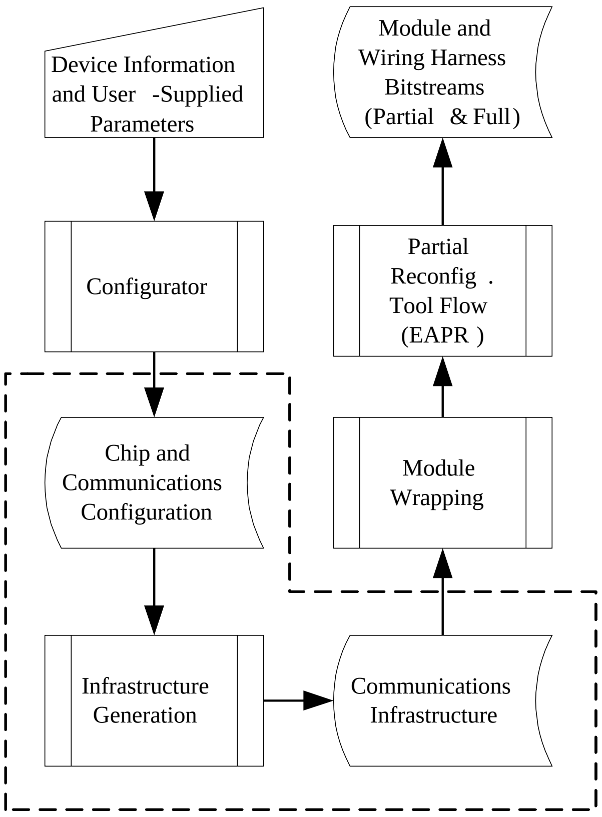
The method and results in this chapter were presented at the 2007 IEEE International Conference of Field Programmable Logic and Applications [52].

## 6.2 Communications Infrastructure Generation Flow

A simplified depiction of the complete tool flow introduced in Section 3.7 is shown in Figure 6.1. The first process in the flow involves obtaining device information (i.e. CLB and IOB grid structure etc.) and user-supplied parameters (e.g. IOB assignments, timing requirements etc.) to create a configuration set containing device- and application-specific parameters. This is followed by the generation of the communications infrastructure for the application, which includes one or more wiring harnesses. Each module is then wrapped in a lightweight or weightless interface to map its ports to specific wires in its wiring harness. The modules and harnesses are then implemented using a toolset such as the Xilinx Early-Access Partial Reconfiguration Toolkit [95].

The "Infrastructure Generation" process consists of several algorithmic steps as depicted in Figure 6.2. Figure 6.2 is a detailed depiction of the dashed area in Figure 6.1. The inputs to the infrastructure generation process are an application specified as a communications graph and the configuration set from the "Configurator" process. The outputs are the low-level details for implementing the wiring harnesses and module wrappers, which are then to be fed into the "Module Wrapping" process. Each of the steps in infrastructure generation will be described in detail in the following sections of this chapter.

The flow goes through a series of stages, each of which is represented as a rectangle (process symbol) in Figure 6.2. The stages are intended to



Device Information and User -Supplied Parameters
Module and Wiring Harness Bitstreams (Partial & Full)
Configurator
Partial Reconfig . Tool Flow (EAPR )
Chip and Communications Configuration
Module Wrapping
Infrastructure Generation
Communications Infrastructure

**Figure 6.1:** COMMA design flow (simplified)

be executed iteratively. If one stage does not produce favorable results, the stage before that should be re-run with different parameters. For example, if the graph merging stage does not produce graphs that meet area and timing constraints, the scheduling stage must be re-run with different optimization targets, e.g., smaller partitions.

## 6.3 Deriving the Communications Graph

The COMMA methodology advocates specifying an application as a communications graph. The communications graph should be derived through

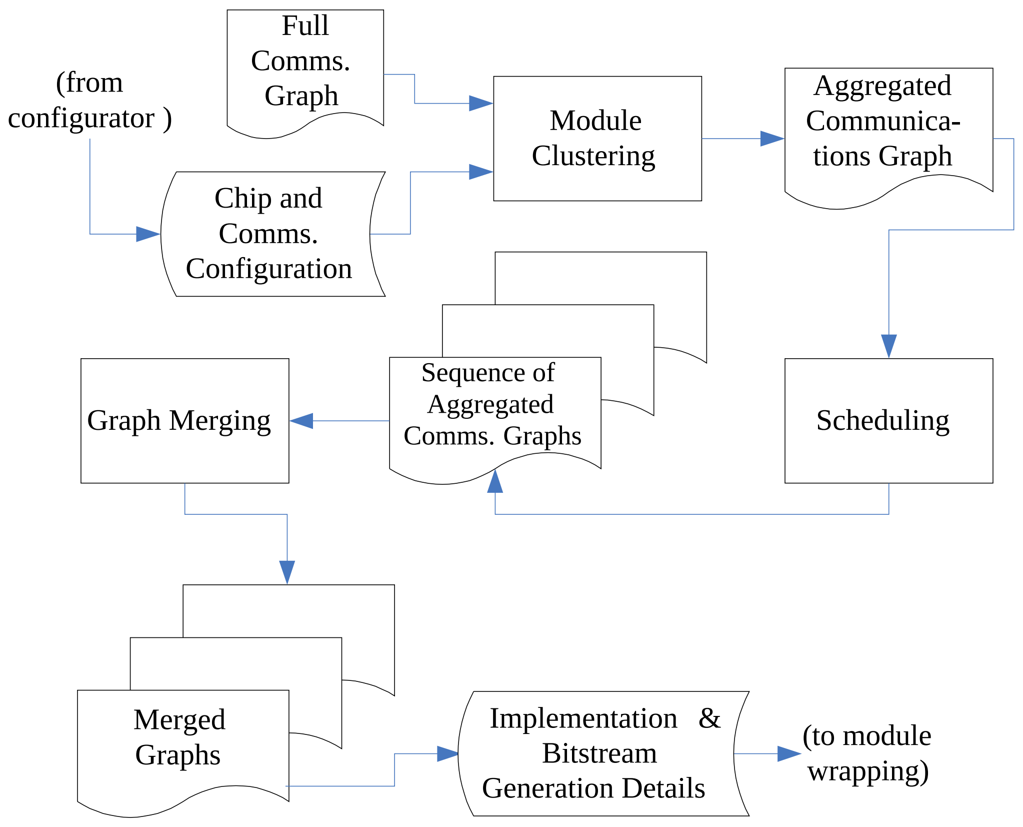

Full Comms. Graph
(from configurator )
Chip and Comms. Configuration
Module Clustering
Aggregated Communica-tions Graph
Sequence of Aggregated Comms. Graphs
Graph Merging
Scheduling
Merged Graphs
Implementation & Bitstream Generation Details
(to module wrapping)

**Figure 6.2:** Communications infrastructure generation flow

the natural partitioning of an application into functional modules. For example, a JPEG application may have DCT and Huffman encoder modules. A communications graph is similar to a task graph where each task node is a module and each edge represents communications between the modules. However, a communications graph also contains physical details about the tasks and inter-task communications. An example is shown in Figure 6.3. Each module has attributes associated with it indicating its approximate size in terms of the target device resources. In Figure 6.3 these are specified by three values “x/y/z” where x is the logic cell count, y is the arithmetic unit or DSP block count and z is the on-chip ram block count that refer to Virtex-4 resources. It is not necessary that these specific types of resource quantifiers are used. Rather, it is important to note that there can be more than one type of resource quantifier. Each edge represents a communications link between two modules and has three attributes: its bitwidth, the output port number of the source module, and the input port number of the destination module. External I/Os can also be represented, depicted as rounded rectangles, with the specific pad numbers or without. The "X" port number is used in Figure 6.3 to indicate a connection to external I/O blocks.

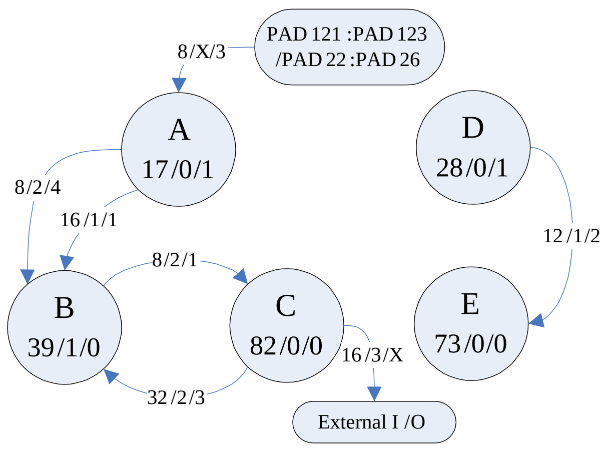


**Figure 6.3:** A sample communications graph

In this work it is assumed that the full bandwidth of each link may be required each clock cycle. This implies that the communications patterns are not specified and thus communication can occur on any link at any time.

## 6.4 Module Clustering

The first step in the infrastructure generation process is to aggregate or split the modules in the full communications graph such that the logic size of each node in the graph fits into the size of the slots chosen by the designer.

The optimization goals in module clustering are as follows:

1. Each slot accommodates as much logic as possible to reduce the number of configurations necessary to implement the entire application.
2. The intermodule communication bandwidth is minimized by clustering modules with wide connecting edges into the same slot and minimizing cut width.
3. The modules aggregated into each slot contain/absorb as much communication as possible.

Clauses 2 and 3 imply that this is a mincut/maxflow problem of partitioning graphs. The additional requirement of clause 1 indicates the partitioning must be constrained by the partition sizes.

If an application is specified in terms of HDL descriptions, it is generally regarded that synthesizing the modules to a flattened netlist rather than maintaining the modular hierarchy and synthesizing each module independently will result in a more efficient implementation in terms of area and delay. Since each aggregated module is flattened, it is assumed that aggregating any two or more modules will result in a resource requirement of less than or equal the sum of their individual resource requirements. That is, if $r(m)$ is the resource requirement of module $m$, then:

$$r(m_1, m_2, \cdots, m_N) \leq \sum_{i=1}^{N} r(m_i)$$

There are many approaches in related domains that can be applied to this problem, including the clustering substep in multiprocessor task assignment [72], and multilevel partitioning algorithms such as METIS [45], which balances the partition sizes according to the combined logic size in each partition. METIS also performs multi-constraint partitioning, i.e., each vertex can have multiple weights. This is especially useful as the specification of application graphs in COMMA (see Section 6.3) allows multiple resource quantifiers in each module. It is proposed that such approaches be used to perform this step.

## 6.5 Scheduling

The output of the module fitting and clustering steps is a communications graph with a similar format to the original communications graph. The main difference with the graph's original graph specification (as described in Section 6.3) is that the resources required for each node are less than or equal to the available resources in each slot because the module clustering step aggregates or splits the modules into the given slot sizes. Each node can thus contain a collection of aggregated and/or split modules. Without loss of generality, it is assumed that this graph may be too large to fit onto the target device. The graph is therefore partitioned into a schedule of *subgraphs*, each of which must contain no more nodes than the total number of slots available on the device.

Henceforth the term *subgraph* will be used to indicate a partition of a full communications graph containing no more nodes than there are slots available in the target device.

Figure 6.4 depicts a schedule of the communications graph from Figure 6.3. In this example, each box represents one of the temporal partitions of the full communications graph the application has been divided into. The sequence of partitions corresponds to the sequence of configurations that are to be loaded onto the device. Each partition comprises a smaller graph that captures the communications between the modules needed while the corresponding configuration is active.

Note that module B appears in the first two subgraphs. The appearance of the same module name in different subgraphs indicates that the logic required to implement that module is the same in each subgraph. However, in this case, module B seems to present a different interface in the two subgraphs. This is because a module only needs to present the *subset* of its interface required in each subgraph. In Figure 6.3, module B is connected to both modules A and C. After scheduling, only module A is present in Subgraph 1, thus only the connections between modules A and B are shown, and in Subgraph 2 only module C is present thus only the connections between modules C and B are shown. The RDP wrapper for module B will direct the ports to their appropriate slice macros. This is the result of the partitioning performed in this scheduling stage and can represent the following behavior:

1. Module B goes through two distinct stages of execution in Subgraph 1 and Subgraph 2. For example, Module B may be accepting some data from Module A in Subgraph 1 and processing it. In Subgraph

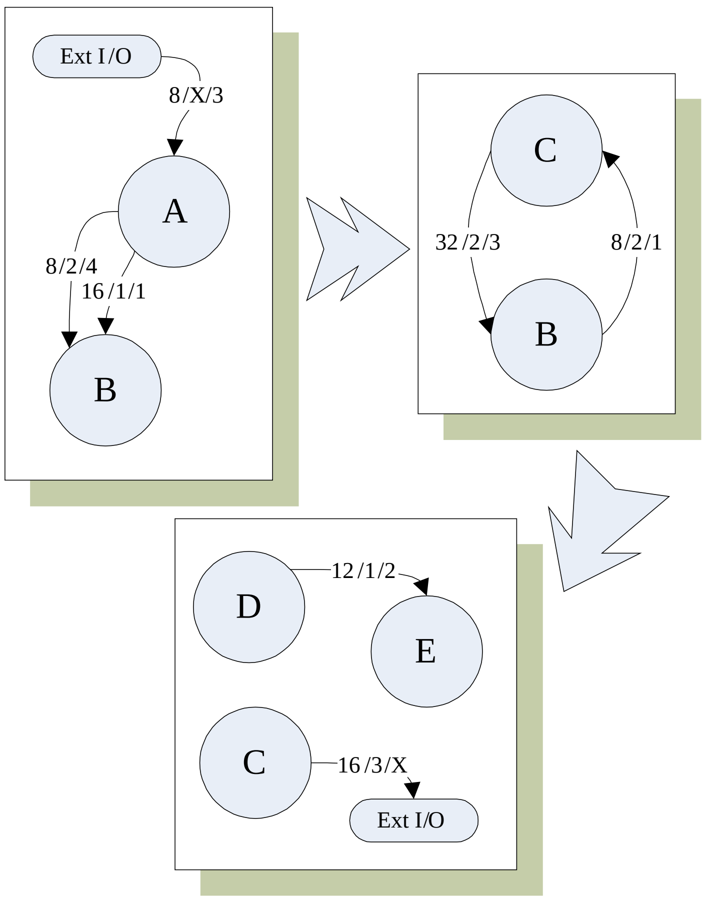


**Figure 6.4:** An example of a scheduled graph

2, Module C consumes the processed data from Module B and also provides some feedback, e.g., a checksum value.

2. Since graph scheduling is performed on a *clustered* graph, Module B may be an aggregation of two different modules, say *B1* and *B2* that were presented in the original full communications graph. In Figure 6.4, B1 is being executed in Subgraph 1 and completes its run. Thereafter the device is reconfigured to Subgraph 2 when B2 is executed.

Other application behavior is also possible. It is important to note that in any case the logic to implement Module B remains the same whichever subgraph Module B is found in.

In this thesis the communications graph shows data dependency assumed by Purna et al., as their algorithm is used in this thesis for scheduling. This enables the scheduler to partition the graph into temporally-correct partitions. In the example in Figure 6.3, it is assumed that Module A completes its task, producing data for Module B to consume. Module A is no longer required after Module B consumes its data. In a different scenario, if Module A is active at some point *after* Module B has completed execution, then another instance of Module A will be present in the full communications graph.

### 6.5.1 Loops

It was noted in Sections 2.2.1 and 4.7 that loops may cause undesirable effects such as configuration thrashing. Thrashing refers to the successive loading of two or more configurations repeatedly and may arise when each configuration contains part of a single loop. Thus, this approach is best suited to DAGs, or cyclic graphs in which the cycles do not span partitions.

This step is implemented using traditional partitioning and scheduling algorithms such as those identified in Section 2.2.2 and 4.7. To prevent thrashing, the chosen algorithm should contain any cycles in the application's communication graph within individual partitions or minimize the number of partitions that contain a single loop.

## 6.6 Graph Merging

As explained in the previous section, the output of the scheduling step is a sequence of subgraphs. Viewed at the top level, without the individual modules in each subgraph, the resulting schedule can be depicted as a linear dependency graph as shown in Figure 6.5. Each node in the schedule is a subgraph, which has a constraint $d(G_i)$ specifying the target maximum critical path delay associated with it (see Section 4.5). The edges between the subgraph indicate temporal dependency, i.e., subgraph $G_i$ is reconfigured to $G_{i+1}$. The weight $r(G_{i,i+1})$ indicates the maximum target delay to reconfigure the modules and, if necessary, the wiring harness, between $G_i$ and $G_{i+1}$. Both the critical path and reconfiguration delay constraints are optional.

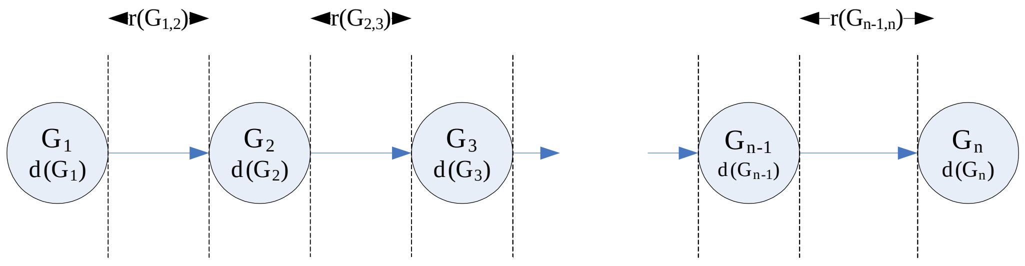


**Figure 6.5:** Another view of a scheduled graph

Graph merging aims to merge contiguous subsequences from the scheduled graph such that for each merged subsequence a fixed harness can be built that supports the communications for all subgraphs in the subsequence. This attempts to reuse previously formed connections and to make use of spare wiring capacity to reduce the overall cost of reconfiguring the wiring at application run time. The reconfiguration delay of a sequence of merged graphs can then be split into two parts: the time to reconfigure individual modules, and the time to reconfigure the wiring harness when it is necessary to do so.

The goals of graph merging are twofold. First, the total reconfiguration delay of the application sequence should be minimized by selecting appropriate contiguous subsequences to merge. Second, the module-slot allocations should be judiciously determined such that the need to reconfigure modules between configurations is minimized. Alternatively, this second objective can be expressed as the goal to maximize the number of modules of the same type placed in the same slot in adjacent configurations.

The critical path delay of each resulting wiring harness must not exceed the minimum $d(G_i)$ for the graphs of the corresponding subsequence.

### 6.6.1 Overview of Graph Merging

An overview of graph merging is depicted in Figure 6.6. The main process is known as "subsequence merging", which is comprised of two subprocesses: "merge two subgraphs" and "map subgraph".

The "subsequence merging" process takes the entire sequence of subgraphs as its input. This sequence is produced from the scheduling stage of the design flow. It then groups subgraphs in the application sequence into *periods*, each of which has a single wiring harness that satisfies the communications requirements of all the subgraphs in the period.

This is done by iteratively choosing two (merged) subgraphs to merge together into a single (merged) subgraph, and then determining if the wiring

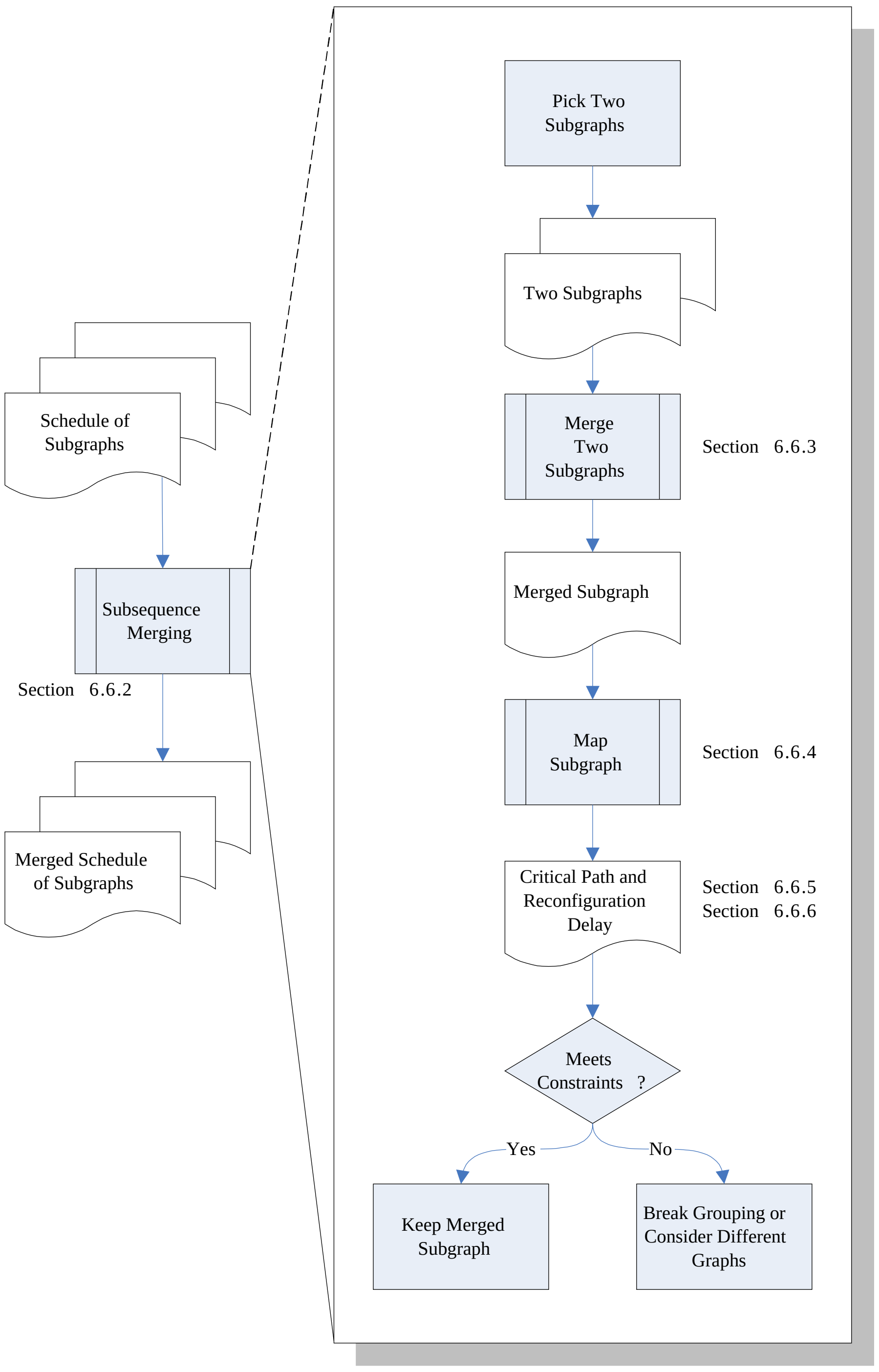


**Figure 6.6:** Overview of graph merging

harness for this subgraph meets timing and area constraints were it mapped to the device.

The "merge two subgraphs" subprocess merges two subgraphs into a single subgraph. In doing so, it attempts to reduce the reconfiguration overhead by allocating modules that are common to both subgraphs to the same slot, thereby removing the necessity to reconfigure the slot between configurations. Ultimately, each merged subgraph that remains in the application sequence after the entire merging process is finished defines a *period*.

The "map subgraph" subprocess maps the wiring harness of the subgraph to the device and determines if it meets the application's area and timing constraints. Global routing paths are determined for each communication arc in the subgraph, and this subprocess reports if there is sufficient area for all the communications in the subgraph to be mapped to the device. In addition, it also reports the estimated critical path delay that can be used to compare against timing constraints, if necessary.

Each of these processes is described in detail in the following subsections. The cost model used to determine the reconfiguration delay between configurations is also presented after the description of the processes.

### 6.6.2 Subsequence Merging

Merging subgraphs into subsequences forms periods that can be implemented using a single wiring harness on the device. The following greedy algorithm was examined for determining which subsequences should be created:

1. Try to merge the first two graphs in the application sequence using the algorithm to merge two graphs (see Section 6.6.3).
2. Map the merged graph using the proposed mapping algorithm (see Section 6.6.4) and examine the area use and critical path delay:
    (a) If the area and timing constraints of the wiring harness for the merged graph are satisfied, then remove the first two graphs from the application sequence and replace them with the merged graph. Return to step 1 and try to merge the next graph in the schedule with the merged graph at the start of the sequence.
    (b) Otherwise, the constraints are not satisfied and the merge is unsuccessful. The first graph in the application sequence forms a subsequence on its own. Remove it from the application sequence and add it to the list of merged subsequences (periods).
3. Return to step 1 and repeat until the application sequence has been processed in its entirety i.e. all subsequences have been formed.

This algorithm tries to greedily merge the first subgraph with the second. If area and timing constraints are met, this merged subsequence replaces both the first and second graphs. Once the constraints are exceeded, the first subgraph (merged subsequence) is removed and added to the list of periods. A more detailed listing of the algorithm appears in Algorithm 1.

**Algorithm 1** MergeSubsequences

```
Input: All subgraphs $G_1 \cdots G_n$
1: $periods \leftarrow \emptyset$ {the sequence of generated periods}
2: $current \leftarrow \emptyset$ {the current period being processed}
3: for $i = 1$ to $n$ do
4:    if $current = \emptyset$ then
5:       $current \leftarrow G_i$
6:    else
7:       $testmerge \leftarrow MergeTwo(current, G_i)$
8:       if $Map(testmerge)$ exceeds area or timing constraints then
9:          Add $current$ to the end of $periods$
10:         $current \leftarrow G_i$
11:      else
12:         $current \leftarrow testmerge$
13:      end if
14:   end if
15: end for
16: Add $current$ to the end of $periods$
Output: The sequence of merged subgraphs: $periods$
```

### Time Complexity

The time complexity of this algorithm and its two constituent algorithms (*MergeTwo* and *Map*) can be measured in relation to two different inputs — the size of the target application, and the size of the device.

Let $n$ denote the number of subgraphs the application was partitioned into. This is the total number of distinct configurations required to implement the application on a device. Keeping the device size constant, $n$ is representative of the application size. This main algorithm, *MergeSubsequences*, consists of a single loop that runs for $n$ iterations, thus the time complexity to merge subsequences with respect to $n$ is $O(n)$.

*MergeTwo* (line 7) has a time complexity of $O(z^3)$ where $z$ is the number of slots on the device (see Section 6.6.3), and *Map* (line 8) has a time complexity of $O(z^3 \log z)$ (see Section 6.6.4).

Therefore, the overall time complexity is $O(n \cdot z^3 \log z)$ where $n$ is the number of subgraphs and $z$ is the number of slots on the device.

### 6.6.3 Merging Two Subgraphs

#### Problem Definition

Graph merging determines which communications subgraphs to merge into subsequences which use a single wiring harness. However, before this can be done it is necessary to describe the approach taken to merge adjacent subgraphs in the schedule. The problem of merging a subgraph $G_1$ with the subgraph $G_2$ following it in the schedule is defined as follows:

> Define graph $S$ to be equivalent to $G_1$ with additional, unconnected "blank" nodes representing empty slots that $G_1$ does not make use of. Overlay the nodes of $G_2$ onto $S$ such that the total number of shared arc-bits is maximized and the total number of required module swaps is minimized. An arc can be shared if there exists an arc $a_{u,v}$ between two nodes $(u, v)$ in S, and there exists an arc $a_{w,x}$ between two nodes $(w, x)$ in $G_2$, and if $w$ replaces $u$, and $v$ replaces $x$.

Maximizing the number of shared arc-bits, or minimizing the number of arc-bits that need to be added to $S$, is equivalent to the problem of finding the smallest supergraph of the two graphs $G_1$ and $G_2$. No obvious polynomial time algorithm to solve this problem has been found. Similarly, no reduction to a known NP-complete problem has been found. Thus the complexity of the problem remains open.

**Approach**

As this problem is very complex, the following heuristic algorithm is proposed to merge two graphs:

1. Sort nodes in $G_2$ in order of the total number of bits of communication required.
2. If there are nodes in $G_2$ that have the same type as nodes in $S$, place them into the same slot. Modules that have the same module type do not require reconfiguration. Module "type" is analogous to the VHDL entity or Verilog module type.
3. For the rest of the nodes in $G_2$, place each node into a slot (in $S$) according to a cost function that accounts for the total number of communication bits that will be shared due to placing the node, the total number of bits that may be shared due to communications between unplaced nodes, and the reconfiguration time.

This algorithm first tries to merge nodes with the same module type together. Ultimately, each node is assigned a slot in the subgraph mapping process. Having the same module type in subsequent configurations in the same slot eliminates the need to reconfigure the slot. This first step tries to minimize reconfiguration delay, and the motivation for doing this is twofold.

First, slots take up a large proportion of the logic area on the FPGA, thus reconfiguring slots generally has a higher cost than reconfiguring wiring. Second, if a module were to remain in the same slot between two configurations it is likely to expose the same interface to the wiring harness. It is in turn likely that the communications requirements for a configuration at time $t+1$ is more similar to that at time $t$ than if the module were to be placed in a different slot. Having similar communications requirements across configurations allows more configurations to share a single wiring harness, thus also reducing the number of wiring harnesses.

Next, the algorithm tries to merge nodes that maximize the amount of communication bits that can be shared. This attempts to minimize the total communications density by reducing the number of wires to be added, thus allowing longer subsequences to be created.

A more detailed description of the proposed heuristic algorithm, with an illustrative diagram in Figure 6.7, is as follows:

1. Create a graph $S$ and initially let $S = G_1$. With each arc label:
    (a) Remove the port information, e.g., "8/1/2" is replaced by "8".
    (b) Add a "sharing factor" of 0 to the label, e.g., "8" becomes "8/0".
    (c) Consolidate arcs with the same source and destination.
    (d) For example, in Figure 6.7, in the graph $S$ after step 1 (after the arrow labeled "1"), the two arcs from $A$ to $B$ have their port information removed, and are consolidated to one arc labeled "24/0".

2. Add unconnected vertices to $S$ until the total number of vertices corresponds to the number of slots on the device, not including external I/O vertices. E.g., in Figure 6.7 the number of slots is 4.

3. Sort all the vertices in $G_2$ in descending order of the number of bits of communication, giving the sequence $M_2$.

4. Place each vertex in $M_2$ with the same module type in the same location in $S$. E.g., in Figure 6.7 this is Step 2 (after the arrows labeled "2"), where the module $C$ is placed in the same location.

5. As for the rest of the modules, they can be placed in any of the available vertices. Reuse any arcs that are present, e.g, in Figure 6.7 there are three possibilities for placing the module $E$:

   (a) 3a would require 1 module configuration and 92 wires (20 additional wires).

   (b) 3b would require 1 module reconfiguration and 84 wires (12 additional wires).

   (c) 3c would require 1 module reconfiguration and 80 wires (8 additional wires).

6. 3a uses the most wires but the module can be pre-placed while $G_1$ is still running. On the other hand, 3c uses the least number of wires.

7. Evaluating every possibility will be of exponential (factorial) complexity. Thus the following heuristic is used to evaluate the possibility of each module:

(a) For each remaining module in $M_2$, go through each vertex in $S$, calculating three cost components: the number of confirmed shared bits $\beta$, the number of unconfirmed shared bits $\mu$ and the reconfiguration cost $\rho$.

   i. The number of confirmed shared bits $\beta$ is the sum of the arc weights that can be shared to and from the module with respect to modules that have already been placed. E.g., in Figure 6.7, placing $E$ as in 3a would share 0 bits, 3b would share 8 bits and 3c would share 12 bits.

   ii. The number of unconfirmed shared bits $\mu$ is the sum of the arc weights to and from the module with respect to modules that have not yet been placed. E.g., in Figure 6.7 there are 18 bits from module $E$ to $H$ in graph $G_2$. In 3a and 3c, no matter where $H$ would be placed in the future, there is no possibility of sharing. In 3b, $H$ can be placed optimally at $B$ in the future, thus $\mu = 18$ if 3b were chosen. The location where each module might be placed in the future is heuristically determined by placing them in order of the largest number of reused bits of communication.

   iii. If placing the module at that location requires a module to be swapped, as in 3b and 3c, versus placing it in a spare slot (as in 3a), let the module reconfiguration cost $\rho$ be 1, otherwise, 0.

(b) Calculate the cost of placing the module $n$ at vertex $m$ with the following formula: $cost(n\_at\_m) = totalbits(m) - \beta - \nu \cdot \mu - \phi \cdot (1 - \rho)$.

$\nu$ is the factor by which an unconfirmed shared wire contributes to alleviating the total cost. If $\nu$ is 1, an unconfirmed shared wire is as a good as a confirmed shared wire. $\phi$ is the equivalent number of wires that have to be reconfigured if a module were to be reconfigured, and sets the trade-off point between placing a module in a free slot or into one where more wires can be shared. $\nu$ and $\phi$ are heuristic optimization variables that are used to fine-tune the algorithm. This thesis uses the following values in the experiments described in Chapter 8:

i. The value of $\nu$ should be set according to how well the algorithm for calculating the unconfirmed shared bits predicts the future sharing. If the value of $\nu$ used is 1, the algorithm used should predict the unconfirmed shared bits perfectly. On the other hand, a value of 0 indicates that unconfirmed shared bits should be left out of the equation entirely. The experiments in Chapter 8 used a value of 0.5.

ii. The value of $\phi$ can be determined as follows: The number of wires per CLB column in the device multiplied by the number of CLB columns in a slot indicates the number of vertical wires that have to be reconfigured if a slot were to be reconfigured. This is further multiplied by what would be considered as a reasonable percentage of utilization of all the wires in a CLB column (see the description of $\sigma$ in Section 6.6.4 under the "Wire Delay Cost Model"). As Virtex-4 devices are reconfigured in vertical frames it may cost more to reconfigure horizontal wires, thus the experiments in this thesis multiplies this value by 0.5 to indicate that it is likely that horizontal wires may be used as well, arriving at the final value of $\phi$. There are approximately 22 wires per CLB column, and the value of $\sigma$ used in this thesis is 0.8, thus the experiments conducted for this thesis uses a value of $8.8\times$ *slotwidth* (rounded up to the nearest whole number) for $\phi$.

(c) Choose the placement with the lowest cost and proceed to the next module.

A listing of the algorithm to merge two subgraphs appears as Algorithm 2.

**Time Complexity**

The main variable in this algorithm that determines the time complexity is the number of slots allocated on the device. Let $z$ denote the number of slots in the device. The run time of this algorithm is the sum of the following three cost components:

1. At line 8, the nodes in $G_2$ are sorted. This can be performed in $O(z \log z)$ time with an algorithm such as in-place merge sort [46].
2. The two main loops, commencing at lines 9 and 17 respectively, contain two nested loops at lines 10 and 21 respectively. Both loops iterate $O(z^2)$ times.
3. The algorithm used to calculate the number of unconfirmed shared bits at line 24 is a straightforward greedy algorithm that consists of single loop running for $O(z)$ iterations.

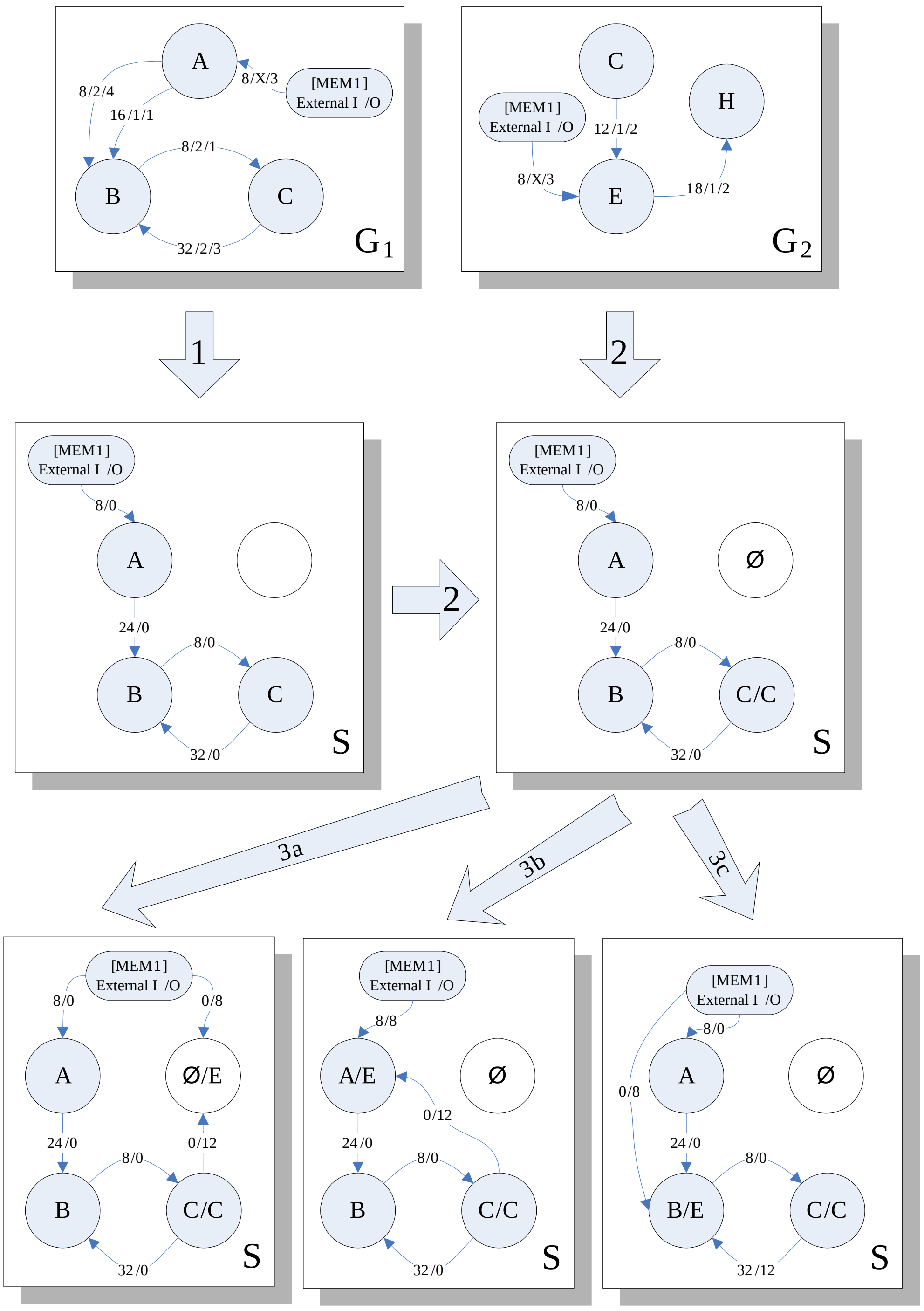


**Figure 6.7:** Example for graph merging

**Algorithm 2** MergeTwoSubgraphs

```
Input: Subgraphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$
 1: $[S = (V_S, A_S)] \leftarrow G_1$
 2: $RemoveAllPortInformation(A_S)$
 3: $SetAllSharingFactors(A_S, 0)$
 4: $ConsolidateArcs(A_S)$
 5: while $|S| < NumSlots$ do
 6:     AddEmptyNode(S)
 7: end while
 8: $M_2 \leftarrow SortDescendingBits(V_2)$ {sort the vertices in $G_2$ in descending
    order of the number of communication bits}
 9: for all $m$ in $M_2$ do
10:     for all $v$ in $V_S$ do
11:         if $m.type$ is equal to $v.type$ and $v$ is not merged yet then
12:             $Merge(v, m)$
13:             $M_2 \leftarrow M_2 - m$ {remove $m$ but retain ordering}
14:         end if
15:     end for
16: end for
17: for all $m$ in $M_2$ do
18:     $lowestcost \leftarrow \infty$
19:     $bestnode \leftarrow \emptyset$
20:     $\rho \leftarrow 1$
21:     for all $v$ in $V_S$ do
22:         if $v$ is not merged yet then
23:             $\beta \leftarrow CalculateConfirmedSharedBits(v, m)$
24:             $\mu \leftarrow CalculateUnconfirmedSharedBits(v, m)$
25:             if $v$ is empty then
26:                 $\rho \leftarrow 0$
27:             end if
28:             $cost \leftarrow totalbits(v) - \beta - \nu \times \mu - \phi \times \rho$
29:             if $cost < lowestcost$ then
30:                 $lowestcost \leftarrow cost$
31:                 $bestnode \leftarrow v$
32:             end if
33:         end if
34:     end for
35:     $Merge(bestnode, m)$
36: end for
Output: The subgraph $S$
```

As the second main loop contains *CalculateUnconfirmedSharedBits*, the time complexity to merge two subgraphs is $O(z^3)$. $z$ is constant with respect to $n$ (the number of subgraphs in the application) and only grows as larger devices are used.

### 6.6.4 Mapping a Subgraph onto a Device

The two metrics used in this thesis to determine the effectiveness of subgraph merging are the contribution to the critical path delay by the wiring harness, and the reconfiguration delay between subgraphs. The critical path delay refers to the maximum pin-to-pin delay of a wiring harness of a merged graph.

This subsection describes a process known as subgraph *mapping*, which is defined as the assignment of slots to each module in a subgraph, followed by the determination of estimated, global routing paths for each arc in the subgraph. The outputs of subgraph mapping are:

1. A set of slot-module allocations;
2. A set of global routing paths that is used for critical path delay and reconfiguration delay estimation (and that can serve as input into a detailed router if required); and
3. An estimated critical path delay of the wiring harness.

Each module in a subgraph is first allocated a slot on the device, with the optimization goals being to first minimize the number of wires across any cut, and then to minimize the total wire length. Minimizing the cut width has two advantages. Firstly, the channel width is minimized, thus maximizing the amount of module logic area. Secondly, if the channel width is fixed, minimizing the cut width decreases the likelihood of the channels being congested. In an FPGA with heterogeneous routing resources such as the Virtex-4, the amount of channel congestion directly impacts the critical path delay. This is because shorter wires may have to be used as the channels become increasingly congested, thus resulting in more switchbox hops.

The second step in graph mapping is to "map" each wire to the device model by estimating the routing path that is expected to be taken by the low-level routing algorithm. This subsection details the device model and interconnect prediction algorithm used to estimate the critical path delay.

Interconnect prediction for ASIC design flows has been studied extensively, and Kannan et al. [44] have compared some routability estimation methods as they are applied to FPGAs [3][4][68][55][59][74]. Although these methods estimate the detailed routing between logic blocks in an island-style FPGA, they are generic enough to be adapted to the slot-to-slot routing estimation in COMMA.

This step is extremely important as it provides two crucial pieces of feedback to the subsequence grouping process. Firstly, it reports the estimated critical path delay to ensure that it does not exceed any predefined timing constraint. Secondly, it determines whether the subgraph can be mapped onto the device or not, i.e., if it meets area constraints. If either of these two constraints are not met, the subgraph cannot be implemented and an alternative merging must be attempted.

The feasibility of implementing a subgraph on an FPGA can of course instead be determined by actually performing the low-level place-and-route processes. However, doing so for a single subgraph can take up a significant amount of time. As shown in Section 5.3, a single place-and-route process using Xilinx ISE's PAR tool can take up to 45 minutes for the smallest XC4VLX15 device. Doing this with each potential merged subgraph as part of an iterative algorithm such as subsequence grouping can take up an inordinate amount of time, thus an estimation method should be investigated.

The interconnect estimation method described here is based on Lou's method [55] of dividing the device model into a grid and the delay-lookup approach of Manohararajah et al. [59].

### Device Model

This device model is based on the routing congestion estimation method by Lou et al. [55]. Lou's method involves dividing the chip area into rectangular grids, where each grid has four properties:

- The routing *capacity* of the grid, i.e., the number of routing tracks available in the grid.
- The *usage* of each grid, i.e., the number of used tracks in the grid.
- The *horizontal congestion cost* and *vertical congestion cost* in each grid. This is the ratio of the usage to the capacity of the grid.

Lou's method also makes the following assumptions:

1. A net is optimally routed with the shortest length.

2. Each net makes at most one direction change in a grid.

3. If a change of direction occurs, it occurs where a straight line can be drawn to the target pin. If a net bends within a grid, half a horizontal track and vertical track is *used*.

In the COMMA approach the device is divided into *cells* (each of which is analogous to a *grid* in Lou's method), also containing a fixed amount of wiring resources. In contrast to Lou's method, each cell records the number of wires *exiting* the cell rather than *within* the cell. This approach was chosen to simplify the calculations when nets bend within a cell.

The timing model is based on an approach by Manohararajah et al. [59], which uses a single delay value for each connection type in a table. This approach predicts "some aspects of interconnect timing", and they have noted that its accuracy is very high when the results are compared with running the placement tool itself. In the COMMA approach, each cell has a fixed optimal delay value that is obtained through experimentation with expected wire delay values for a Virtex-4 FPGA.

An example of a very simple graph consisting of two nodes is mapped onto the device in Figure 6.8. There are two modules placed onto the device with a single wire connecting M1 to M2.

A device is split into two types of cells: *slot cells* (depicted in white or black, where white slots are empty and black slots are filled with a module), and *channel cells* (depicted in light grey).

Each slot cell has the following attributes:

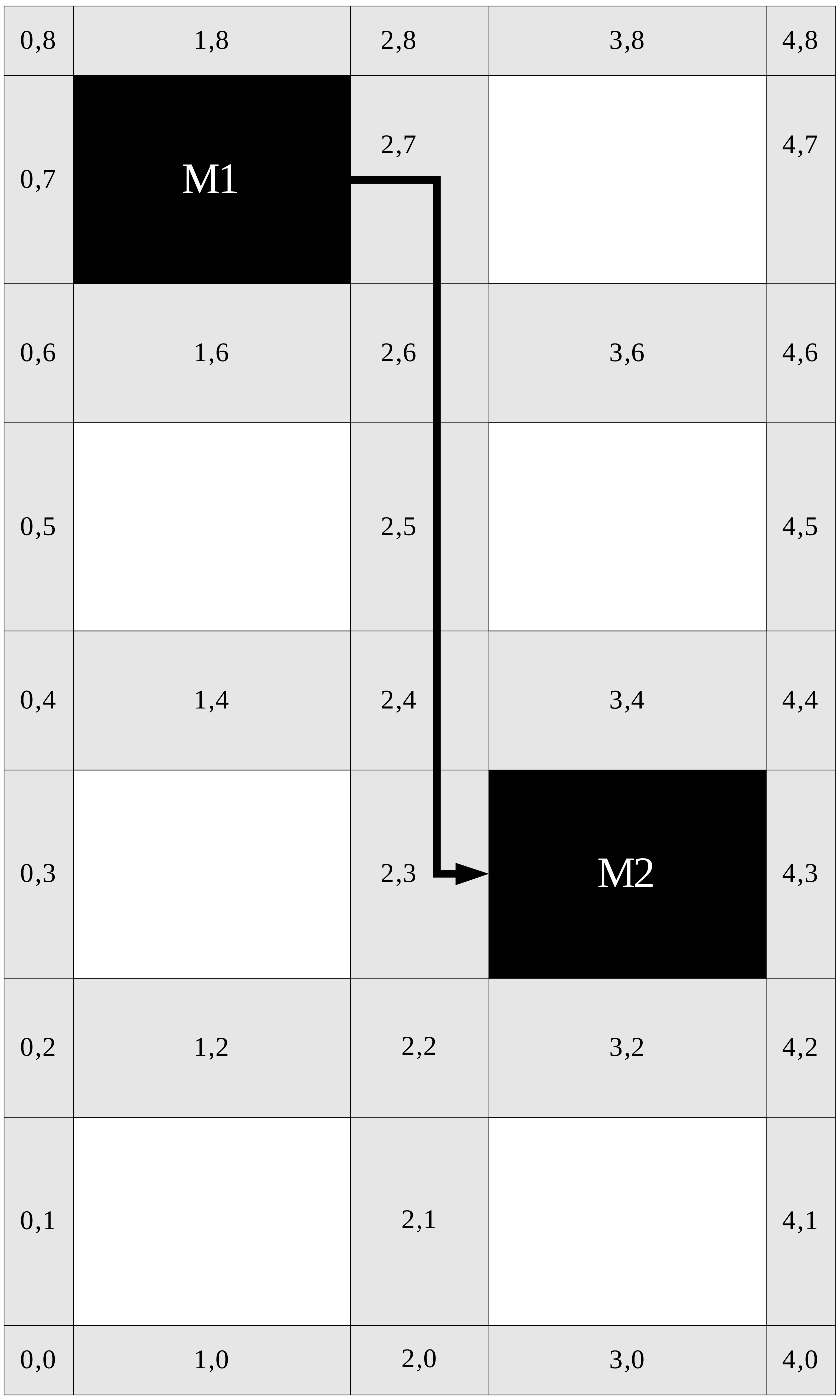


**Figure 6.8:** Device mapping layout

- $x$ and $y$, which are the horizontal and vertical coordinates of the slot, starting from (0, 0) on the bottom left to ($columns - 1$, $rows - 1$) on the top right of the device.
- $r$, which is the time taken to reconfigure the slot to a different module; this attribute may be set to a constant for all slots.
- *ncap*, *ecap*, *scap*, and *wcap* which are the maximum number of wires that can enter or exit the slot at each slot boundary. This is bounded by the number of slice macros that can be placed around the slot.
- *nutil*, *eutil*, *sutil* and *wutil* which are the northbound, eastbound, southbound and westbound utilizations of each slot boundary. They are incremented as wires are used and are bounded by *ncap*, *ecap*, *scap* and *wcap*.

Each channel cell has the following attributes:

- $x$ and $y$ which are the x and y locations of the channel, starting from (0, 0) on the bottom left to ($2 \cdot columns$, $2 \cdot rows$). Note that slot cells coordinates are skipped, and slot coordinates can be translated to channel coordinates by multiplying by 2 and adding 1.
- *hcost* which is the cost to cross this channel horizontally; this is a heuristic estimate of the delay to get from one side of the channel cell to the other side. This estimate is used because it is impossible to accurately predict the detailed routing at this stage.

- *vcost* which is the cost to to cross this channel vertically; again this is a heuristic estimate of the delay from the top/bottom to the bottom/top of the channel cell.
- *ncap*, *ecap*, *scap* and *wcap* which are the northbound, eastbound, southbound and westbound capacities of the channel boundary. E.g., having a capacity of 10 for the northbound boundary of channel (2,7) indicates that at most 10 wires can cross the north boundary of the channel.
- *nutil*, *eutil*, *sutil* and *wutil* which are the northbound, eastbound, southbound and westbound utilizations of each channel boundary. They are incremented as wires are used and are bounded by *ncap*, *ecap*, *scap* and *wcap*.

The reconfiguration cost for channel cells involves an estimation algorithm and is detailed in Section 6.6.6.

The example shown in Figure 6.8 has the following slot mappings for the modules:

- **Placement:** M1:(0,3), M2:(1,1).
- **Utilizations of the following slot boundary variables:** $slot_{0,3}.eutil = 1$, $slot_{1,1}.wutil = 1$.

The following channel mappings are used for the arc between M1 and M2:

- **Cost of arc between M1 and M2:** $\frac{1}{2}(channel_{2,7}.hcost) + \frac{1}{2}(channel_{2,7}.vcost) + channel_{2,6}.vcost + channel_{2,5}.vcost + channel_{2,4}.vcost + \frac{1}{2}(channel_{2,3}.vcost) + \frac{1}{2}(channel_{2,3}.hcost)$

- **Utilizations of the following channel boundary utilization variables:** $channel_{2,7}.sutil = 1$, $channel_{2,6}.sutil = 1$, $channel_{2,5}.sutil = 1$, $channel_{2,4}.sutil = 1$

Instead of keeping track of every single wire in each channel cell, the utilization of wires exiting slot and channel **boundaries** are recorded. Not having to keep track of every wire simplifies the model as the detailed routing is not recorded. This is adequate for routing estimation purposes as knowing which boundaries a wire enters from and exits to is sufficient to know if it bends in the channel cell.

### Problem Statement

The problem of device mapping can be stated in two stages. The first stage is to place the modules in the slots:

> Given a subgraph $G = (V, A)$, place each module $v \in V$ in a slot such that the estimated channel width utilization and longest wire length are minimized.

After a placement has been obtained for each module, each arc must then be mapped into the available channels:

> Given a subgraph $G = (V, A)$ and for each $v \in V$ a placement $p(v) = (x, y)$, determine a route through the channels for each arc $a \in A$ such that the cost of implementing these routes is minimal, and that the slot and channel boundaries do not exceed their capacities.

### Approach

The first sub-problem, slot allocation, can be solved with methods such as the integer linear program described in Chapter 5, or with floorplacement algorithms such as Capo [71]. Both these methods aim to minimize the cut-width and wire lengths. The ILP method produces a set of optimal allocations for any graph, but does not scale well as device sizes grow. This is because the number of possible solutions grows factorially with the number of slots in the device. On the other hand, a floorplacer may not be optimal but is fast, and has been shown to produce very good results in standard cell placement experiments performed by the scientific community. If a standard cell floorplacer such as Capo [71] is used, one approach to use is to model the device with a standard cell layout as depicted in Figure 6.8. Channel cells can be marked with a “do-not-place” constraint or flagged as obstructions.

As for the second problem, even at this global routing level, routing every wire arc into optimal path assignments is very time consuming. It is thus proposed that heuristics be used to solve this problem. The heuristic used here is to route the connections in descending order of the distance between the modules they connect.

Manohararajah et al. [59] have noted that place-and-route tools always use faster, longer wires for more critical connections and slower, shorter wires for less critical connections. The *criticality* of a connection refers to how close the connection is to being *critical*. While the *critical* path is the longest delay path in the circuit, there is a strong correlation with the longest physical path. Longer connections have a greater probability of becoming critical if faster wires have already been used for shorter connections. Thus, sorting the connections in descending order mimics what a detailed router does at a higher level by allowing longer connections to use faster routing paths through the channel cells. The steps to perform routing are as follows:

1. Sort all arcs in descending order of length.
2. For each arc, perform an A* search [32] from the source to the destination nodes with a modified priority strategy that attempts to minimize congestion.
    (a) If the full width of the arc cannot be implemented due to insufficient wire capacity, determine the maximum width implementable.
    (b) "Use" this route by decrementing the slot and channel boundary capacities by the maximum available width.
    (c) Record this route as being used by this arc with the number of bits used.
    (d) If the full arc width cannot be implemented, return to step 2 and find a route for the remaining width.

The detailed algorithm for mapping arcs is described in Algorithm 3.

This algorithm tends to favor the placement of "preferred" routing arrangements for arcs with source and destination modules placed further apart. The "preferred" routing paths are chosen by choosing the exit boundary of the slot (line 5–13).

It is preferred for communicating modules in the same column to use the side channels, or if they are vertically adjacent, to use the channels between

**Algorithm 3** MapSubgraphToDevice

```
Input: A subgraph $G = (V, A)$
 1: Sort all arcs in A in descending order of length
 2: for each arc $a \in A$ do
 3:   repeat
 4:     Route from the source to the destination slot using a modified A*
        search where $g(x)$ and $h(x)$ is the Manhattan distance to the goal
        calculated using the vcost and hcost attributes with the following
        preferred exit heuristic:
 5:     if the source and destination are in the same column then
 6:       if they are vertically adjacent then
 7:         Exit Preference = { N/S(DIRECT), SIDE, CENTER }
 8:       else
 9:         Exit Preference = { SIDE, N/S(DIRECT), CENTER }
10:       end if
11:     else
12:       Exit Preference = { CENTRE, N/S(DIRECT), SIDE }
13:     end if
14:     if the route cannot be found then
15:       Return UNSUCCESSFUL
16:     end if
17:     $u \leftarrow a.width$
18:     for all cells $c$ crossed along the route do
19:       $p \leftarrow c.\{nesw\}cap - c.\{nsew\}util$ {the remaining capacity of the
          boundary that has to be crossed}
20:       if $p < u$ then
21:         $u \leftarrow p$
22:       end if
23:     end for
24:     for all cells $c$ crossed along the route do
25:       $c.\{nsew\}util \leftarrow c.\{nsew\}util + u$
26:     end for
27:     $a.width \leftarrow a.width - u$
28:     Record the route taken by these $u$ bits of arc $a$
29:   until $a.width \leq 0$
30: end for
```

the slots in order to minimize pressure on the center channel. Communicating modules in different columns use the center channels as a first preference as these have the shortest routing lengths. As long as the correct exit boundary is chosen, the A* search coupled with the Manhattan distance heuristic will result in the correct choice of channels.

### Time Complexity

The mapping step consists of two parts — allocating the merged modules to the device slots and routing the wires.

Allocating the merged modules using an ILP solver is of exponential time complexity, but using a standard cell placer such as Capo [71] requires $p \log p$ time for $p$ pins. For a given channel width, the number of pins is bounded by the device size and is a function of the number of rows in the device $r$ and the number of columns $c$.

An A* search is performed to determine a global route for each connection. The time complexity of the search is $O(q \text{ log } q)$ per route, where $q$ is the number of *channel cells* in the device (see Section 4.4). Since $q$ is proportional to the number of slots $z$, the complexity of the search is therefore $O(z \text{ log } z)$ per route.

The number of connections in the merged subgraph is $O(z^2)$. Thus the time complexity of subgraph mapping is $O(z^3 \text{ log } z)$.

### 6.6.5 Wire Delay Cost Model

The delay over a given distance is less when longer wires rather than shorter wires are used because less switch boxes are traversed. Since there is a limited number of long wires, the cost model described in this section is applied to impose an increasing penalty the more a channel is saturated.

$$t_{wd}(G) = max\left(\sum_{c \in channels\_used(a)} t_{pd}(c)\right) \qquad \forall a \in A_G \tag{6.1}$$

Equation 6.1 states that the critical path delay $t_{wd}$ of a subgraph $G$ is the maximum delay of any arc. The delay of a single arc is the sum of all the estimated costs of each channel $c$ used by the wire $t_{pd}(c)$.

$$t_{pd}(c) = channel\_cost(c) \times \left[1.0 + \frac{\theta \cdot num\_wires\_exiting\_channel(c)}{channel\_boundary\_capacity(c) \cdot \sigma}\right] \tag{6.2}$$

Equation 6.2 is based on Lou's definition of the congestion cost, i.e., the ratio between the number of wires used and the capacity in a routing channel [55]. In addition, it introduces a "reasonable saturation rate" $0 < \sigma \leq 1$ and an "optimality factor" $\theta \geq 0.0$ to increasingly penalize the wire delay the more the channels become saturated. The "reasonable saturation rate" $\sigma$ specifies what percentage of the total number of available wires is reasonably used. The "optimality factor" $\theta$ specifies what percentage penalty of the channel cost should be applied as the number of wires approaches the saturation rate. The cost is calculated after the mapping is performed.

The delay of a Virtex-4 wire is approximately 80$ps$ per CLB distance. Each switchbox hop also takes up about 80$ps$. These figures are estimates and were obtained experimentally by running the delay mediator in the Xilinx FPGA Editor for typical routes. If the propagation delay between adjacent CLBs is about the same as a switchbox hop, then using only single wires may cause the delay to be close to double that as compared to using longer wires. In this case, if $\sigma$ is 1.0 then $\theta$ should be close to 1.0. In other words, if all the wires are used, then it is possible that delay is doubled.

### 6.6.6 Reconfiguration Delay Cost Model

In order to estimate the reconfiguration time across the application, a cost model must be defined to provide a metric for reconfiguring modules and wires. The basis of this model is the Virtex-4 routing architecture and reconfiguration mechanism.

#### Virtex-4 Routing Architecture

Each CLB in a Virtex-4 device contains a switch box. The logic resources (i.e., LUTs, flip-flops) in each CLB are attached to the switch box via a connection box. Each switch box is in turn connected to other switch boxes through the global routing resources available on the device. The global routing resources as a whole are referred to by Xilinx as the General Routing Matrix (GRM). The numbers and types of *outbound* wires per CLB are listed in Table 6.1. The layout of the routing resources is generally homogeneous, thus there are about 24–25 outbound wires in each direction (north, south, east and west) per CLB.

| Wire Type | Count (Per CLB) | Span |
|---|---|---|
| OMUX (Single) | 16 | 1 |
| DOUBLE | 40 | 2 (can be tapped at 1) |
| HUNIHEX (Horizontal Hex) | 20 | 6 (can also be tapped at 3) |
| VUNIHEX (Vertical Hex) | 20 | 6 (can also be tapped at 3) |
| HLONG (Horizontal Long) | 2 | 18 (can also be tapped at 6, 12) |
| VLONG (Vertical Long) | 1 | 24 (can also be tapped at 6, 12, 18) |

**Table 6.1:** Virtex-4 routing resources (outbound)

A switch box is composed of Programmable Interconnect Points (PIPs), and the wires connecting these PIPs. Each PIP is connected to one or more wires in the GRM. A switch box is not a full crossbar, i.e., not every PIP is connected to every other PIP. An example of a switch box and how a connection is to be made is depicted in Figure 6.9. The circles on the edges of the switch box represent PIPs, and the arrows represent examples of wires connected to those PIPs. This is a simplified version of a switch box. An actual Virtex-4 switch box contains 307 PIPs.

In Figure 6.9 only the wires connected to PIPs C and H are shown. PIP H is an inbound PIP which can receive an input from either the single wire "OMUX18" or the hex wire "VUNIHEX21" in the GRM. The PIP acts as a multiplexer to select its input, thus the configuration bits for a PIP are the respective selection and enable bits. PIP C is an outbound PIP that can receive an input from PIPs 7, H or J, and provide outputs to three wires in the GRM, a long (HLONG1), a double (DOUBLE19) and a single (OMUX12). Each inbound PIP receives input from one or more wires in the GRM or the logic resources in the CLB, and provides output to one or more PIPs in the switchbox. Accordingly, each outbound PIP receives input from one or more PIPs in the switchbox and provides output to one or more wires in the GRM or the logic resources in the CLB. As an example, consider that

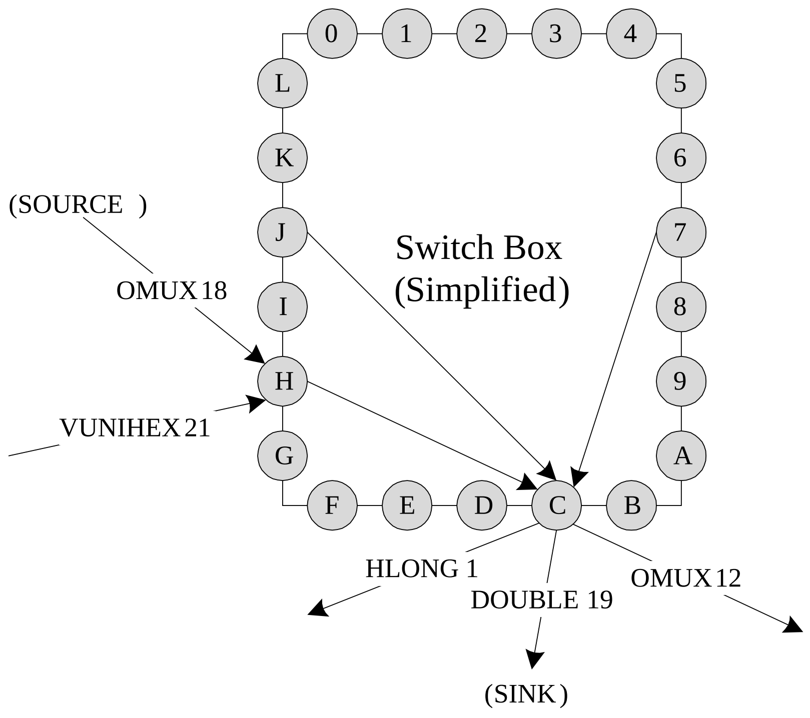


**Figure 6.9:** Virtex-4 switch box connection example (simplified)

the switchbox connects the single eastbound wire “OMUX18” to the double southbound wire “DOUBLE19”. To make this connection, PIP H must be enabled and configured to select input from OMUX18, and PIP C must be configured to receive input from PIP H. PIP C provides output to all three of its connected wires in the GRM, and all of them can tap its input, thus if the double wire “DOUBLE19” should receive its input, then the inbound PIP that is connected to “DOUBLE19” in the switchbox two CLBs south must select it.

**Virtex-4 Reconfiguration Architecture**

A Virtex-4 device is reconfigured in frames, each spanning 16 CLBs vertically and approximately $\frac{1}{22}$ of a CLB horizontally. In the 22 frames it takes to reconfigure a vertical column of 16 CLBs, routing information is contained in the first 20 frames and logic in the last 2 frames [6]. Since there are 307 PIPs per switch block, each frame theoretically configures an average of 15–16 PIPs.

Having 64 CLB rows and 24 CLB columns, an XC4VLX15 device, as shown in Figure 6.10, has 4 different "reconfiguration rows". The second "reconfiguration row" from the top is shown subdivided into CLB-width reconfigurable frames for illustrative purposes. The channel width shown here is 4.

The base variable for reconfiguration delay is the length of time to load and reconfigure a frame, $r_f$. Reconfiguring via the SelectMAP-32 interface at 100MHz, $r_f = \frac{41}{100 \times 10^6} = 0.41\mu s$, as there are 41 words (32 bits per word) in a frame.

In the EAPR tool flow, reconfiguring a slot entails a reconfiguration of the entire slot. With reference to Figure 6.10, the cost to reconfigure a module is thus $r_{slot} = 22 \cdot r_f \cdot \left(\frac{devcols}{2} - channelwidth\right)$. For the XC4VLX15 with a channel width of 4, $r_{slot} = 72.16\mu s$.

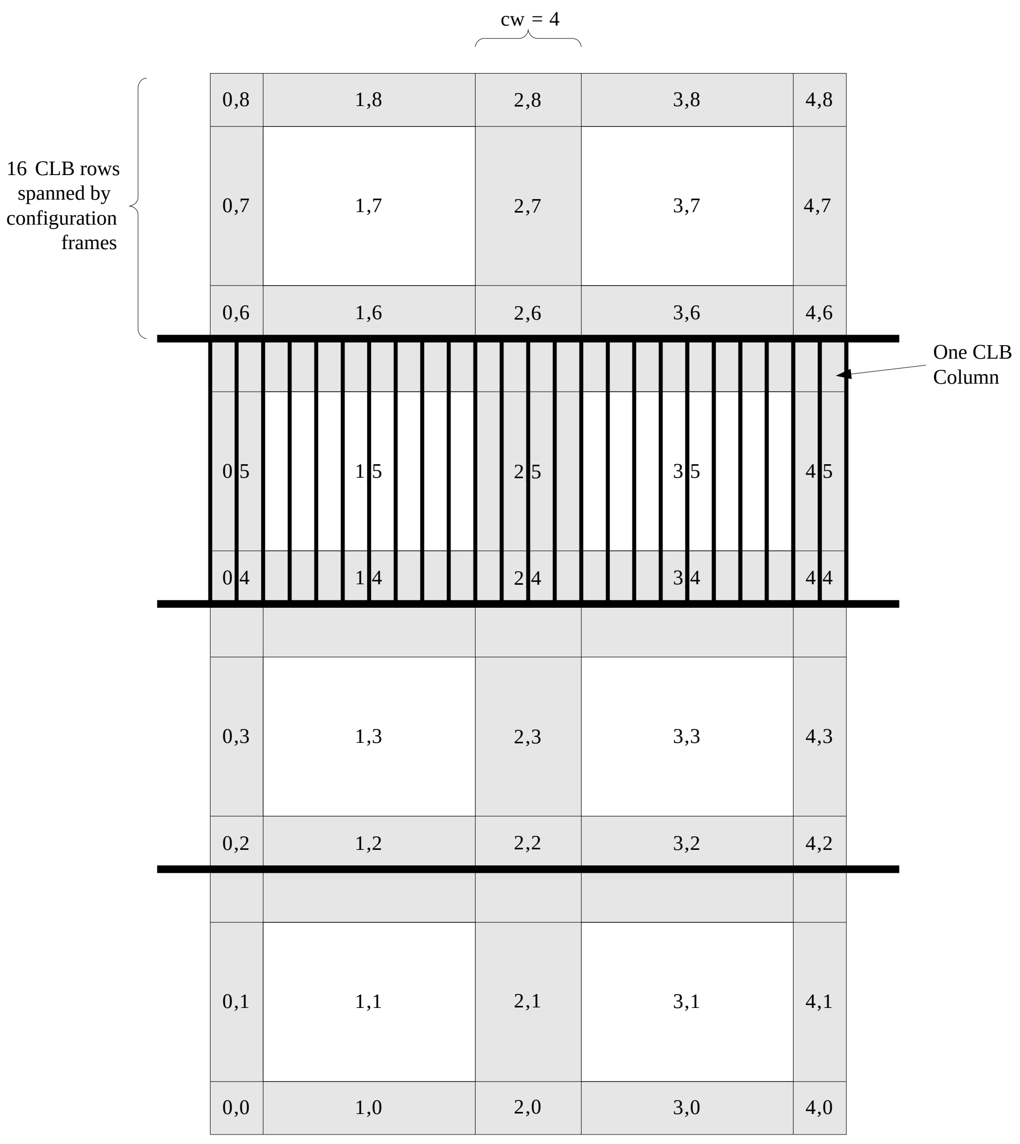


**Figure 6.10:** Reconfiguration areas

**Cost Model**

The model for reconfiguring wires is derived from the way the routing estimates are performed in Section 6.6.4. Consider the connection from M1 to M2 in Figure 6.11 as an example.

The shaded area in Figure 6.11 depicts the area that contains the configuration frames required to reconfigure the connection in the top reconfiguration row of the device. Because there is a bend in the connection to move south, this area encompasses about half of the frames in channels (2,8) and (2,7) and a quarter of (2,6). In Figure 6.11 the bend is nearer the left of the channel, thus this area is less than half of the channel cell. When the connection bends again to reach the sink module M2 in channel (2,3), the reconfiguration area is slightly more than half of the channel as it was less than half in channel (2,8).

Note that the shaded areas represent the areas that *contain* the frames required to reconfigure the wire, and does not imply that *all* the frames in the areas need to be reconfigured. The worst-case scenario is that the connection is implemented only using single wires, which implies that two PIPs need to be reconfigured per CLB. It is likely that the configuration data for these two PIPs are located in different frames, thus this thesis uses the heuristic estimate of two frames per CLB to reconfigure a connection. However, no more than 20 frames per column of 16 CLBs needs to be reconfigured no matter how many connections are present in that column.

This is an overestimate as it is unlikely, if not impossible, that all the connections are implemented using single wires because a detailed router

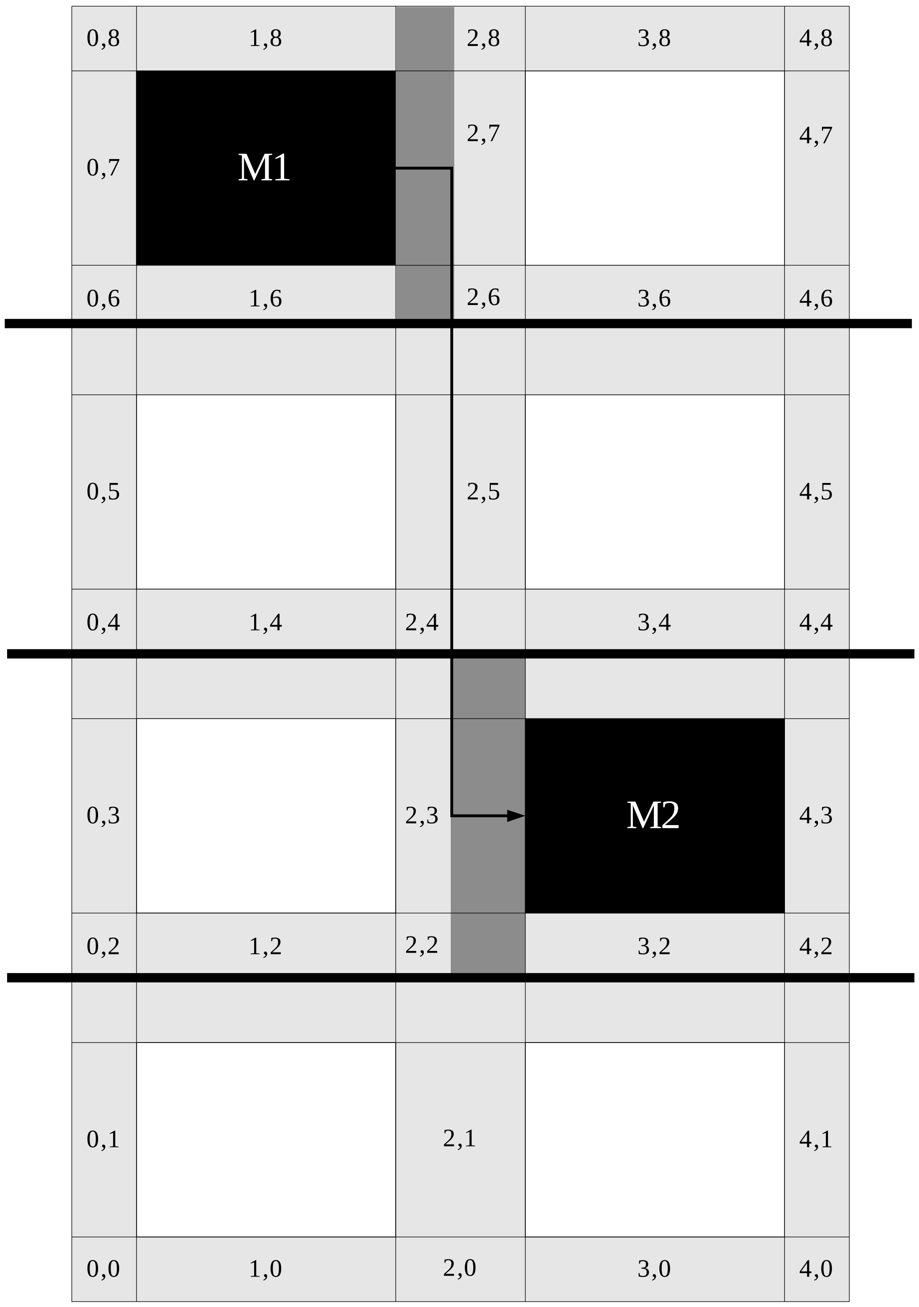


**Figure 6.11:** Wire reconfiguration

will undoubtedly use longer wires if they are available. In addition, PIPs in the same CLB column may also be configured with the same frames. However, overestimating the reconfiguration delay at this stage is a much better alternative to underestimating it as it increases the probability of a feasible implementation at the lower-level implementation stages.

The software implementation used for the experiments described in this thesis divides the reconfiguration delay by a relaxation factor (that defaults to 1) so that a system designer can reduce the estimated reconfiguration delay to more accurately represent the actual reconfiguration delay. This is useful if the estimated reconfiguration delay exceeds timing constraints, preventing the design flow from moving on to the next stage. The system designer can increase the relaxation factor to allow the application to be implemented in the lower-level stages described in Chapter 9, which provide actual reconfiguration delay results. The design flow can then revert back to this stage if the actual delays exceed timing constraints to try other possibilities.

The reconfiguration delay model is defined as such:

- **Slot reconfiguration:** A flat reconfiguration delay of $r_{slot} = 22 \cdot r_f \cdot \left(\frac{devcols}{2} - channelwidth\right)$. This accurately estimates the slot reconfiguration delay as the EAPR tool flow generates partial bitstreams that reconfigure the entire slot.
- **Wire reconfiguration:** For each channel that a connection occupies:
  - For horizontal segments of the connection, 2 frames for every CLB the connection crosses up to a maximum of 20 frames per column of 16 CLBs.

- If there is a bend in the connection, 2 frames are required for the bend. In the device model used, a bend in a channel always requires a subsequent bend in another channel.
- For vertical segments of the connection, a fixed estimate of 4 frames (i.e. $\frac{1}{5}$ of the CLB routing resources), obtained through experimentation with difference bitstreams.

The wire reconfiguration delay is estimated assuming that a *difference reconfiguration* is carried out between periods rather than a full reconfiguration of the device. Difference-based reconfiguration is supported by the EAPR tool flow and results in significantly lower reconfiguration overheads. With difference-based reconfiguration the reconfiguration overhead is minimized not only in the wiring but also in the modules. This is especially true as it has been noted by Malik [58] that bitstream differences between modules can be very small, in many cases less than 5%. Again, this implies that the reconfiguration delay model overestimates the delay and can be fine-tuned to the application's needs as previously indicated.

## 6.7 Optical Flow Case Study and Evaluations

Optical flow computations calculate the velocity between pixels in successive frames of a video stream obtained from a camera source, mounted, for example, on an unmanned vehicle. Each frame goes through a two-step process — first it is smoothed and prepared into tensors (products of pixel values), which are then fed into an iterative process until the optical flow converges.

In previous studies it was noted that placing the entire application onto a single device has requirements that greatly exceed the device area (by about 8 times) and on-chip RAM (by about 15 times) of a medium-sized Virtex-4 device [67]. Dynamic reconfiguration through hardware virtualization and off-chip buffering is suggested as a means of overcoming these constraints. The feasibility of this approach was tested through a case study.

### 6.7.1 Graph Preparation

A block-level design was constructed of an implementation of the Gauß-Seidel method [15] with successive overrelaxation. This was then transformed into a communications graph. Each block was implemented in VHDL and synthesized to the smallest Virtex-4 device (XC4VLX15) in order to obtain FPGA-resource estimates.

The module clustering stage was performed using METIS [45] and some manual adjustments for customization purposes and to ensure that the application behaves correctly and as efficiently as possible, e.g., the decision was made not to cluster some nodes to reduce reconfiguration delay. The resulting clustered graph has 51 nodes, each of which fits into a $10 \times 14$ CLB slot. In contrast, the device provides 8 COMMA slots corresponding to the 8 natural pages on the device.

The scheduling stage was carried out using a modified version of the time-based algorithm of Purna et al., again modified to improve the efficiency of this application. The resulting schedule had 8 partitions, each containing 6 to 8 nodes. Partitions 1–4 form the pre-processing stages and partitions 5–8 form the iterative stages.

## 6.7.2 Graph Merging Results

The scheduled graphs were processed using Algorithm 1 of Section 6.6. The characteristics of each graph are reported in Table 6.2. The eight subgraphs were merged into three periods (1-3, 4-6 and 7-8) with one wiring harness each. It must be noted that the parameters used were conservative with respect to the area and timing constraints so as to have a higher chance that the merged subgraphs could be physically implemented by the low-level place-and-route tools.

**Table 6.2:** Optical flow comparison results

| Subgraph | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| [Orig.] Module | - | 504$\mu s$ | 360$\mu s$ | 504$\mu s$ | 432$\mu s$ | 432$\mu s$ | 432$\mu s$ | 144$\mu s$ |
| Reconfig. Delay | *(432$\mu s$)* | - | - | - | *(360$\mu s$)* | - | - | - |
| [Orig.] Harness | - | 234$\mu s$ | 201$\mu s$ | 322$\mu s$ | 221$\mu s$ | 261$\mu s$ | 361$\mu s$ | 116$\mu s$ |
| Reconfig. Delay | *(392$\mu s$)* | - | - | - | *(232$\mu s$)* | - | - | - |
| [Orig.] Crit. Path | 4.0 ns | 5.1 ns | 6.2 ns | 4.0 ns | 4.6 ns | 5.7 ns | 5.6 ns | 5.6 ns |
| [Merged] Module | - | 216$\mu s$ | 0$\mu s$ | 360$\mu s$ | 360$\mu s$ | 0$\mu s$ | 72$\mu s$ | 0$\mu s$ |
| Reconfig. Delay | *(288$\mu s$)* | - | - | - | *(72$\mu s$)* | - | - | - |
| [Merged] Harness | - | - | - | 321$\mu s$ | - | - | 398$\mu s$ | - |
| Reconfig. Delay | *(302$\mu s$)* | - | - | - | *(298$\mu s$)* | - | - | - |
| [Merged] Crit Path | 6.8ns | | | 6.2ns | | | 6.0ns | |

The rows prefixed with "Orig." report the results when no merging was performed, i.e., each graph has its own wiring harness. These are compared to when the subgraphs were merged into periods, as reported in the rows prefixed with "Merged".

Note that there are two subrows of values for each row reporting the reconfiguration delays for the modules and wiring harnesses. A reconfiguration delay value in each top subrow (with non-parenthesized values) represents the reconfiguration time into a subgraph from the one before it. The reconfiguration delay for Subgraph 1 is not reported as it is the initial configuration and is performed before the application is executed. In each bottom subrow (with parenthesized values), the value for Subgraph 1 represents the delay to reconfigure the system to the initial state when an entire video frame has been processed, i.e., the reconfiguration delay form Subgraph 8 to Subgraph 1. Also in each bottom subrow, the value for Subgraph 5 represents the delay from the end of each loop in the iterative stages to the beginning of the loop, i.e., from Subgraph 8 to Subgraph 5. Note that in the merged case, wiring harness reconfiguration is only necessary when transitioning between merged graphs.

There are several $0\mu s$ entries after merging e.g. reconfiguring from partition 2 to partition 3 because all the required modules for partition 3 were already configured during partition 2.

The "Crit. Path" rows report on the estimated critical path delay obtained through the mapping process (described in Section 6.6.4) of the *wiring harness* for each configuration. Note that merged configurations share the same critical path delay as they use the same wiring harness.

Fig. 6.12 shows the reconfiguration times as the application iterates between graphs. The pre-processing stage starts at $G_1$ and runs through to $G_5$ where the iterative stage begins. The iteration loops from graphs $G_5$ through to $G_8$ until convergence, before a new frame commences processing at $G_1$. Assuming that convergence is reached in 100 iterations, the total reconfiguration time is shown with an overall improvement of 64%.

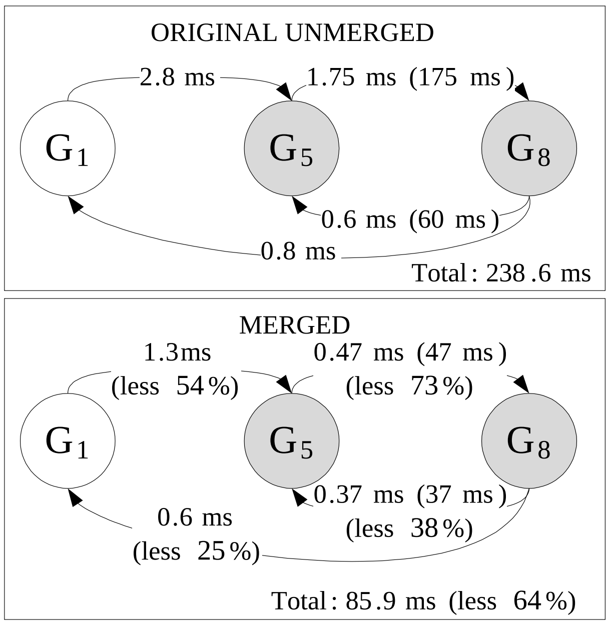


**Figure 6.12:** Reconfiguration times for iterations

### 6.7.3 Analysis

Looking at the individual graph timings, the wiring harness reconfiguration times are small compared to those of the modules because the channel width chosen was small. The harness reconfiguration time is approximately equivalent to that without merging because most of the channels are already used by the unmerged graphs. The critical path delays are only slightly higher in the merged case. We believe that an actual router is also likely to display this difference as the extra delay is due to congestion rather than wire length.

The total timing results show a large improvement in terms of reconfiguration time (64%), especially since a lot of savings are achieved during the iterative stages. In the unmerged case, an optimization step can be performed to reduce the number of module swaps, but this may be detrimental to the critical path delay as the placement of the modules is not optimized for each configuration. This performance hit may grow significantly as the device size scales.

The smallest possible device was used in this case study to examine the results when a large application is mapped to it. One of the goals in COMMA is to use its framework to reprocess the communications graph with a variety of device sizes to determine the best balance of device size and performance. This will allow designers to choose the most economical device for their needs. A study in this direction is presented in Chapter 8.

## 6.8 Conclusion

The infrastructure generation flow was described in detail in this chapter. A central process in the flow, known as subgraph merging, groups subsequences of subgraphs together and creates a wiring harness for each subsequence. Algorithms were described for subgraph merging and its two constituent subprocesses — merging two subgraphs and subgraph mapping.

The advantage of subgraph merging was demonstrated in terms of significantly reduced overheads when applied to the mapping of a large optical flow communications graph to a very small device. These experimental results were produced using high-level tools. Positive trends are expected to be reproduced when the produced designs are mapped to devices using vendor tools, e.g., the EAPR tool flow.

The greedy heuristics described in this chapter for merging communications graphs and mapping these merged graphs to the device could be improved upon. An improved algorithm, based on dynamic programming, is described in Chapter 7. A thorough investigation of the reconfiguration overheads and critical path delays achieved by COMMA over a range of device sizes for both algorithms is presented in Chapter 8.

# Chapter 7

# Improved Algorithm for Subsequence Grouping

## 7.1 Introduction

The subgraph merging process was introduced in Chapter 6 as a means of deriving wiring harnesses from an application specified as a schedule of subgraphs. The main process in subgraph merging groups subsequences of contiguous subgraphs together to form *periods*. A single wiring harness is implemented for each period that supports all the communications requirements for every subgraph in it. Application execution proceeds by configuring a wiring harness and the modules for the first subgraph in a period. For the remaining subgraphs in the period, only module reconfiguration is required. At the end of the period the wiring harness for the next period is configured along with the module reconfigurations needed to implement the first subgraph of the following period.

The algorithm used for subsequence merging introduced in Chapter 6 is based on a greedy method. Starting from the first non-merged subgraph, the method tries to merge as many subgraphs into a period as possible without exceeding wiring harness area or delay constraints. This is sub-optimal as there is no guarantee that always trying to merge the first and second subgraph together at each iteration results in the lowest reconfiguration delay. Better period arrangements that do not always begin with the first unmerged subgraph may be possible. For example, periods could be chosen to exploit similarities in structure between consecutive communications subgraphs.

The problem looks suited to a dynamic programming approach [20] in which the solutions to longer sequences are formed by combining the solutions to shorter sequences. Unfortunately, the solutions to shorter sequences cannot be readily concatenated since the reconfiguration cost at the start of a period depends, in part, upon the wiring harness of the previous period and the arrangement of configured modules at the end of the previous period. It is therefore difficult to assign a fixed cost to a given merged subsequence. However, this problem can be overcome by imposing a heuristic assumption that at the start of each period a complete reconfiguration of the FPGA is performed to implement the wiring harness and the module arrangement for the first subgraph of the period. With this assumption in place, the cost of merging a subsequence of subgraphs can be assumed to be independent of the previous period and the problem assumes optimal substructure.

It then becomes the goal of the algorithm to minimize the sum of the module reconfiguration times within the periods, and the total number of periods required to implement the schedule. When the algorithm estimates the reconfiguration cost, starting a new period will always have a higher reconfiguration delay than reconfiguring between subgraphs in a period. However, it is not always the case that the minimum number of periods results in a total minimum reconfiguration delay as the module-slot allocations within each period are different. Thus, different groupings with a larger number of periods should be considered as well.

A paper describing this algorithm and assessing its performance has been accepted for presentation at the 2008 IEEE International Symposium on Field-Programmable Custom Computing Machines [53].

## 7.2 Dynamic Programming Algorithm

Let a *split* be defined to be a position between two subgraphs in the schedule where there is a possibility of ending a period and starting another. Let a split at position $k$ be defined as a split between subgraph $k$ and subgraph $k+1$ in the scheduled subgraph sequence.

Consider a scheduled subgraph sequence of length $n$.

Establish the $n \times n$ memoization table. The $i^{\text{th}}$ row in this table corresponds to all subsequences of length $i$ considered for splitting into optimal periods. Entries in column $j$ at row $i$ record the optimal split arrangement and the corresponding total reconfiguration delay for splitting a subsequence of length $i$ commencing with subgraph $j$ in the schedule. Note that the total reconfiguration cost does not include the delay of the initial complete reconfiguration required to configure the wiring harness for the first period in the subsequence and the modules for the first subgraph in the subsequence.

Record a reconfiguration delay $r_{\text{opt}} = 0$ for each element of the first row of the table. For each subgraph in the schedule, this records the *zero cost* of configuring its modules after the complete configuration undertaken to implement the wiring harness and the modules for the period consisting of the subgraph $j$ on its own.

For all subsequences of the schedule of length $i : 1 < i \leq n$, the algorithm does the following (without loss of generality, let the subsequence under consideration span subgraphs $G_1$ through $G_i$):

1. Consider forming a period over the entire subsequence using the algorithms described in Sections 6.6.3 and 6.6.4. Let $r_0$ be the reconfiguration delay incurred by reconfiguring the modules for all the subgraphs in the subsequence. If merging all subgraphs in the subsequence exceeds area/time constraints on the wiring harness, then let $r_0 = \infty$.

2. Consider every possible position $k : 1 \leq k \leq i - 1$, for a single split in the subsequence. Determine the reconfiguration cost $r_k$ for that position by adding the following three cost components:

   (a) the reconfiguration cost of the optimal arrangement of splits for the subsequence of subgraphs $1 \ldots k$, i.e. for the part of the subsequence to the left of the split $k$, as determined when subsequences of length $k$ were considered,

   (b) the reconfiguration cost of the optimal arrangement of splits for the subsequence of subgraphs $k + 1 \ldots i$ (to the right of $k$, found for subsequences of length $i - k$), and

(c) the cost of the full reconfiguration that is incurred when commencing a new period after position $k$.

3. Let $r_{\text{opt}}$ be the minimum of $r_0 \dots r_{i-1}$, which is memoized along with the corresponding split arrangement in the dynamic programming table at cell $(i, j)$.

Thus, as longer subsequences are considered, the estimation of the reconfiguration delay relies on the memoization of shorter subsequences. The best splits at length $n$ finally indicate the optimal set of periods for the scheduled subgraph.

An example of this memoization is depicted in Fig. 7.1. The iteration currently considered is at a subsequence length of $i = 7$ commencing at subgraph $G_1$. If a split were to be placed at $k = 3$, the optimal split arrangement for subgraphs $G_1$ to $G_3$ obtained from the iteration $i = 3$ is used to the left of the split. Correspondingly, the optimal splits for subgraphs $G_4$ to $G_7$ obtained from the iteration $i = 4$ are used to the right of the split at $k = 3$. The total reconfiguration delay is calculated by adding the previously calculated values from the split arrangements from $G_1$ to $G_3$ and $G_4$ to $G_7$ together with the full reconfiguration delay incurred between $G_3$ and $G_4$.

The dynamic programming algorithm is specified in detail in Algorithm 4.

### 7.2.1 Complexity

Half the memoization table needs to be filled in. To fill in an entry, a trial merge of all subgraphs in the sequence is attempted during Phase 1 and on the order of $n$ trial splits and cost comparisons need to be performed in Phase

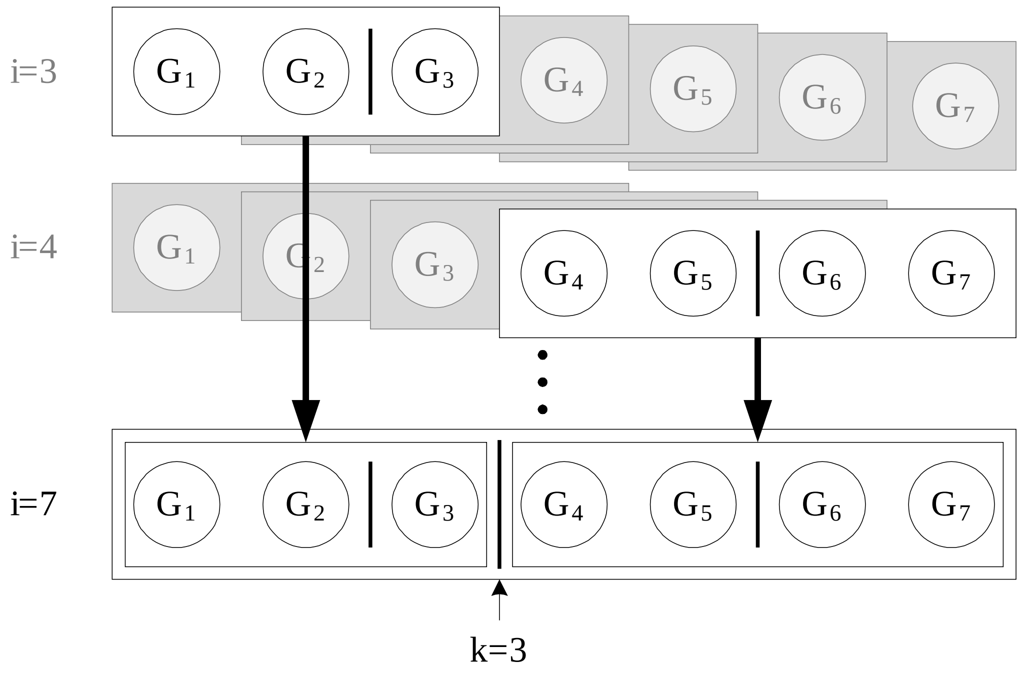


**Figure 7.1:** Memoization example in the dynamic programming algorithm

2. Phase 2 thus contributes $O(n^3)$ to the time complexity of the dynamic programming algorithm.

The merging step involves sorting and then linearly processing the modules for each subgraph being merged (see Section 6.6.3). As the number of modules in a subgraph is a constant dependent upon the device size, it can be argued that the merging therefore takes $O(n)$ time for $n$ subgraphs. Mapping the merged subgraph involves allocating the merged modules to the device slots and routing the wires, and as described in Section 6.6.4, is dependent on the device size, which again is constant with respect to the number of subgraphs. Thus, for a given device size, the merging and mapping step combined takes time proportional to the number of subgraphs being merged, which is $O(n)$.

As Phase 1 can be abandoned beyond a certain subsequence length, it contributes $o(n^3)$ to the dynamic programming algorithm, which therefore has $O(n^3)$ overall time complexity.

### 7.2.2 Optimality

The algorithm is optimal when the simplification of a full reconfiguration between periods is assumed. When the periods are actually implemented on an FPGA, rather than performing a full reconfiguration of the device at period start, a difference reconfiguration is performed, and thus the algorithm overestimates the reconfiguration delay. This is beneficial when the application is actually implemented on the device, but the impact of applying the heuristic simplification should be analyzed.

If this simplification were not applied, then the reconfiguration delay between periods could be reduced in two foreseeable ways: by maintaining the same module allocation between periods, and by trying to implement wiring harnesses that exhibit minimal difference. Investigating algorithms that deal with this additional complexity is a challenging area for further work.

If module allocations were to be maintained between periods, this would almost certainly result in different merging arrangements. This particular algorithm will no longer exhibit optimal substructure because each period depends on the one before it. A different algorithm would therefore need to be employed.

However, trying to implement wiring harnesses that exhibit minimal difference is a natural extension to this algorithm as it will simply result in the actual reconfiguration delays being reduced.

## 7.3 Summary

This chapter introduced a dynamic programming algorithm for merging subsequences of subgraphs as an improvement over the greedy method. The algorithm tries to minimize the total number of wiring harnesses by assuming that a full reconfiguration of the device occurs between periods. An additional advantage of this new algorithm is that it considers many more possible subsequence groupings than the greedy method. Keeping the device size constant, the time complexity increases to $O(n^3)$ from $O(n)$ for the greedy method. Note that this disregards the $z^3\ logz$ factor due to merging and mapping. The quality of the resulting merge is assessed in Chapter 8.

**Algorithm 4** Dynamic programming algorithm

```
1: Create an array $Splits[n, n]$
2: {Dimensions — [subsequence length, start position]}
3: for $i = 1$ to $n$ do {$i$: subsequence length}
4:    for $j = 1$ to $n - i + 1$ do {$j$: start position}
5:       if $i = 1$ then
6:          $Splits[i, j] \leftarrow 0$
7:       else
8:          {Phase 1: Whole subsequence merged}
9:          Create period $P_{j,j+i-1}$
10:         if $map\_success(P_{j,j+i-1})$ then
11:            $MinRecfg \leftarrow EstimateRfgDelay(P_{j,j+i-1})$
12:         else
13:            $MinRecfg \leftarrow \infty$
14:         end if
15:         $BestSplits \leftarrow 0$
16:         {Phase 2: Determine best split}
17:         for $k = 1$ to $i - 1$ do {$k$: splitposition}
18:            $TestSplits \leftarrow Splits[k, j] \mid Splits[i - k, j + k]$
19:            $S \leftarrow CreateSolutionInstance(TestSplits)$
20:            $CurrentRecfg \leftarrow EstimateRecfgDelay(S)$
21:            if $CurrentRecfg < MinRecfg$ then
22:               $BestSplits \leftarrow TestSplits$
23:               $MinRecfg \leftarrow CurrentRecfg$
24:            end if
25:         end for
26:         $Splits[i, j] \leftarrow BestSplits$
27:      end if
28:   end for
29: end for
```

# Chapter 8

# Results

## 8.1 Introduction

This chapter presents results of experiments that were performed to evaluate the graph merging method for generating wiring harnesses discussed in Chapter 6. This method consists of three subprocesses: subsequence merging, graph mapping, and the merging of two graphs. Both the algorithms presented for subsequence merging are assessed in these experiments — the greedy method introduced in Chapter 6, and the dynamic programming approach described in Chapter 7.

The goals of these experiments are:

- To assess the efficacy of graph merging as compared to *not merging*, i.e., where each configuration has its own customized wiring harness;
- To evaluate the performance of the dynamic programming approach as compared to the greedy method; and

- To assess the effectiveness of the point-to-point communications infrastructure as application and device parameters vary.

This chapter is organized as follows: the experimental method is introduced, and then the results are discussed based on the total reconfiguration delays of the tested applications, and the contributions to the critical path delay by the wiring harnesses generated for those applications. These metrics are most appropriate to a systems designer for evaluating the efficacy of the algorithms.

A subset of the results in this chapter were accepted for presentation at the 2008 IEEE International Symposium on Field-Programmable Custom Computing Machines [53].

## 8.2 Experimental Method

Benchmarks for dynamically-reconfigurable computing are not yet available, and manually developing applications such as the Optical Flow algorithm from Chapter 6 is too time-consuming. Thus, the methodology used to conduct these experiments involved generating synthetic applications with a variety of parameters representative of actual applications, and then subjecting these applications to the infrastructure generation process using the full range of device sizes and architectural parameters. This section describes the experimental setup, then details the methods used to generate communications graphs, cluster and schedule them, before describing and explaining the parameters chosen to perform the experiments.

### 8.2.1 Experimental Procedure

The overall goal of the experiments was to analyze the results obtained when a variety of target applications were mapped onto a range of different device sizes with a range of architectural parameters. In order to do this, the experimental procedure shown in Figure 8.1 was developed.

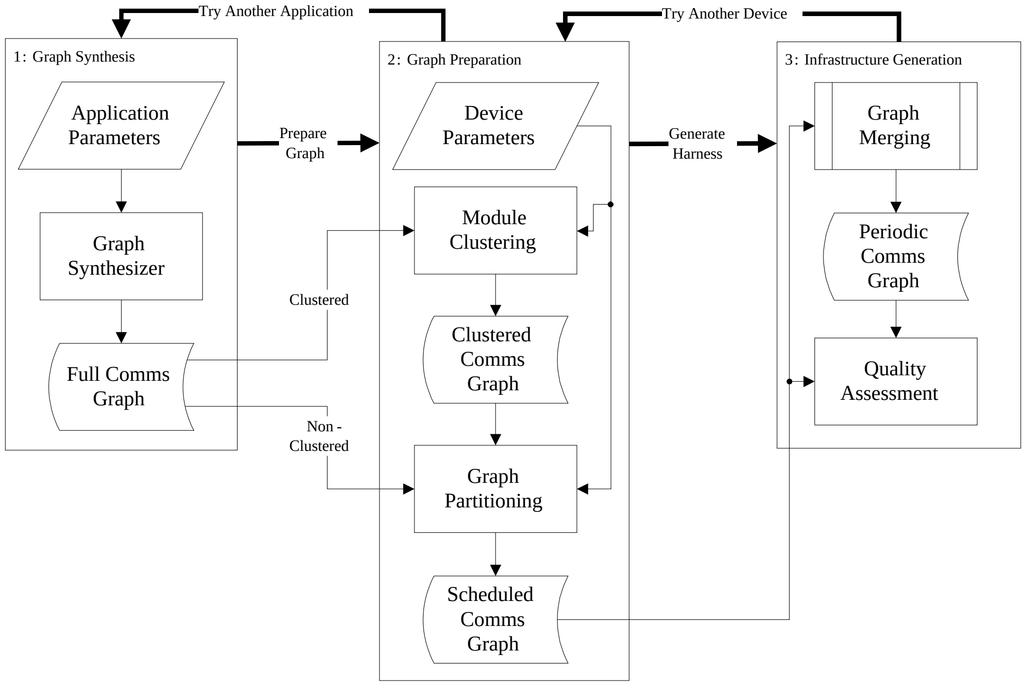


**Figure 8.1:** Experimental procedure

The experimental procedure consists of three main phases:

1. **Graph synthesis:** This phase generates a full applications graph based on parameters such as the number of modules and communications density. The graph synthesis process is described in detail in Section 8.2.2.

2. **Graph preparation:** The input to this phase is a full communications graph from the graph synthesis phase representing an application. An actual applications graph can also be used provided it is annotated with logic size specifications in each module node. This phase clusters and partitions the graph into a schedule of *subgraphs*. The two processes in this phase are described in Sections 8.2.3 and 8.2.4.

3. **Infrastructure generation:** The schedule of subgraphs produced by the graph preparation stage is then input into this phase where the wiring harnesses are generated. The "graph merging" process in this phase consists of the "subsequence merging", "merging two graphs" and "graph mapping" processes described in Sections 6.6 and 7.2. The outputs of the graph merging process are the period groupings, module-slot allocations and slice macro positions. The total reconfiguration delay of the application and critical path delays of the wiring harnesses are then assessed and recorded.

These three phases are executed iteratively. During Phase 1, a graph representing a single application is generated. During Phase 2, this graph is clustered if desired, and partitioned into a schedule of subgraphs based on a target device. If the partitioning is successful, the schedule is input into the next phase. If not, a different target device or channel width is chosen and Phase 2 is executed again. If all the available devices have been assessed, then execution returns to Phase 1, where a new applications graph is generated.

If Phase 2 is successful, the schedule of subgraphs is subject to graph merging and reconfiguration and critical path delay analysis in Phase 3 for a particular device. If this succeeds, the results are recorded, and execution returns to Phase 2 to assess another device.

A test *run* consists of obtaining the reconfiguration and critical path delay results for a particular application, a particular device and a chosen channel width.

The experimental procedure also allows for two different modes of operation in the graph preparation phase — with and without module clustering (see Section 6.4). If module clustering is not required, that step is skipped and the graph is partitioned and scheduled as is. Both modes are used for every test run, thus each test run yields two sets of results.

In summary, the experimental procedure iterates through application parameters at the first level of nesting at Phase 1, then through the list of devices and channel widths at the second level of nesting at Phase 2. Phase 3 is executed once per execution of Phase 2, and if and only if Phase 2 succeeds.

### 8.2.2 Graph Synthesizer

The graph synthesizer generates unpartitioned communications graphs for dataflow-like applications such as image processing (e.g., JPEG, Optical Flow) or cryptographic applications (e.g., DES).

These applications graphs are directed acyclic graphs that show the data dependencies between modules in time. They do not have discrete temporal representations of communications patterns, rather they capture the data that each module requires for its processing at any point in time. For example, if an application consists of two modules where module $A$ sends some data to module B, followed by module B sending some data back to module A, the generated applications graph has the following representation: $A \rightarrow B \rightarrow A$. This graph has three processing *stages*, the first one containing the first $A$, the second the module $B$ and the third stage the second $A$. There is no limit as to how many modules a stage can have, although it would not make sense to have a stage with no modules. It is important to note that even as this full communications graph has three modules there are only two module types. One of the modules simply appears twice in the execution of the application.

The input parameters to the graph synthesizer that were of particular interest to the experiments performed are:

- **Number of modules:** The total number of modules in the graph.
- **Module type variation:** The amount of variation in the types of modules in the graph, specified as a percentage. A variation of $p$% indicates that there are $p \times n$ module types (with a minimum of one) where $n$ is the total number of modules. E.g., a variation of 0% indicates that all modules are of the same type, and a variation of 20% with 200 modules indicates that there are 40 different module types.
- **Module size:** The average size of each module, specified in CLBs.
- **Module size variation:** An option to vary the size of the modules based on a standard distribution and a variance. The "module size" parameter was used as the mean. When the module size variation parameter was zero, all modules were of the same size.

The following other parameters to the synthesizer were available:

- **Number of stages or stage size:** The number of processing stages in the graph. A layered graph in which the modules in the same stage do not communicate with each other is assumed. Data is input only from modules in the previous stage, and output only to modules in the next stage. As the number of stages decrease, the average stage size increases. As a result, the total amount of communications in the graph increases if all other parameters remain constant. This is because the amount of inter-stage communication is higher if there are more modules in each stage. Consider an application where the communications density is 30% and the number of modules is 100. If there are 10 stages, the average stage size is 10 and there is an average of $10 \times 10 \times 30\% = 30$ arcs between each stage, and a total of about $30 \times 10 = 300$ arcs. In contrast, if there are 20 stages, the average stage size is 5 and there is an average of $5 \times 5 \times 30\% = 7.5$ arcs per stage and a total of $7.5 \times 20 = 150$ arcs.

- **Communications density:** The minimum desired percentage of the maximum number of edges between stages in the graph. 30% has been used for these experiments, meaning that each module connects to at least a third of the modules in the next stage.

- **Bitwidths:** The number of bits of each edge in the graph, specified as a minimum and maximum, in powers of two, with an exponentially decreasing distribution.

As the average stage size, communications density and bitwidths of the graphs increase, the total amount of intermodule communications in the graph also increases. Initial experiments that were performed over a wide variety of parameters indicate that there is a direct relationship between the parameters and the reconfiguration and critical path delays. The aim of the experiments described in this chapter is to determine what a systems designer might expect when an application is mapped onto different device sizes with alternate architectural parameters. Thus, the following fixed, realistic values were provided for the number of stages, communications density and bitwidths: 12% of the number of modules in the graph as the number of stages, 30% minimum communications density and bitwidths in exponentially decreasing distribution of 1, 2, 4, 8, 16 and 32 bits per edge.

### 8.2.3 Module Clustering

The input to the module clustering stage is the full communications graph, where each node represents a module in the application design. These nodes are then clustered together to form larger nodes, each of which do not exceed the size of the slots on the device. The main objectives of this are to reduce the total number of modules in the graph, to maximize the use of available logic area, and to reduce the total number of configurations.

The clustering algorithm of Purna et al. was used, which tends to reduce the total communication overhead by trying to cluster modules with common parents. Since the algorithm already limits the size of each partition based on the total logic area consumed, the size of each slot was used as the area constraint in the algorithm.

Thereafter, each ‘partition’ represented a new module type, and all the nodes within it were clustered together into a single node. It is assumed that all the intermodule connections between the nodes within a partition are absorbed into this clustered module. A new communications graph was then generated consisting of these clustered modules.

If a clustered module did not fill an entire slot, it was assumed that its logic was packed into the thinnest vertical slice of the slot that could accommodate it. This can be achieved by enforcing area constraints in the place-and-route tools. This was assumed in order to reduce reconfiguration delay as much as possible as the devices are reconfigured using vertical frames.

However, at the implementation level, the current Early-Access Partial Reconfiguration Tool Flow [95] requires that Partially-Reconfigurable Regions (PRRs) have fixed sizes. When a PRR is reconfigured, the partial bitstream for the entire region is loaded. Each slot must have a PRR, and it should be of the minimum size that can accommodate any clustered module that is to be placed into the slot. Thus, the reconfiguration delay of a slot is the reconfiguration delay of the largest clustered module that is to be placed in it.

### 8.2.4 Graph Partitioning

In this step, the input communications graph was partitioned into a schedule of subgraphs irrespective of whether the communications graph was clustered or not.

The experiments described in this chapter also used the clustering algorithm of Purna et al. for this step. However, the area constraint check was modified such that it generated partitions containing no more nodes than the number of available slots in the device. Each partition then represented a configuration to be loaded onto the device. The order in which the subgraphs were to be loaded was the same as the order in which the partitions were generated. This is valid because the algorithm is time-based and ensures that intermodule communication dependencies are resolved.

In order to model realistic application scenarios, two modules were added to each partition to buffer data between configurations. An output buffer module sends data into an on- or off-chip buffer for a future configuration, and an input buffer module reads this data from a previous configuration, also from on- or off-chip memory. As the results in this chapter present the reconfiguration and critical path delays, the actual time taken to buffer data on- or off-chip was considered to be part of the application run time and was therefore taken into account.

### 8.2.5 On the Quality of the Clustering and Partitioning Algorithms

The clustering and partitioning algorithms can affect the resulting schedule in several ways.

When modules are clustered together, two important effects are apparent. Firstly, the intermodule communications between the modules to be clustered together are absorbed. The amount of communications that is absorbed depends on how the clustering algorithm targets this objective. Secondly, new module "types" are created, which can affect how much reconfiguration delays can be reduced by trying to place the same module types in the same slots in adjacent configurations.

The reconfiguration delay between subgraphs depends on how much similarity exists between two adjacent configurations. When the graph is partitioned into a schedule of subgraphs, the similarity between each subgraph is affected by how the partitioning algorithm works. Both the similarity in the module types as well as the intermodule communications affect the reconfiguration delay. In addition, the critical path delay of each subgraph is affected by how dense the intermodule communications within it are. The partitioning algorithm may have to be mindful of this with applications that have very strict critical path delay constraints.

However, the main aim of architectural exploration in these experiments was to analyze the effects of graph merging on different applications and different devices and to evaluate the greedy and dynamic approaches toward the subsequence merging problem. Thus, the application-neutral clustering algorithm of Purna et al. was chosen. Furthermore, it is also suitable for the dataflow-like directed acyclic graphs, as were synthesized, and it is readily-integrated into the the iterative experimental procedure.

In real applications, different algorithms may be more suitable given the types and characteristics of those applications. For example, a combination of a modified version of the clustering algorithm by Purna et al., METIS [45] and manual adjustments made with knowledge of the application was used in clustering and partitioning the Optical Flow algorithm presented in Section 6.7.

Some effects of the neutrality of the clustering algorithm of Purna et al. are apparent in the results. This motivates further work in clustering and partitioning to target the COMMA approach.

### 8.2.6 Graph Merging

As detailed previously, the graph merging step consists of three processes: subsequence merging, merging two graphs and graph mapping.

Both subsequence merging algorithms, i.e., the greedy method introduced in Section 6.6.2, and the dynamic programming approach presented in Section 7.2, were used in order to compare their effectiveness. The experimental procedure also considered a third option, i.e., not applying the subsequence merging algorithm at all. In this case, each subgraph formed a single period with its own wiring harness. Judicious module-slot allocation was still performed to reduce module reconfiguration delay, but the wiring harness had to be reconfigured at every subgraph transition. Comparing the results of graph merging with the two subsequence merging algorithms with these results provides an unbiased assessment of the benefits of the subsequence merging algorithms.

Regardless of the subsequence merging algorithm, graph mapping was performed for each period and the reconfiguration and critical path delays were assessed for each period, as described in Section 6.6.4.

### 8.2.7 Quality Assessment

This process obtained the critical path and reconfiguration delay results from the graph mapping process in the graph merging stage, recorded them, and processed them into tables and graphs that are presented in the later sections of this chapter where the results are analyzed.

### 8.2.8 Parameters Chosen for the Experiments

#### Application Parameters

The parameters used to generate the applications graphs were as follows:

- **Number of Modules:** 200. Considering large graphs allowed full architectural exploration to be performed.
- **Module Type Variation:** Primarily 20% (i.e., 40 module types) to observe the effects of reducing reconfiguration delay by allocating modules belonging to the same type in neighbouring epochs in the same slots. 0%, 40% and 60% were also used to observe the effects of different amounts of variation.
- **Module Size:** Primarily 60 CLBs. This is approximately the size of a DES core when mapped onto a Virtex-4 device. As a comparison, a MicroBlaze processor takes up 226 CLBs, which is slightly larger than a slot on a medium sized XC4VLX40 with a channel width of 2 (224 CLBs). Module sizes of 35 and 85 CLBs were also used to observe the effects of different amounts of clustering.

- **Module Size Variation:** Primarily fixed but one experiment investigated the effect of varying the module size according to a normal distribution with a 10 CLB standard deviation.

### Device and Architecture Parameters

Each application subgraph was mapped onto all of the available devices in the Virtex-4 LX series. The LX series was chosen to be most suitable for this experiment because it contains mainly logic. Channel widths of 2, 4 and 8 were chosen as these are the smallest possible channel widths. A channel width greater than 8 would shrink the slot size to less than 8 rows, which results in wide and short slots that would take a very long time to reconfigure with the frame-based reconfiguration approach of the Virtex-4 devices. Table 8.1 lists the devices tested and the number of slots and slot sizes for each device and channel width.

| Device | CLB | Number | CW:2 | CW:4 | CW:8 |
|---|---|---|---|---|---|
| Code | Array | of Slots | Slot Size | Slot Size | Slot Size |
| XC4VLX15 | $64 \times 24$ | 8 | 140 | 96 | 32 |
| XC4VLX25 | $96 \times 28$ | 12 | 168 | 120 | 48 |
| XC4VLX40 | $128 \times 36$ | 16 | 224 | 168 | 80 |
| XC4VLX60 | $128 \times 52$ | 16 | 336 | 264 | 144 |
| XC4VLX80 | $160 \times 56$ | 20 | 364 | 288 | 160 |
| XC4VLX100 | $192 \times 64$ | 24 | 420 | 336 | 192 |
| XC4VLX160 | $192 \times 88$ | 24 | 588 | 480 | 288 |
| XC4VLX200 | $192 \times 116$ | 24 | 784 | 648 | 400 |

**Table 8.1:** Device parameters — CW:channel width and slot size in CLBs

## 8.3 Metrics

To a systems designer, the main aim of architectural exploration is to determine the optimal device size and channel width for a given application. The experimental method used produces two metrics to determine the optimality of a mapping to a particular device — the reconfiguration overhead and the critical path delay.

## 8.4 Reconfiguration Delay Results

The plot in Figure 8.2 shows the total reconfiguration delays for an application consisting of 200 modules with 20% module type variation, with a module size of 60 CLBs (about the size of a DES core). The plot shows the results of mapping the synthesized applications graphs to the family of devices studied using the greedy and dynamic algorithms for merging subgraph sequences representing device configurations with channel widths of 2, 4, and 8 averaged over 120 runs. In this plot the modules have undergone clustering to fit them into the slot sizes.

Note that for channel widths of 2 and 4 a reconfiguration delay of 0 was recorded for the LX200 device. This is because the entire graph was implemented within a single epoch and thus no reconfiguration was necessary. There is a delay involved in loading the initial configuration for the FPGA but this was not added into the reconfiguration delay for three reasons. First, the delay is unavoidable and is not due to *reconfiguration*. Second, it is assumed that sufficient time is available before the application begins in order to load the initial configuration. If more than one application is to be executed in

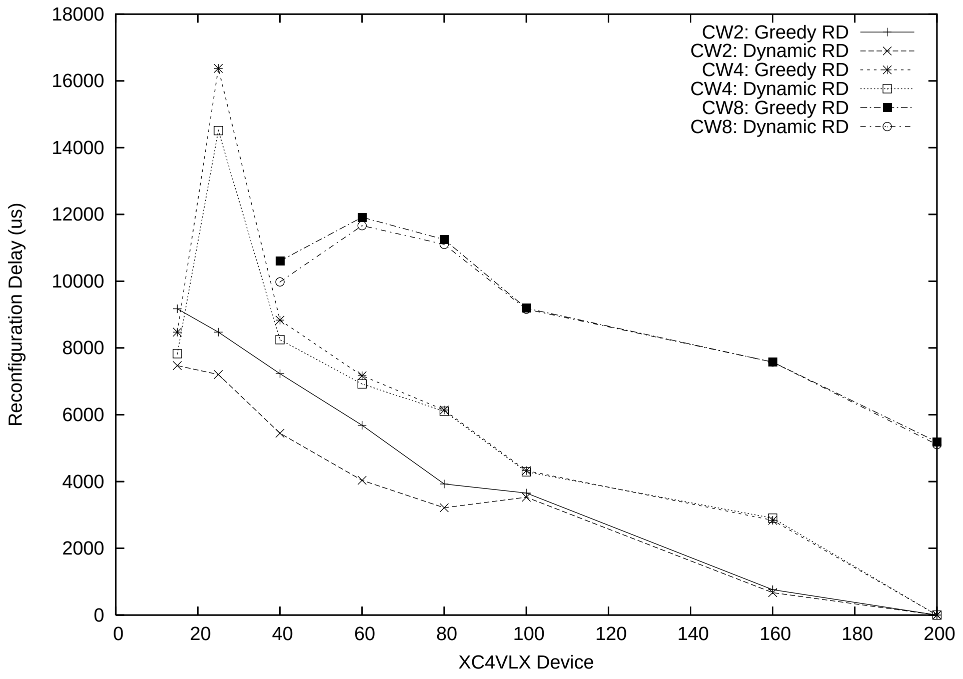


**Figure 8.2:** Reconfiguration delays for 200 modules, 20% type variation, 60 CLB exact module size, clustered, 120 runs

sequence, the subsequent applications should be included in the sequence of graphs. Third, adding this initial configuration delay penalizes larger devices heavily, e.g., the initial configuration delay for the LX15 is 865.92$\mu s$, and for the LX200 it is 12555.84$\mu s$. Adding these delays into the results in Figure 8.2, the LX15 would have an average reconfiguration delay of 8336.88$\mu s$. The LX200 will always have a higher delay than the LX15 and it is the intention of the investigation to show that larger devices can accommodate more logic and thus require less reconfiguration. However, this is an important result to note, and demonstrates the effectiveness of hardware virtualization. Even as the LX15 device needs to be reconfigured at run time, the total run time including the initial configuration delay can be less than that of devices that are large enough to implement the application circuit in its entirety.

### 8.4.1 Device Sizes

Overall, it is apparent from the downward-sloping curves that using larger device sizes results in lower reconfiguration delays. As the device sizes scale in width, and as the channel width remains constant, the slot sizes grow larger. More modules can then be clustered into a single slot, thus reducing the total number of epochs required to map the application. As the device sizes scale in height, the number of slots increase and thus more modules can fit into a single epoch. The number of epochs should also decrease. As the number of epochs decreases the number of periods should also decrease, at least to a maximum of the number of epochs, until it may be possible that the entire application can fit onto the device, e.g., in the case of the LX200 for channel widths of 2 and 4.

Figure 8.2 depicts "anomalies", where using a larger device results in a larger reconfiguration delay, e.g., between the LX15 and LX25 in the case when the channel width is 4 and between the LX40 and LX60 when the channel width is 8.

To explain this, observe that the LX15 device has a slot size of 96 CLBs when the channel width is 4, and the LX25 has a slot size of 120 CLBs, as shown in Table 8.1. The module size used in these graphs is 60 CLBs, thus the LX15 device only manages to fit one module into each slot, whereas the LX25 manages to fit two. As the type variation used here is 20%, the total number of possible module types for the LX15 is 40, as each slot can only support one module. In the case of the LX25, however, two modules are clustered into a single slot, thus the number of apparent module "types" grows to $40^2 = 1600$. This severely reduces the probability of neighbouring epochs having modules that are of the same type. The LX15 therefore has a greater chance of reducing the slot reconfiguration time by allocating modules of the same type in neighbouring epochs in the same slots, whereas the LX25 has fewer opportunities to do so. Note that this anomaly is not present when the channel width is 2, because the LX15 device has a slot size of 120 then, allowing it to fit two modules as well.

In addition, it takes twice as long to reconfigure each slot in the LX25 compared to the LX15 because two modules are clustered into one slot, and the amount of time needed to reconfigure the wiring is about 50% greater because the device is 50% taller.

As the device sizes grow even larger, slots can accommodate many more modules and the number of epochs and periods is substantially decreased. The improvement in reconfiguration delay is less dependent on the variation in module types since the number of possible types grows to a degree where the probability of having the same module type is extremely low. This may be an argument for a better clustering approach that aims to minimize the total number of module types, and/or to maximize the number of modules with the same type in neighboring epochs.

This analysis highlights the benefits of performing architectural exploration with the target application: it is not always the case that using a larger device results in a shorter reconfiguration delay.

To investigate this effect further, an experiment was performed with a normal distribution of module sizes having *mean* module sizes of 35, 60 and 85 CLBs respectively and a fixed standard distribution of 10 CLBs. The results are shown in the plot of Figure 8.3.

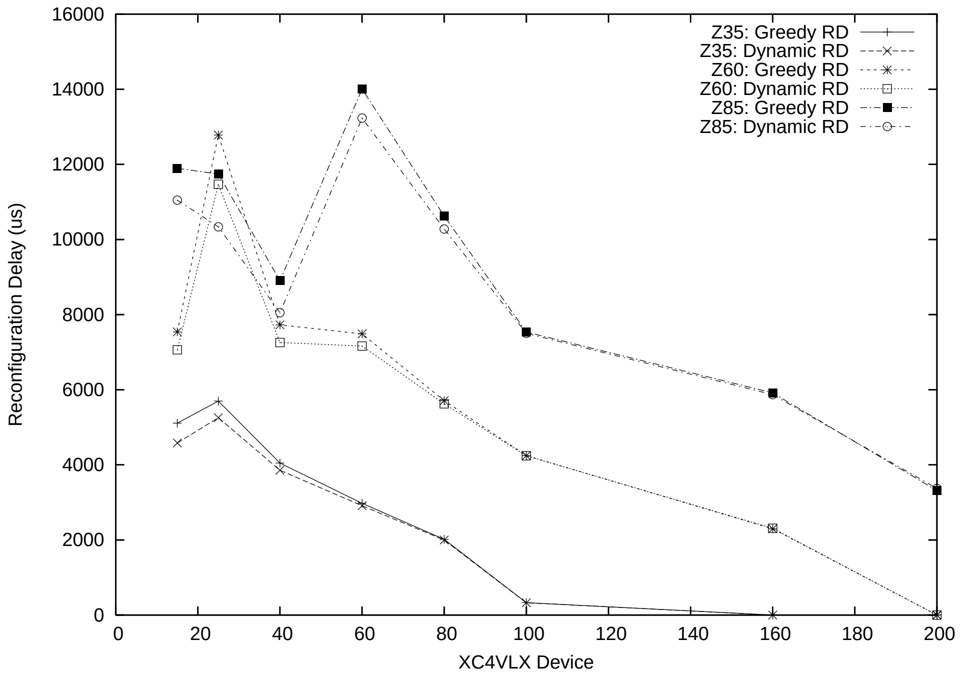


**Figure 8.3:** Comparison between mean module sizes of 35, 60 and 85 CLBs, channel width 4, clustered, 120 runs

When the mean module size is 35, the anomaly between the LX15 and LX25 is not as pronounced as when the module size is 60. This is because there is a much larger *difference* in apparent module type variation between the LX15 and LX25 when the mean module size is 60 (1 vs. 2 modules clustered into a slot) as compared to when the mean module size is 35 (2 vs. 3 modules clustered into a slot), as shown in Table 8.2.

| Device | Average Modules Clustered Per Slot | | |
|---|---|---|---|
| | Mean Size 35 CLBs | Mean Size 60 CLBs | Mean Size 85 CLBs |
| LX15 | 2 | 1 | 1 |
| LX25 | 3 | 2 | 1 |
| LX40 | 4 | 2 | 1 |
| LX60 | 7 | 4 | 3 |

**Table 8.2:** Number of modules clustered per slot at channel width 4

When the module size is 85, the effect is seen between the LX40 and the LX60, where in the Table 8.2 note that the average number of modules clustered into a slot increases from 1 to 3.

It is useful also to investigate how different amounts of variation in the types of modules contribute to this effect. The plots in Figure 8.4 show the results of testing different amounts of type variation. With no type variation, i.e., all the modules are of the same type, there is not much variation in reconfiguration delay across the device sizes and this is expected as the devices are only reconfigured when the wiring harness changes.

As noted before, the number of possible module types due to clustering grows significantly when the type variation increases to 20% from approximately 40 module types to 1600, thus the "anomaly" between the LX15 and LX25 is seen again. As the type variation increases beyond 20% the possibility of having the same module type in neighbouring epochs is even lower and the same pattern is therefore observed.

An important observation in Figure 8.4 is that the same patterns are observed as long as there is any amount of module type variation. This is because the number of apparent module types grows exponentially as more modules can be clustered into a slot. This implies that module type variation should not cause significant differences in these experiments.

### 8.4.2 Channel Widths

It is apparent from the plots in Figure 8.2 that the reconfiguration delay increases as the channel width increases. This is to be expected since the

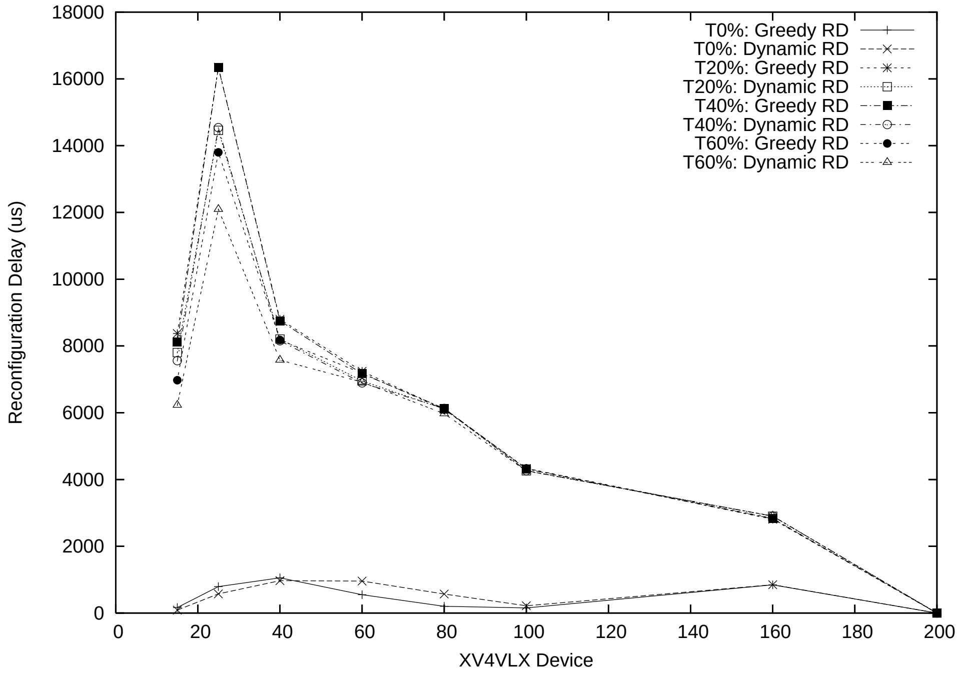


**Figure 8.4:** Comparison between different type variations (0%, 20%, 40% and 60%), channel width 4, clustered, 120 runs, 60 CLB fixed module size

number of epochs for a particular application increase as the channel width increases. This is because the slot sizes decrease as more logic area is allocated for the wiring channel. This also implies that the time to reconfigure the wiring between periods also increases.

The system designer can thus try different channel widths in decreasing order to find the smallest channel width that can accommodate the application.

### 8.4.3 Comparison between the Greedy and Dynamic Methods

The results indicate an improvement by as much as 29% (in the LX60 device with a channel width of 2) using the dynamic programming approach over the greedy method.

As can be seen in Figure 8.2, this advantage decreases as the device size increases because there are far fewer possibilities to better group subsequences as the number of epochs needed to implement the application decreases.

As noted in Chapter 7, the greedy method can allocate modules of the same type in the same location, not only in the next epoch but also between periods, while the dynamic programming approach can only do so between epochs. However, this advantage only appears to make a difference when there is little variation in module type, as can be seen in Figure 8.4 for the LX60 and LX80 when there is 0% type variation. There is no slot reconfiguration delay when the type variation is 0%, which causes the dynamic algorithm to make poorer choices because it assumes that the entire device is reconfigured at the start of a period.

The amount of improvement between the greedy and dynamic programming algorithms largely depends on the application, thus an average difference over all the test runs may not be as significant as comparing the results for individual applications. The plot in Figure 8.5 therefore shows the percentage reduction in reconfiguration delay against the percentage of all the test runs that achieved that reduction. This graph has been derived from the results plotted in Figure 8.2. From this summary plot it can be seen that

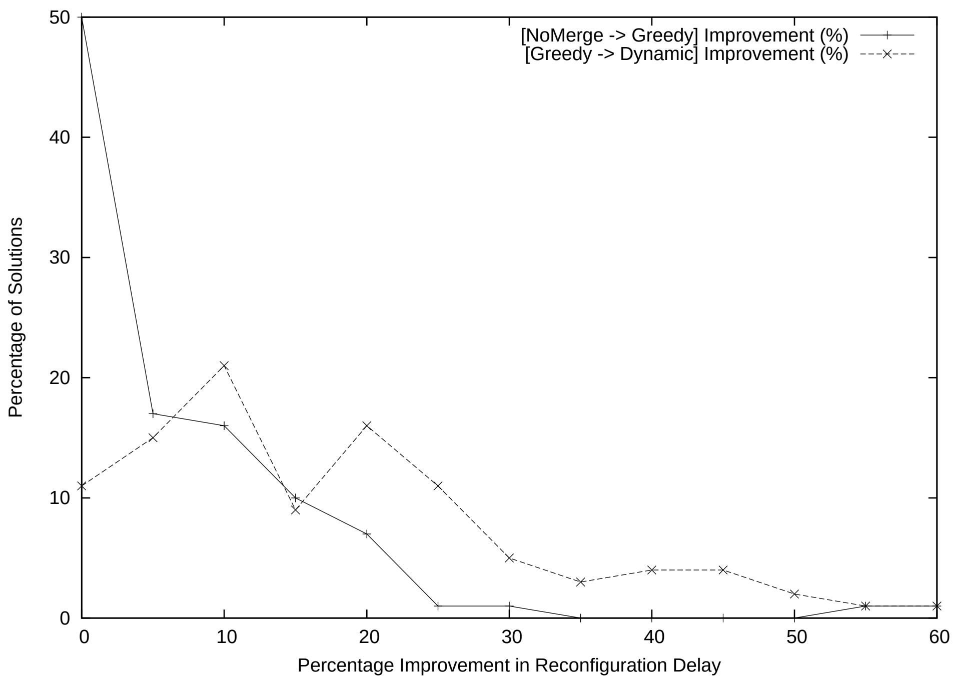


**Figure 8.5:** Percentages of reconfiguration delay reduction for graphs with 200 modules, 20% type variation, 60 CLB exact module size, clustered, 2880 runs

performing graph merging with the greedy method results in improvements in reconfiguration delay of up to 60% for half of the total number of solutions.

Using the dynamic programming algorithm provides further improvements over the greedy method. Only 11% of the test runs did not have a reduction in reconfiguration delay, and this is over and above the reduction achieved by the greedy method.

### 8.4.4 Disabling Clustering

The aim of clustering modules to fit slot sizes (see Section 3.7 and Section 8.2.3) is to reduce the number of total configurations to implement the entire application by maximizing the utilization of the available slot area.

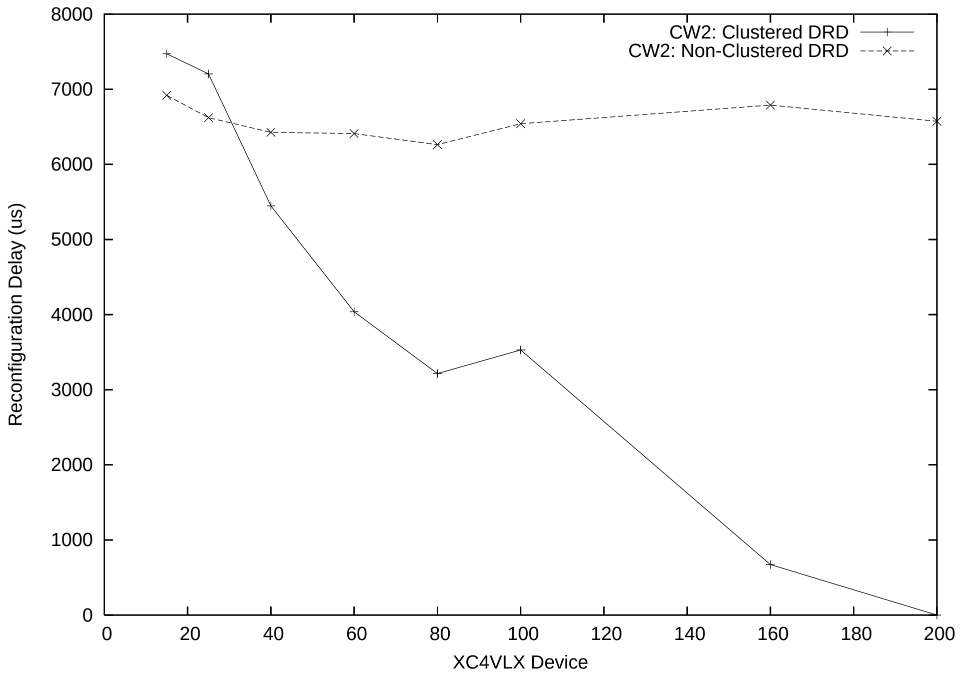


**Figure 8.6:** Comparison between clustered and non-clustered reconfiguration delays, channel width 2, 200 modules, 20% type variation, 60 CLB exact module size, dynamic algorithm, 120 runs

The desired effect of this is to reduce the total reconfiguration delay as less reconfigurations are necessary.

However, a side effect of clustering is that the total number of apparent module types increases. This is because a clustered module is of a different type to that of any of its constituents. Thus, the total number of possible module types is exponentially increased from $n$ to $n^m$ where $m$ is the number of modules that can be clustered into a single slot. This therefore reduces the opportunities for judicious allocation of modules to slots so as to reduce overheads due to module reconfiguration.

The plots in Figures 8.6, 8.7 and 8.8 show comparisons between clustered and non-clustered reconfiguration delay estimates for channel widths of 2, 4

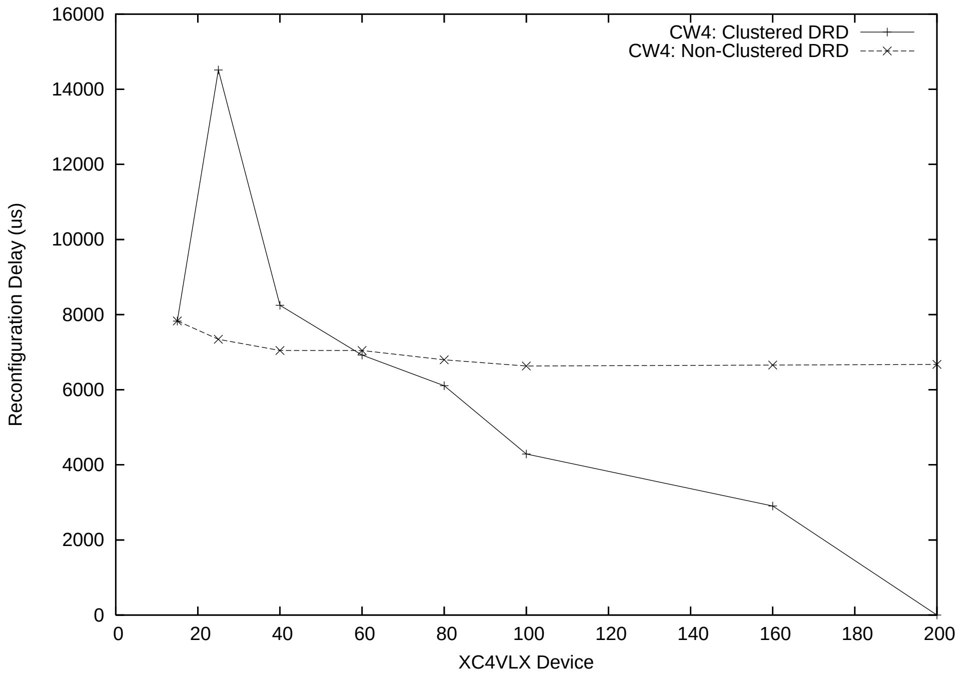


**Figure 8.7:** Comparison between clustered and non-clustered reconfiguration delays, channel width 4, 200 modules, 20% type variation, 60 CLB exact module size, dynamic algorithm, 120 runs

and 8 CLB rows or columns respectively. The reconfiguration delays for the non-clustered runs show little variation in reconfiguration overhead because the number of epochs decreases at the same rate as the device size grows, i.e., at the same rate as the number of available slots increases. On the other hand, clustering leads to a reduction in reconfiguration delays for larger devices since the total number of epochs is dramatically reduced.

It is interesting to note, howwever, that there are regions in each graph where not clustering the modules is more advantageous than clustering them. With a channel width of 2 (Figure 8.6), this region is for device sizes equivalent to the LX25 in size and below. The LX15 and LX25 both cluster two

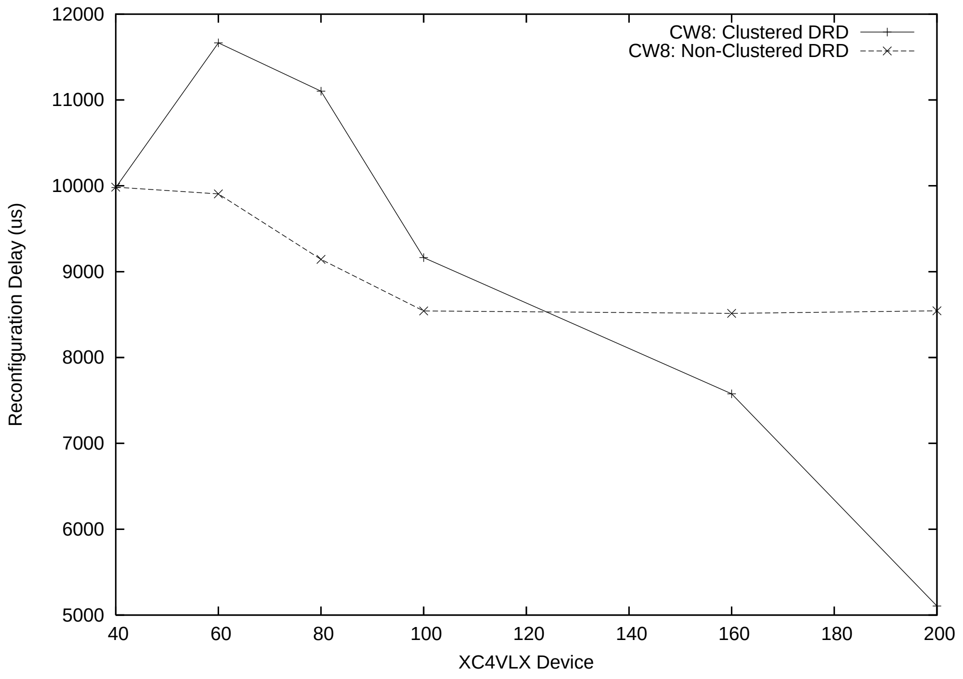


**Figure 8.8:** Comparison between clustered and non-clustered reconfiguration delays, channel width 8, 200 modules, 20% type variation, 60 CLB exact module size, dynamic algorithm, 120 runs

modules into a slot, incurring the penalty of higher apparent type variation as previously explained.

Table 8.3 shows the number of periods generated for a particular synthetic application graph mapped onto the LX25, the LX40 and the LX60 devices with a channel width of 2. The LX25 (for which, from the plot it can be seen, that the non-clustered case is better) requires 5 periods for the clustered case and 4 in the non-clustered case. The LX40 has the same number of periods for both cases, but it must be noted that there are more epochs (15 vs. 5, as shown in parentheses in the table). This observation follows on with the LX60, where the number of periods for the non-clustered case is greater than that for the clustered case. The LX60 has the same height as the LX40, thus it has the same number of slots and the same number of epochs in the non-clustered case.

| | LX25 | LX40 | LX60 |
|---|---|---|---|
| Clustered | 5 (10) | 4 (5) | 3 (3) |
| Non-Clustered | 4 (20) | 4 (15) | 4 (15) |

**Table 8.3:** Number of periods and epochs (in parentheses) for application graph 211002 (channel width 2)

This cut-off point where not clustering is more advantageous shifts to the right, i.e., towards larger devices, as the channel width increases. When the channel width is 4 (see Figure 8.7), this point shifts to the LX60. It is also noteworthy that the LX15 has the same results for both the clustered and non-clustered cases because no clustering is possible for the LX15 at a channel width of 4. When the channel width is 8 (see Figure 8.8), the cut-off point shifts further up the device size scale to the LX100. Again, the LX40 has the same results for both the clustered and non-clustered cases because the LX40 has a slot size of 80 (see Table 8.1) with a channel width of 8, thus no clustering is possible. This is to be expected because the sizes of the slots diminishes as the channel width increases, allowing less modules to be clustered into each slot.

In real applications, a better clustering technique might minimize the penalty incurred at small device sizes. However, the number of periods is still larger in the clustered case, and it is still to be expected that not clustering will produce a better result. It should also be noted that the amount of difference will depend upon the degree of module variation present in the application.

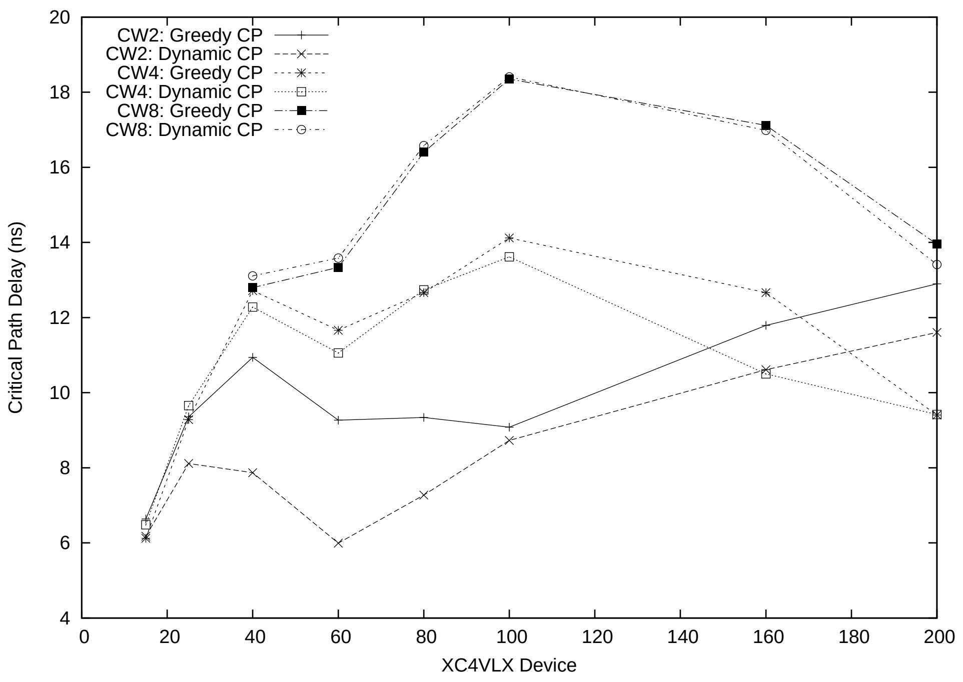


**Figure 8.9:** Contribution to the maximum critical path delays of the wiring infrastructure for 200 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

## 8.5 Critical Path Delay

The next set of results that a system designer needs to consider are the estimated critical path delays. Figure 8.9 shows the estimated contribution of the generated wiring harnesses to critical path delays, i.e., the chart plots the maximum critical path delay among all the periods in an application, averaged among all the applications. These results were obtained by mapping the previously synthesized graphs to the LX device family with architectural parameters varied as before.

### 8.5.1 Device Sizes and Channel Widths

When the COMMA methodology is employed it is to be expected that as the device size increases the critical path delay also increases because the number of module slots increases and the distance between those that are furthest apart grows. It is also to be expected that for smaller devices the delays are higher with larger channel widths because opportunities for clustering are diminished, and thus the wiring harness suffers more congestion as more subgraphs are merged per period. Diminishing critical path delays for larger channel widths on large devices (LX100 and above) illustrate the benefit of having sufficient channel capacity to satisfy the wiring needs of large subgraphs.

A "dip" in the plots can be seen in the LX60, which is particularly pronounced in the case of the smaller channel widths 2 and 4. Examining the results for a particular run for the three devices centered on the LX60 at a channel width of 2 yields the subsequence groupings shown in Table 8.4.

| Device | Grouping | Epochs/Periods |
|---|---|---|
| LX40 | 1 2 \| 3 \| 4 5 | (5 epochs, 3 periods) |
| LX60 | 1 \| 2 \| 3 | (3 epochs, 3 periods) |
| LX80 | 1 2 | (2 epochs, 1 period) |

**Table 8.4:** Subsequence groupings for application graph 221003 (channel width 2)

As the device size grows, it is expected that the number of epochs will decrease, and so will the number of periods. The critical path delay, however, increases as the wiring channels become increasingly congested. Detailed routers initially use long wires (e.g., hexes and longs) to minimize the number of switch box hops. As the channel utilization increases, the router is forced to use shorter wires (e.g., singles and doubles), thus requiring more hops, which results in a higher critical path delay. This means that sparse wiring harnesses should generally have a lower critical path delay, as can be seen in the subsequence groupings in Table 8.4, where the LX60 packs just a single epoch per period.

### 8.5.2 Comparison between the Greedy and Dynamic Methods

It can be seen in Figure 8.9 that the dynamic programming approach produces favorable critical path delay results despite targeting reconfiguration delay. This may be due to the greedy algorithm *always* making the most complex wiring harness possible, whilst the dynamic programming algorithm tries to minimizes the reconfiguration delays, which may not result in wiring harnesses that are as congested.

### 8.5.3 Disabling Clustering

The critical path delays for the clustered and non-clustered cases have also been compared in Figures 8.10, 8.11 and 8.12.

The plots clearly show that clustering results in shorter critical path delays. The reason for this is because the wiring harnesses in each *period* are likely to be more dense in the non-clustered case than in the clustered case. The wiring harnesses in each *period* in the non-clustered case are complex because the intermodule connections in each *epoch* are **less** dense than in the

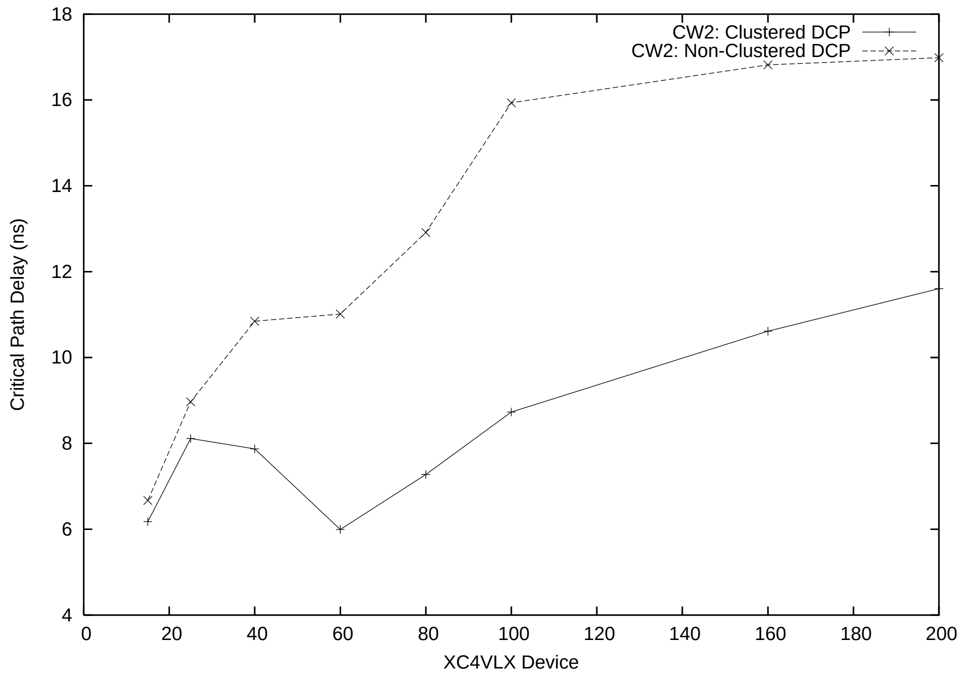


**Figure 8.10:** Comparison between clustered and non-clustered contributions to the maximum critical path delay of the wiring harness (channel width 2, 200 modules, 20% type variation, dynamic programming algorithm)

clustered case. Thus more epochs can be merged into a single period if the individual epochs have sparse intermodule connections.

For example, if providing the intermodule wiring for each epoch takes up about 30% of the wiring channel area in the non-clustered case, three epochs with dissimilar intermodule connections can be merged into a single period, thus the wiring harness for such a period takes up a total of 90% of the wiring area. However, if the intermodule wiring for a single epoch takes up about 60% of the wiring channel area in the clustered case, then a period can only consist of a single epoch, thus the wiring harness takes up 60% of the area.

This can also be seen in Table 8.3 where the average number of epochs in a period when clustered is much less than when it is not clustered. E.g., for

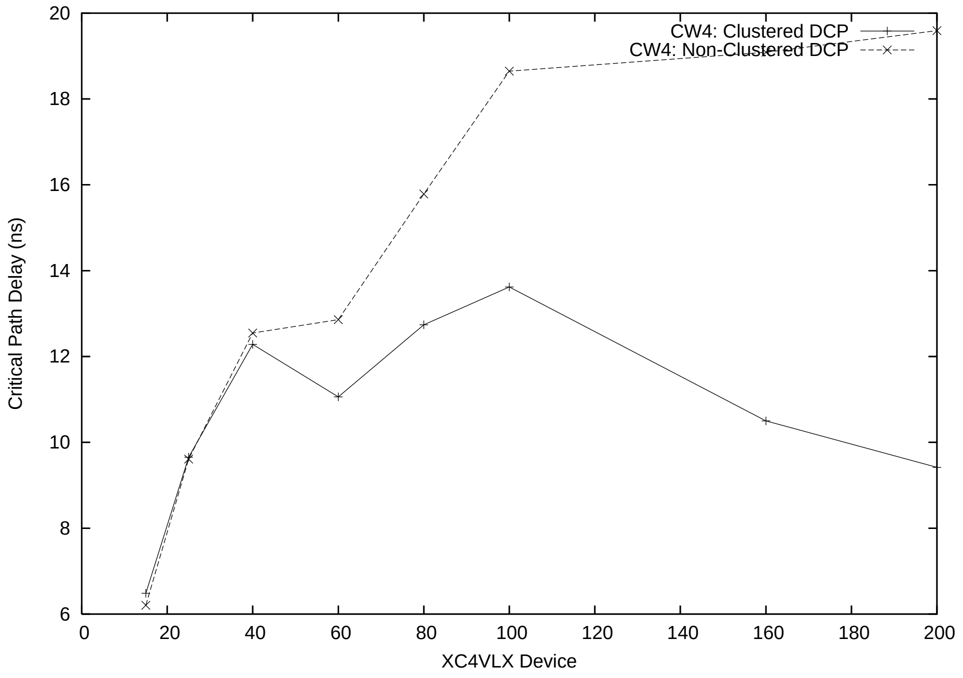


**Figure 8.11:** Comparison between clustered and non-clustered contributions to the maximum critical path delay of the wiring harness (channel width 4, 200 modules, 20% type variation, dynamic programming algorithm)

the LX25 device there was an average of 2 epochs per period when clustered, but 5 epochs per period when not clustered.

## 8.6 Conclusions

It can be seen from the results that it is generally the case that using larger devices reduces the reconfiguration time. However, in certain cases it was shown that a smaller device yields a better result because the mean module size and type variation of the modules limited the possibility of allocating modules in neighboring epochs with the same type to the same slot.

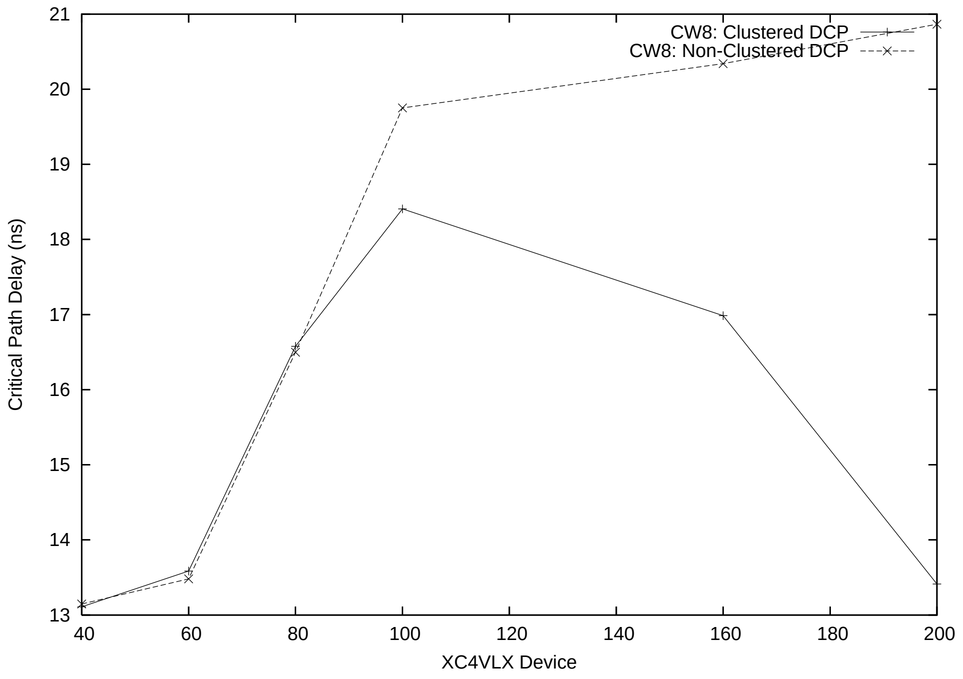


**Figure 8.12:** Comparison between clustered and non-clustered contributions to the maximum critical path delay of the wiring harness (channel width 8, 200 modules, 20% type variation, dynamic programming algorithm)

The system designer should use the smallest channel width that can accommodate the application, since both the reconfiguration and critical path delays increase as the channel width increases. The success rates for mapping the applications with a module size of 60 to different channel widths in these experiments is shown in Table 8.5. It should be noted that there is a large proportion of applications that cannot be mapped at the channel width of 2, which should generally be because there is insufficient wiring area to accommodate the communications. At the other end of the spectrum, as the smallest device that can accommodate a channel width of 8 is the LX40, this option may not be feasible if the budget for purchasing devices is limited to the LX15 or LX25. At the same time, because the slot sizes shrink as the channel width increases, the slots may no longer accommodate the module sizes in the application.

| | CW:2 | CW:4 | CW:8 |
|---|---|---|---|
| Clustered | 593/960 (61.8%) | 956/960 (99.6%) | 718/720 (99.7%) |
| Non-Clustered | 679/960 (70.7%) | 958/960 (99.8%) | 718/720 (99.7%) |

**Table 8.5:** Success rates for different channel widths for the full range of LX device sizes, 200 modules, 20% type variation, 60 CLB exact module size, 120 runs

Performing module clustering results in significantly lower reconfiguration and critical path delays than when it is not performed, and the advantage increases as the device size increases. However, it was shown that for small device sizes there is a possibility that disabling module clustering results in smaller reconfiguration delays.

In comparing the greedy method and the dynamic programming approach, the dynamic programming approach produces more favorable results as it considers many more possibilities for subsequence grouping.

Overall, it has been shown that the COMMA methodology is able to allow a system designer to perform architectural exploration and obtain the results for mapping a target application onto different devices. The system designer can then make a more informed choice based on the budget and constraints that have been imposed, and choose the device and channel width that is shown to have the best results.

Note: The plots for tests performed using 40, 80, 120 and 160 modules are included in Appendix A. It can be seen from the plots that similar patterns emerge, with the main difference being that with less modules, smaller device sizes are needed to obtain the results reported in this chapter, i.e., the plots shift to the left.

# Chapter 9

# Tool Support

## 9.1 Introduction

This chapter describes the design implementation stages of the COMMA design flow. Implementation commences with a sequence of merged subgraphs and produces a set of configuration bitstreams to be loaded onto the FPGA.

The COMMA design flow integrates seamlessly with the Early Access Partial Reconfiguration Tool Flow [95] and augments the EAPR toolset at every step. The need for low-level circuit implementation and bitstream modification tools is eliminated, and the risk of implementation errors is reduced.

This chapter first presents a brief introduction to the EAPR tool flow and its steps to implement partially-reconfigurable applications. The COMMA implementation flow is then described in the sequence of steps followed by the EAPR tool flow.

## 9.2 The EAPR Tool Flow

The steps in the EAPR tool flow are depicted in Figure 9.1.

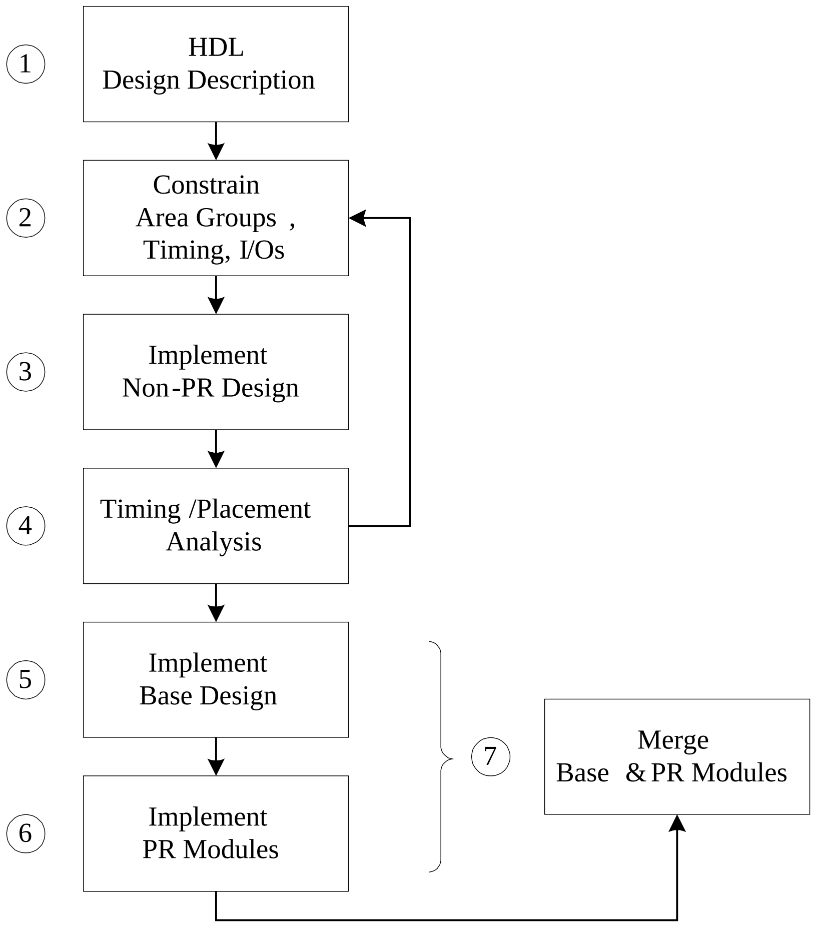


**Figure 9.1:** Steps in the EAPR tool flow, reproduced from [95]

Step 1 is similar to that of a traditional, static design flow where the design is described in HDL ("soft" cores) or Relatively-Placed-Macros (RPMs), i.e., placed-and-routed ("firm") cores. In addition to describing the logic, it is also necessary at this stage to be mindful of how partially-reconfigurable modules are to communicate with the base design and each other. Thus all the modules that are to be placed in the same area, or are to be swapped with one another, need to present the *same interface*. This interface is to connect to *slice macros* that have been placed in fixed positions on the FPGA. These slice macros connect the modules to the base design or to each other. Although the actual locations and sizes of the areas are not yet defined in this step, the designer must already be mindful of the number of required partially-reconfigurable areas and the set of modules that is to be placed in them. The final task in this step is to synthesize the HDL into device-independent netlists.

Step 2 involves defining regions on the FPGA where partially-reconfigurable modules can be placed using area group and location constraints. These regions are known as Partially Reconfigurable Regions (PRRs), and modules that can be placed into them are known as Partially Reconfigurable Modules (PRMs). The FPGA area that is not part of a PRR is collectively known as the **base region**, where static modules, if any, are placed, as shown in Figure 9.2. Intermodule wiring also crosses the base region to other PRRs or static modules. In this step, each slice macro is also assigned a fixed location on the edges of the PRRs.

PRMs can be placed into PRRs as shown in Figure 9.2. It is important to note the relationship between a PRR and PRM. A partial module bitstream must be generated for each module type that is to be placed into a PRR. Thus a PRM differs from a normal VHDL entity or Verilog module in that it is specifically implemented for a particular PRR. This essentially implies that every PRM that is to be placed into a PRR must have the same entity or module name and present the same interface.

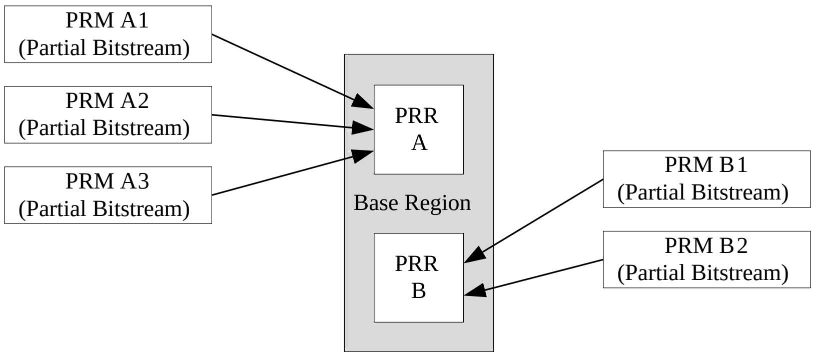


**Figure 9.2:** Partially reconfigurable regions (PRRs), partially reconfigurable modules (PRMs) and the base region

Steps 3 and 4 are used to verify that area and timing constraints are met, and may be skipped according to the designer's wishes. These steps are analogous to the timing verification stages in a standard static FPGA design flow.

In Step 3 a non-partially-reconfigurable design is implemented, i.e., it is fully placed and routed. This circuit design involves placing any one of the possible modules into the PRRs and is meant to represent a possible configuration during the lifetime of the application.

In Step 4, the timing and placement reports from the place-and-route tools are analyzed to ensure that they meet constraints. Note that, as in the standard static FPGA implementation flow, an iteration occurs from Step 4 to Step 2 in case any constraints are not met. In the EAPR tool flow this iteration can also serve another purpose — to test different configurations of the application. Implementing only one possible configuration in Step 3 may be insufficient if there is a good deal of variation in the PR modules, so multiple verification steps may be necessary to fully ensure that constraints are met.

In Step 5 the base design that consists of static modules (if any) and the connections between the slice macros, the static modules, and I/Os are placed-and-routed. This differs from Step 3 in that in this step the PRRs are empty, whereas in Step 3 each PRR must have one PRM placed into it. The output of this stage is a placed-and-routed netlist without any partially-reconfigurable modules in the PRRs. An important issue to note here is that the EAPR tool flow, in contrast to XAPP290, allows routes to cross through PRRs in order to maximize the probability that timing constraints are met. An *exclusion list* of routes is produced at this stage that is used in the next step to prevent PRMs from using any wires that pass through their PRRs. The ISE 8.2i version of the EAPR toolset supports a constraint to prevent routes from passing through module areas. It is not an intention of the COMMA methodology to allow routes to pass through module areas, but this does not pose any problems as long as timing and area constraints are met. In the ISE 9.1i version of the EAPR toolset the option to disable routes passing through PRRs is removed, and it is unclear as to whether it will be reinstated in the future. Although it is the original intention of the COMMA methodology to route around module areas, it is not necessary and the methodology will still hold in case the system designer decides to use the ISE 9.1i EAPR toolset, although care must be taken to prevent area constraints from being exceeded if these routes take up too much area within the PRRs. The EAPR toolset does not support relocation of modules thus a separate circuit and bitstream must still be generated for each PRM in each PRR. One advantage of using the new toolset is that slice macros can take up only a single slice rather than two, cutting area overhead by half and eliminating the need to define pin directions at design time. At the time of writing of this thesis single slice macros have been announced but are not yet available.

In Step 6, each Partially-Reconfigurable Module (PRM) is individually placed-and-routed. The *exclusion list* of routes produced in Step 5 is used at this stage to prevent the modules from using any of the wires that are already allocated for the base design and/or intermodule communication. The output of this stage is a collection of *partial* placed-and-routed netlists, each one implementing one PRM in the design.

In Step 7, the base netlist from Step 5 and the module netlists from Step 6 are merged together to produce two sets of bitstreams. The first is a set of *full configuration bitstreams*, one for each necessary possible starting configuration of modules. A full bitstream consists of the base design and one PRM in each PRR. This is used as a startup configuration, and one is produced for as many possible incarnations of module allocations at application startup. The second set consists of *partial module bitstreams*. One logic bitstream and one blanking bitstream is produced for each PRM that is to be placed in a PRR. Thus the total number of partial module bitstreams for *each PRR* is the number of modules that can be placed into that PRR. A blanking bitstream is also produced for each PRM at each PRR to completely wipe the PRR of module logic. However, a full reconfiguration *of the PRR* occurs whenever a module is swapped, not a difference configuration, so the blanking bitstream is only provided for when it is truly necessary to remove the module. An example of this may be if outputs need to be disconnected to prevent data from being corrupted. Swapping modules does not require loading the blanking bitstream first.

## 9.3 The COMMA Implementation Flow

The COMMA implementation flow is depicted in Figure 9.3. The inputs to this phase are the set of periods derived from the graph merging process, the module HDL and the Chip and Communications Configuration (CCC) file. The final outputs consist of one initial configuration bitstream per period, and a set of module bitstreams for each epoch. The brackets on the left indicate the specific steps of the EAPR tool flow that the stages of the COMMA implementation flow apply to.

The rest of this chapter describes the COMMA implementation in each EAPR step, while detailing the individual processes in each step.

> **Note regarding the graphics in the following subsections:** Portions of Figure 9.3 are reproduced in the following sections detailing the COMMA implementation for each of the steps of the EAPR tool flow. As a partial diagram for a particular step may include portions of other steps, a box with a dashed outline is used to highlight the parts that are relevant to the discussion.

### 9.3.1 EAPR Step 1: Design Description

In this step in the standard EAPR tool flow a systems designer decides how many Partially-Reconfigurable Regions (PRRs) are necessary, plans the

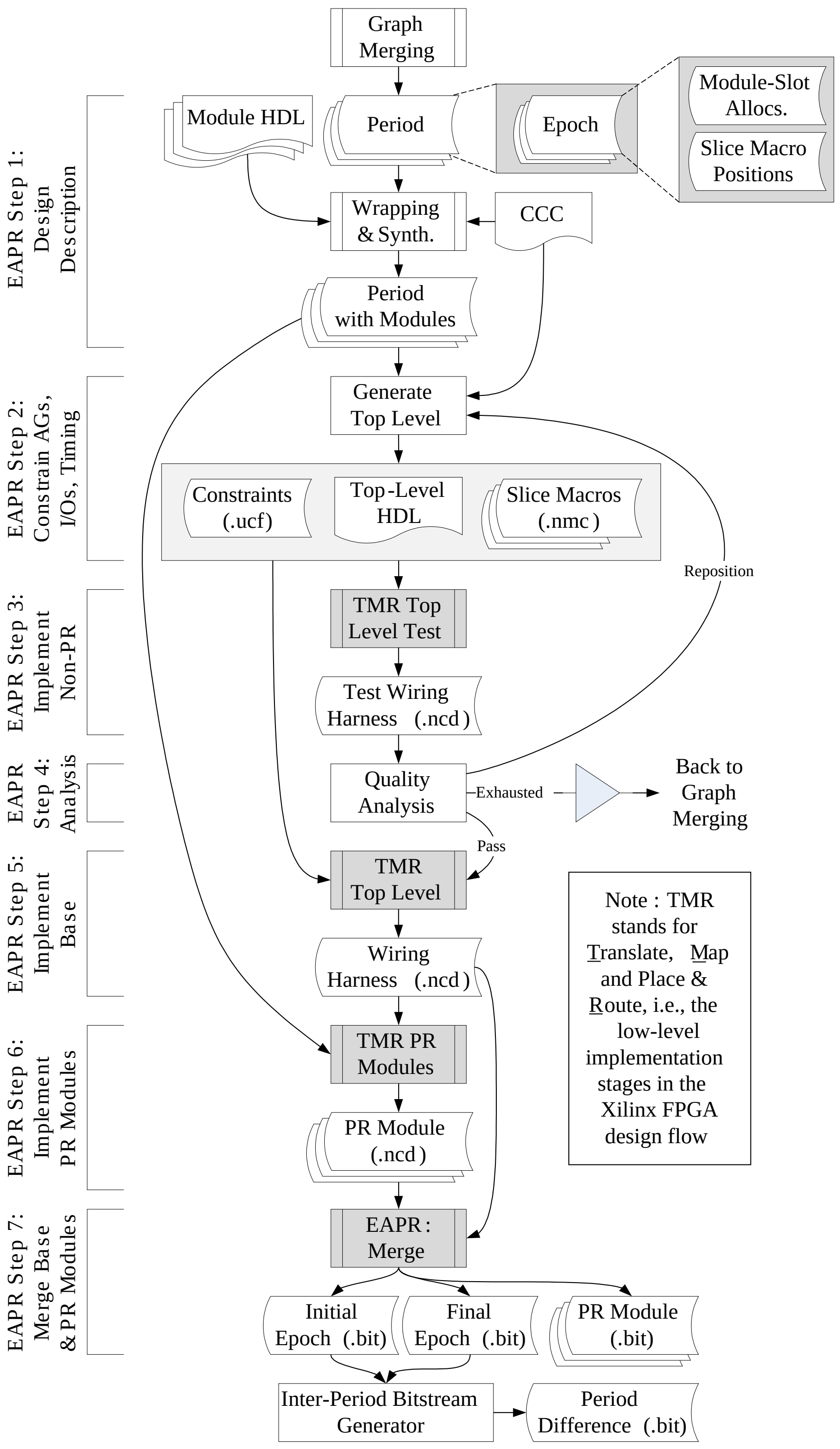


**Figure 9.3:** Implementation flow

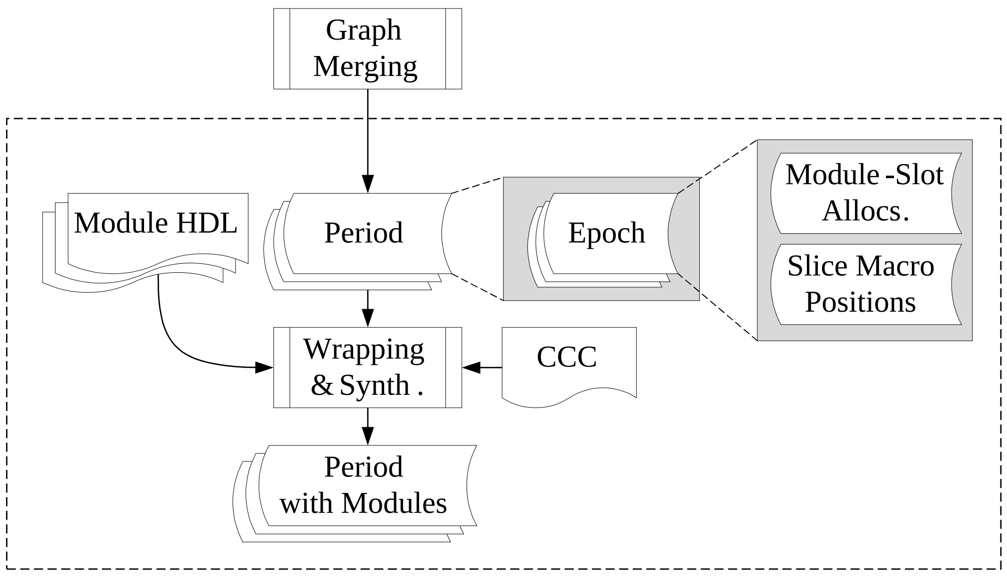


**Figure 9.4:** COMMA implementation of step 1 in the EAPR tool flow

allocation of Partially-Reconfigurable Modules (PRMs) to PRRs, and maps the module interfaces to slice macro interfaces.

The inputs to the COMMA implementation of this step include the periods from the graph merging phase, the module HDL, and the Chip and Communications Configuration file. In turn, these are input into the "Wrapping and Synthesis" process, which generates HDL wrappers for each module to connect its interface to the slice macros and synthesizes each module to a device-independent netlist. These synthesized modules are stored together with the periods to which they belong, forming the output of this step.

### Periods and Epochs

Each period from the graph merging phase consists of a set of epochs, as shown in Figure 9.4. Each epoch in turn consists of the module-slot allocation for that epoch, i.e., a list of the modules in that epoch and their slot allocations, and the slice macro positions for each bit of I/O communication for each module. The modules are identified by the VHDL entity or Verilog module name.

The module HDL should be provided as standard VHDL entity-architecture descriptions or Verilog modules. The COMMA implementation maps the ports to slice macros automatically in the "Module Wrapping" process.

### Chip and Communications Configuration

The Chip and Communications Configuration file is an output from the "Configurator" process described in Section 3.7.1. The "Configurator" is a program that sources device and package information from the package and device pinout files provided by Xilinx and the package reports generated by the PartGen tool in ISE specifying IOB details. The "Configurator" also serves as a means for the system designer to specify the COMMA device layout, select IOBs, and set other implementation-level options. The "Configurator" obtains all this information and packages it into a "Chip and Communications Configuration" (CCC) file that is used by various tools in the COMMA design flow.

The CCC file includes the following information:

- **Device information:** The device code, size, package and timing information.
- **Layout information:** The exact dimensions and locations of every slot. Note that the higher-level "channel width" parameter is not used

here as the lower-level implementation caters for any slot layout, not just the one used in the algorithms and experiments in this thesis.

- **Logical pad map:** Each IOB used is assigned a logical name so as to abstract away the actual low-level IOB name and location from the module implementation. This also allows the same application to be mapped to different devices and packages by modifying the logical pad map.

- **Slot/macro map:** If necessary the system designer can allocate predefined locations for slice macros on the slot boundaries. This mapping is optional but can be used to fine-tune the timing results.

### Wrapping and Synthesis

Each period is input into the wrapping and synthesis process, which performs the following functions, in sequence:

1. **Generate slice macros:** Each period has its own set of slice macros connecting the PRRs to the wiring harness. This subprocess analyzes all the slice macro positions from every epoch and generates the Xilinx Description Language (XDL) code for them, with the appropriate number of bits and their respective directions, e.g., left-to-right, top-to-bottom, etc. XDL is a language that describes detailed circuits (logic and wiring) mapped onto an FPGA and is specified in a text-based format. The XDL files are then converted to hard macros (NMC files) and stored together with the period. This subprocess is not run if the ISE 9.1i version of the EAPR toolset is used with single-slice macros, since

there is no requirement to predefine the I/O directions with single-slice macros. However, note that single-slice macros are still not available at the time of writing of this thesis, thus this subprocess can be turned off in the future.

2. **Allocate slice macros:** This subprocess analyzes the slice macro positions from the epochs again and produces a list of slice macro instantiation locations for each of the PRRs.

3. **Generate slot interfaces:** Now that all the slice macros have been generated and allocated, each PRR has its own set of slice macros allocated along its boundaries. An interface HDL file consisting of a VHDL entity or Verilog module and its instantiation template is created for each slot in each period. Each slot will have its own slot interface that remains unchanged for all epochs within a period.

4. **Generate module wrappers:** Each module is then wrapped with an RDP wrapper (see Section 3.6.2) that maps each bit of data at each port to a slice macro input or output. This is done by obtaining the HDL slot interface declaration and creating an HDL file that encapsulates the original module. Each of the ports on the original module are mapped to the slot interface. Since every module that is to be placed in a PRR must have the same name, this subprocess also abstracts the module renaming away from the system designer. As only point-to-point connections are supported at present, an RDP wrapper simply maps the original module to the slot interface. A wrapper is usually weightless, i.e., it contains no logic, unless more than one module *output*

bit is mapped to the same slice macro, or one module *input* bit is mapped to more than one slice macro. The subgraph mapping stage described in Section 6.6.4 tries to assign unique slice macro bits to each port if possible. Point-to-multipoint/multipoint-to-point/multipoint-to-multipoint connections may be supported in the future, which would entail including multiplexers in these RDP wrappers.

5. **Synthesis:** The wrapped module HDL is finally synthesized with a chosen synthesis tool into device-independent netlists (XST NGC or EDIF files) and are stored together with the periods to which they belong.

The output of this subprocess, and effectively that of Step 1 of the COMMA implementation of the EAPR tool flow, is the set of periods that were input into this step combined with the module netlists, slice macros and physical slice macro allocations.

### 9.3.2 EAPR Step 2: Constrain Area Groups, I/Os and Timing

In this step, the periods with module netlists and the CCC file are analyzed and a set of top-level implementation files are generated. As shown in Figure 9.6, each slot is implemented as a PRR with the wiring harness being the “base design”. This “base design” contains one static module, known as the Reconfiguration Control Module (RCM) that links the off-chip bitstream storage memory to the on-chip Internal Configuration Access Port (ICAP) controller. See Section 9.4 for more information on this module.

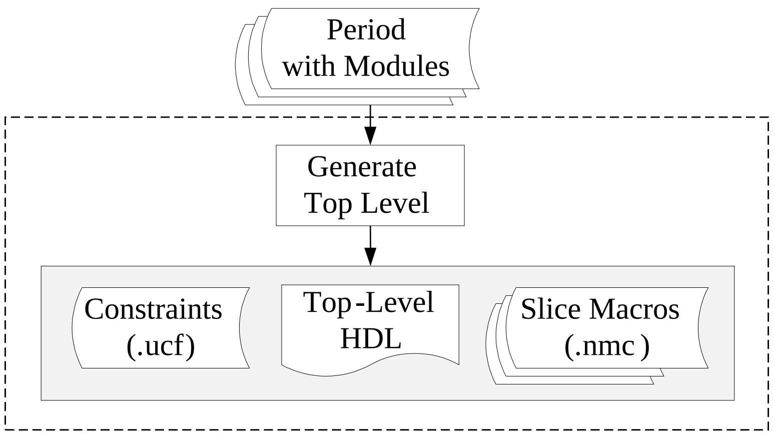


**Figure 9.5:** COMMA implementation of Step 2 in the EAPR tool flow

A top-level HDL file is created with all the slots and slice macros instantiated. Accordingly, a User Constraints File (UCF) is also generated specifying the dimensions and locations of each slot, and the locations of the slice macros. The timing constraints (TIMESPEC) for each period are also added at this point to provide a goal for the wiring harness to meet. As shown in Figure 9.5, this step simply produces HDL and constraints from the specifications in the CCC file and the set of periods and netlists obtained from Step 1. The NMC files are then copied from the set of periods and packaged into the base design.

## 9.3.3 EAPR Step 3: Implement Non-PR Design

This step completely implements one of the epochs in each period to generate a "test wiring harness". This is a static implementation of an epoch and is not partially-reconfigurable. In Figure 9.7 the gray process marked "TMR Top Level Test" indicates that the "***T***ranslate", "***M***ap" and "Place and

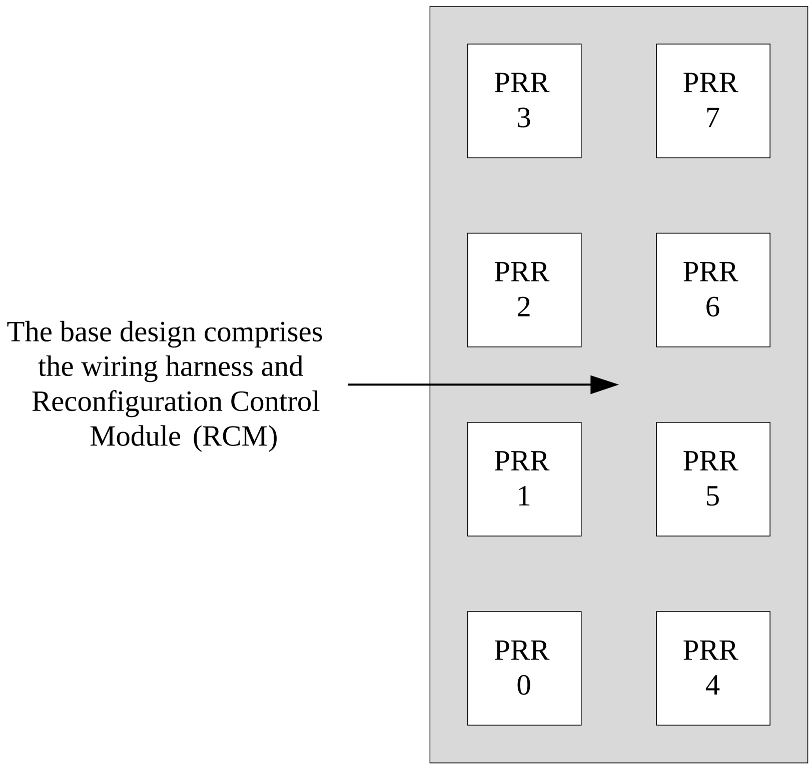

PRR 3
PRR 7
PRR 2
PRR 6
The base design comprises the wiring harness and Reconfiguration Control Module (RCM)
PRR 1
PRR 5
PRR 0
PRR 4

**Figure 9.6:** Base design and PRR layout

***R***oute" tools in the EAPR toolset are used. These three tools perform the final implementation of an FPGA design before a bitstream is generated.

When the top level HDL is implemented, the connections between the slice macros and other slice macros and IOBs is routed. The resulting wiring is representative of how the wiring harness will be implemented for partial reconfiguration in Step 6 since the connections between the slice macros and IOBs are the same for every epoch in a period. This "test wiring harness" is checked to see if it meets the area and timing constraints in Step 4, providing a final verification of implementation feasibility over and above the estimations performed during subgraph mapping (described in Section 6.6.4). Although the area and timing estimations in subgraph mapping are overes-

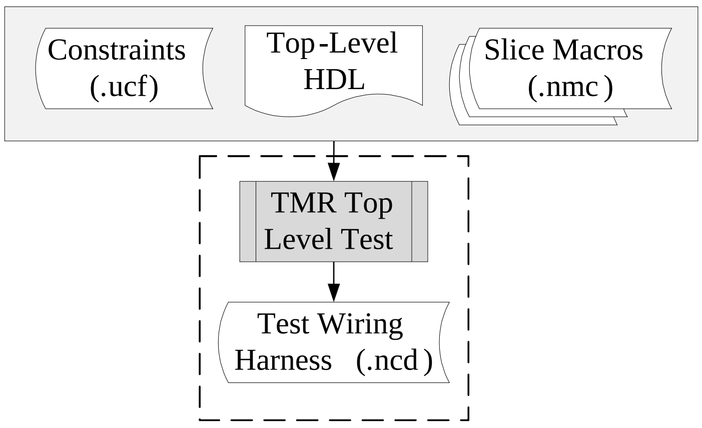


**Figure 9.7:** COMMA implementation of step 3 in the EAPR tool flow

timates, it is still safest to perform these final verification steps and possibly return to the graph merging phase if the constraints are not met due to unexpected complications. The quality analysis is detailed in the next section.

Only one of the epochs per period needs to be implemented as the wiring harness is the same for all epochs in a period. However, the system designer may opt to implement each epoch to check that all module arrangements meet area and timing constraints.

### 9.3.4 EAPR Step 4: Analyze Timing and Placement

In this step each test wiring harness from Step 3 is analyzed to ensure that it meets area and timing constraints. If all constraints are met execution then proceeds on to Step 5 to implement the actual wiring harnesses and modules.

If area constraints are not met the place-and-route process will not succeed. The Timing Report and Circuit Evaluator (TRACE) tool reports any paths that fail timing constraints, and control can return to Step 2 to make an attempt to reposition slice macros such that the timing constraints are met.

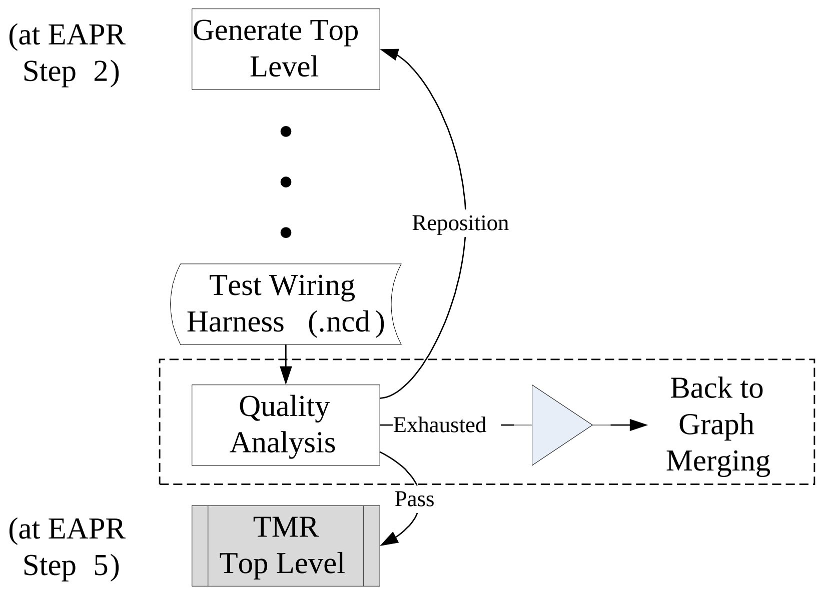


**Figure 9.8:** COMMA implementation of Step 4 in the EAPR tool flow

This is shown in Figure 9.8 with the arrow marked "Reposition". The slice macros attached to the paths that fail timing constraints are repositioned such that they are closer to each other. When all possibilities for repositioning slice macros are exhausted, then control can return to the graph merging phase to determine a different set of period arrangements. This is supported via the "SamePeriod" and "NoMerge" constraints for graph merging described in Sections 4.7.3 and 4.8.

### 9.3.5 EAPR Step 5: Implement Base Design

In Step 5 the actual wiring harness for each period is implemented using the top level HDL, constraints and slice macros generated in Step 2. This step is identical to that in the standard EAPR tool flow, as shown in Figure 9.9.

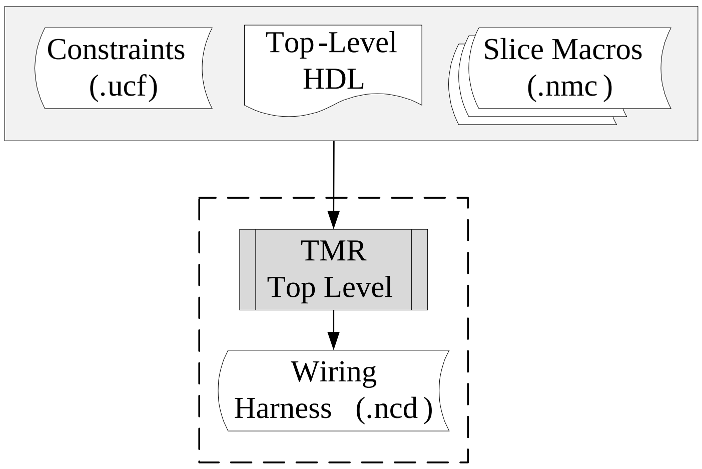


**Figure 9.9:** COMMA implementation of Step 5 in the EAPR tool flow

Perhaps the most important issue to note here is that the wiring harness is actually the base design. The base design also contains a very small Reconfiguration Control Module (RCM) (see Section 9.4), which facilitates the loading of bitstreams. The resulting circuit (NCD file) for each period thus consists of the RCM and the slice macros, and the wiring between them and the IOBs.

### 9.3.6 EAPR Step 6: Implement Partially-Reconfigurable Modules

In this step each module is implemented and a circuit description (NCD file) is generated for it, as shown in Figure 9.10. Note that one NCD file is generated for each module type in each slot in each period. The COMMA methodology does not currently support bitstream relocation, so different NCD files are generated for a module placed in different slots. This is in line with the standard EAPR tool flow and thus no explicit bitstream manipula-

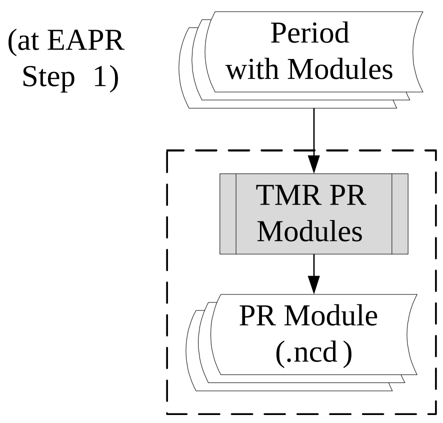


**Figure 9.10:** COMMA implementation of Step 6 in the EAPR tool flow

tion is performed, minimizing the risk of implementation errors and possible damage to the FPGA.

### 9.3.7 EAPR Step 7: Merge Base and Partially-Reconfigurable Modules

In this step the wiring harness and partially reconfigurable modules are merged together to form two separate full circuits — one each for the first epoch and the final epoch in the period. Full bitstreams are generated for these epochs, and partial bitstreams are generated for each module in each slot.

The initial and final epoch bitstreams are stored into the inter-period bitstream generator process to generate difference bitstreams between period transitions as shown in Figure 9.11. Instead of performing a full configuration between periods, the bitstream difference between the final epoch of an outgoing period and the first epoch of an incoming period is generated and used to reduce the reconfiguration overhead between periods.

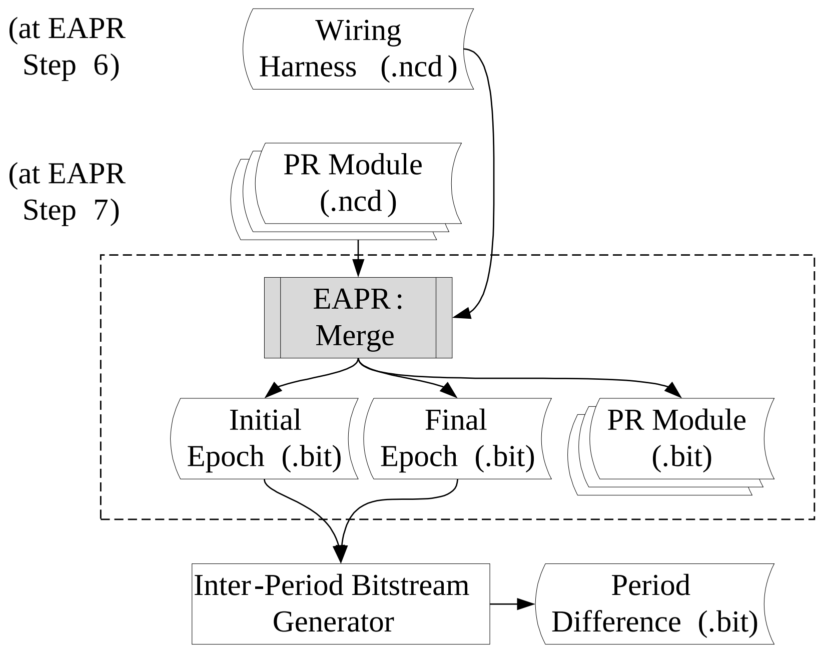


**Figure 9.11:** COMMA implementation of Step 7 in the EAPR tool flow with the inter-period difference bitstream generator

The final output of the implementation phase is a configuration bitstream for the initial epoch of each period, which is a full bitstream if it is the first period, or a difference bitstream if not, and a partial module bitstream for each module in each slot in each epoch.

## 9.4 Reconfiguration Control

The COMMA methodology currently employs *internal reconfiguration*, i.e., it assumes that the impetus for reconfiguration comes from within the FPGA, rather than being managed by off-chip controllers. Current Virtex-4 devices do not have sufficient on-chip memory to contain the bitstreams, so the bitstreams are stored off-chip in flash memory or RAM.

A Reconfiguration Control Module (RCM) is built into every wiring harness to facilitate the loading of bitstreams. The RCM is extremely lightweight, consisting only of a register that provides the control signals to the ICAP.

At present the graph merging process adds two modules into each epoch, which are input and output buffer control modules supporting data transfer between epochs. During the run of an epoch the output data is buffered by the output buffer module into on- or off-chip memory. This data is then used by the next epoch through the input buffer module, sourcing the data from the same on- or off-chip memory.

The input module also provides bitstream reconfiguration control. At the end of each epoch, it triggers the RCM to reconfigure each module that needs to be replaced in the next epoch. The start and end addresses of each partial bitstream for each slot are pre-registered with the RCM, thus allowing it to choose the appropriate bitstream and to perform the module reconfiguration. It registers the reconfiguration of its own slot last. When the input module for the next epoch is configured it triggers the start of the epoch by providing the buffered data from the previous epoch to the modules in the current epoch.

## 9.5 Conclusion

This chapter described the tool flow and tools necessary to perform the low-level implementation stages, and thus completes the description of the COMMA methodology.

The aim of this is to provide an automated, straightforward and tightly integrated implementation process. The system designer works at a higher level of abstraction, as the implementation flow described in this chapter does not require any intervention from the system designer at any point. However, the system designer can still specify options in the configuration files to tweak the process, such as constraining the placement of slice macros *a priori*.

# Chapter 10

# Conclusions and Future Work

## 10.1 Summary and Conclusions

Module-based dynamic reconfiguration of FPGAs enables the use of smaller devices through hardware virtualization and enhances throughput by multi-tasking. It also provides greater flexibility by allowing user-dependent application execution while maximizing the use of logic resources by loading modules when they are needed. This thesis identified that there is a necessity to provide top-down methodologies to assist systems designers in implementing applications that use module-based dynamic reconfiguration. Such methodologies should start from an application specification, properly partition and schedule it for dynamic reconfiguration, and finally implement it onto an actual FPGA device. It is desirable to apply low-level device implementation constraints, e.g., area and timing constraints, across all the stages of the methodology so as to it maximize the probability of implementation feasibility.

This thesis proposed a top-down methodology for implementing dynamically reconfigurable applications to target the new page-reconfigurable Xilinx Virtex-4 FPGA family. The methodology structures the designer's use of the EAPR partial reconfiguration toolflow provided by Xilinx [95].

The focus of the thesis has been to provide a communications infrastructure within the methodology that supports the dynamically changing communications interfaces of the modules. To implement such an infrastructure, the methodology uses a fixed module slot layout on the FPGA with a point-to-point wiring harness surrounding the slots providing the intermodule communications needs. The layout takes advantage of the two-dimensional paged reconfiguration architecture of the Virtex-4 FPGA family by allowing independent reconfiguration of each slot in each page. The point-to-point wiring also minimizes communications overheads and does not enforce adherence to any particular communications protocol.

Using such a regular layout may incur penalties to the critical path delay as compared to traditional place-and-route methods that encourage the compaction of module logic freely placed about the FPGA. A study was performed to analyze the critical path overheads and it was found that the penalty incurred was acceptable at high communication densities. In some cases the regular layout was more beneficial than traditional free-form compaction and placement methods. Given that the methodology encourages maximal utilization of the device area, the communications densities will foreseeably be high and thus the overheads are acceptable. This is a remarkably good result since the layout provides substantial benefits in isolating regions that are reconfigured independently (in the form of modules).

In addition, the time to place-and-route circuits in this regular layout was markedly reduced (see Figure 5.9).

A method was proposed to generate wiring harnesses that implement the communications infrastructure for dynamically placed modules. The method involves merging subgraphs, each of which is a temporal partition of the full communications graph representing a configuration that is loaded onto the FPGA. Each merged subgraph is then transformed into a wiring harness that support the communications for all the modules in the subgraph. The aim of subgraph merging is to reduce reconfiguration delay by implementing wiring harnesses that support the longest possible configuration sequence on the device. An application may require more than one wiring harness for its entire execution, thus wiring harnesses may need to be reconfigured as well.

Two algorithms were proposed for the merging of subgraphs, an initial one based on greedy methods and an improved algorithm based on dynamic programming. The dynamic programming algorithm considers many more possibilities for subgraph merging and aims to minimize the total reconfiguration delay. An algorithm based on FPGA routability estimation was proposed to map merged subgraphs to the device. The algorithm estimates area consumption, reconfiguration delay and the contribution to the critical path delay from a wiring harness.

An experimental framework was developed and the greedy and dynamic methods for graph merging were assessed. Experiments with large communications graphs were performed on all devices in the Virtex-4 LX family. The results show that graph merging is clearly beneficial and reduces the reconfiguration delays significantly. The dynamic programming algorithm consistently provides better reconfiguration delay and critical path delay results than the greedy method. The experimental framework also demonstrates the effectiveness of the methodology in aiding systems designers to perform architectural exploration in order to select the best device size given the budget and other constraints of the application.

The results of the experiments illustrate an interesting phenomenon whereby the use of a smaller device may result in a lower reconfiguration delay than a larger one. This was found to be due to a sub-process that is performed during graph merging known as module clustering. Module clustering packs as much module logic into each slot as possible to maximize area utilization, but the number of apparent module types also increases as a side effect of this. As a result, fewer opportunities are present to reduce reconfiguration delay by placing modules of the same type in the same slot in consecutive configurations. Disabling clustering, or using a smaller device mitigates this effect, but improvements to the clustering and scheduling algorithms to support the methodology are desirable.

In general, clustering was found to be beneficial to reducing reconfiguration delay and critical path delay as device sizes increased.

Finally, the low-level implementation stages were described, involving integration into the existing Xilinx Early Access Partial Reconfiguration tool flow. The final output of the methodology is a set of bitstreams to be configured onto the FPGA.

## 10.2 Directions for Further Study

Several immediate directions for further investigation can be identified from the methodology proposed in this thesis and from the results that were obtained.

The focus of the thesis was on generating communications infrastructures to support the changing communications interfaces of dynamic modules. Application- and methodology-neutral clustering and scheduling (partitioning) algorithms were used, and it was identified from the results that improvements can be made to these algorithms in order to customize them to specifically target this methodology. Feasible enhancements include minimizing the number of apparent module types during module clustering, and maximizing the number of equivalent module types in adjacent partitions during scheduling.

The methods described in this thesis apply to application problems that can be modeled as a linear sequence of configurations. In general, more sophisticated applications that require looping, forking or joining sequences are not currently supported. Some manual constraint methods are described in Chapter 4 and a thorough assessment of them or other methods is desirable.

The task graphs studied to date have an implied data dependency, i.e., a module produces some data for another module to consume, and completes its execution. The modeling of the task graphs could be improved to be more realistic, i.e., to capture the timing of events and communication patterns of the modules in more detail.

The graph merging algorithms can also be very well improved upon to minimize reconfiguration overheads, as the dynamic programming algorithm assumes a heuristic of a full reconfiguration delay between periods. The wiring harnesses are currently generated automatically through ISE's place-and-route. The amount of bitstream difference between wiring harnesses can be minimizes by building custom harnesses or trying to guide ISE's place-and-route.

The lack of benchmarks for reconfigurable computing makes it difficult to test the method and compare the results. Further investigation into creating benchmarks and representative applications for reconfigurable computing is desirable.

Finally, the specific problem that was tackled in this thesis is one in which the application's modules, their communication requirements and temporal relationships are all known at design time. This information enables the optimized synthesis of communications infrastructures when a standard module interface is not imposed. When information about the modules and their communications is not available *a priori*, the methods described in this thesis encourage the pre-allocation of the estimated communications requirements ahead of time. In effect, these methods are not currently practical for such applications. Adapting the methodology to a more general communications infrastructure such as a network-on-chip may allow an FPGA to support paged modular reconfiguration without prior knowledge of the communications needs of applications or explicit prior knowledge of the set of applications that are active at run time. This is an essential step towards using reconfigurable hardware in general-purpose multitasking environments.

# Appendix A

# Result Plots for 40, 80, 120 and 160 modules

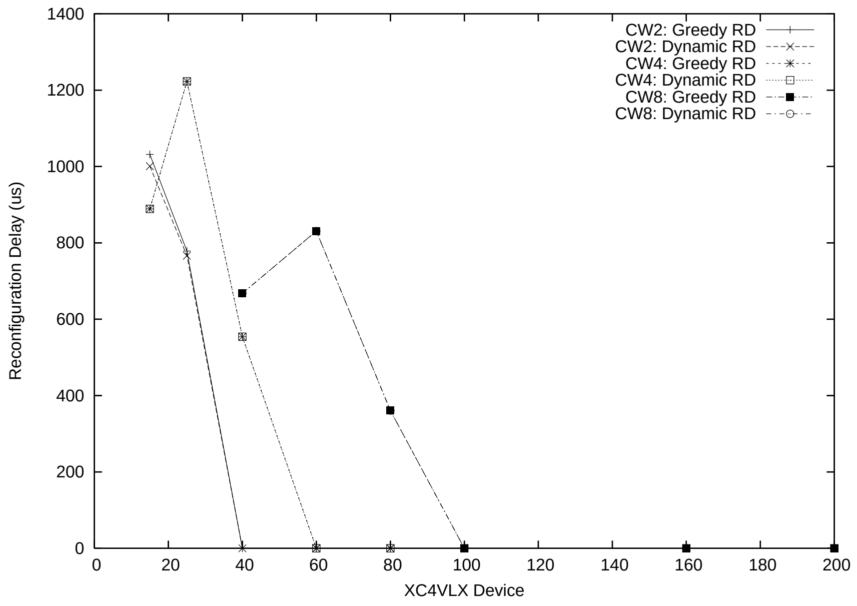


**Figure A.1:** Reconfiguration delays: 40 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

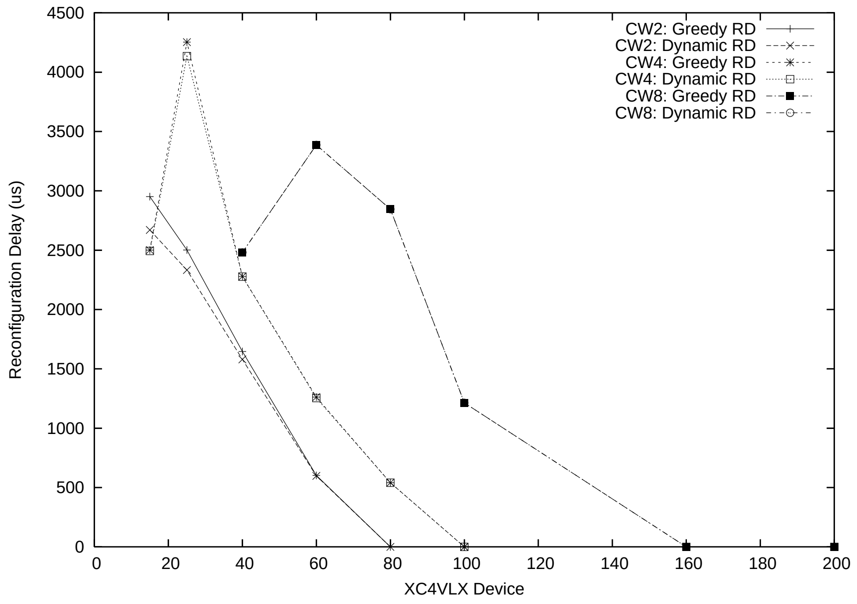


**Figure A.2:** Reconfiguration delays: 80 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

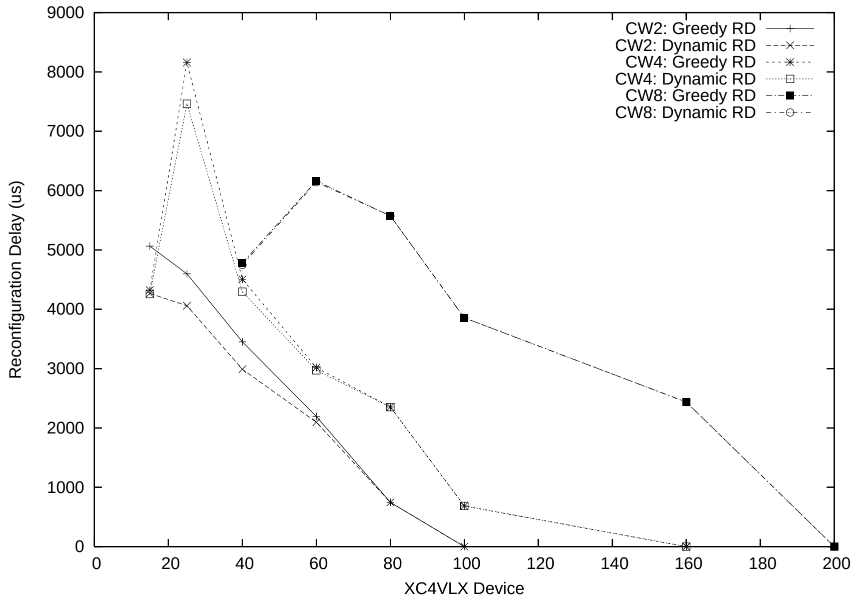


**Figure A.3:** Reconfiguration delays: 120 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

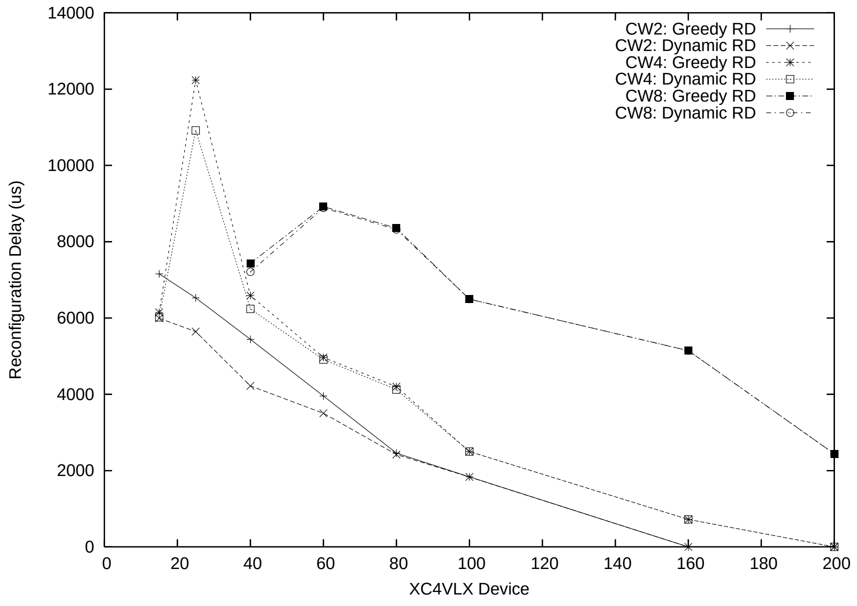


**Figure A.4:** Reconfiguration delays: 160 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

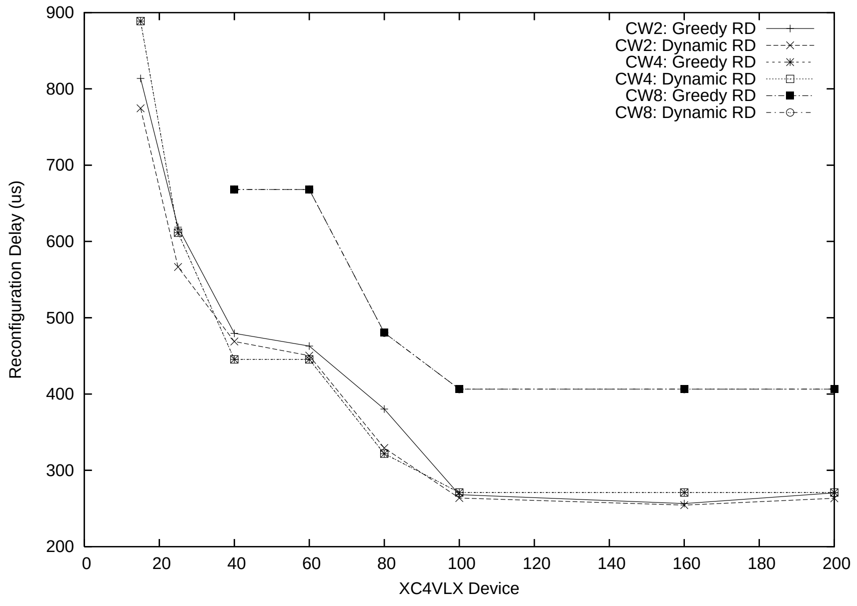


**Figure A.5:** Reconfiguration delays: 40 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

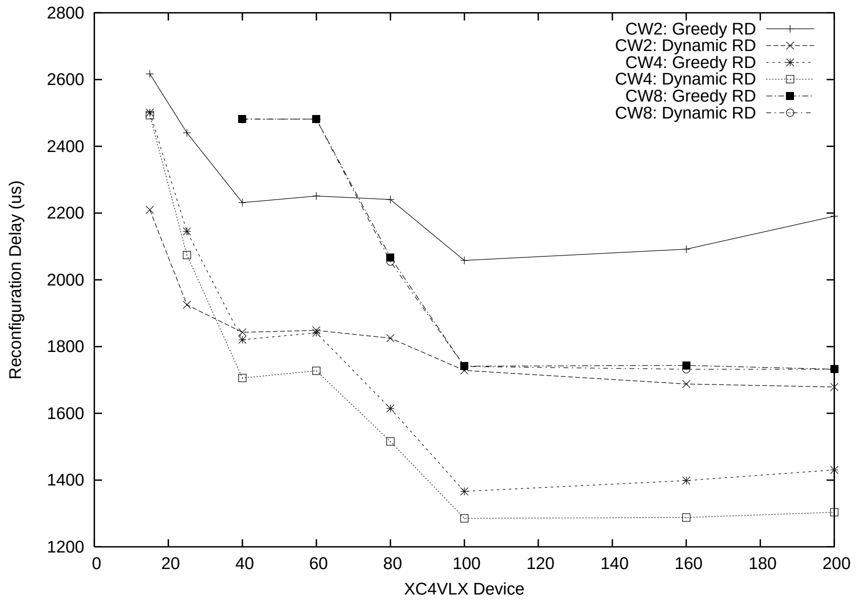


**Figure A.6:** Reconfiguration delays: 80 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

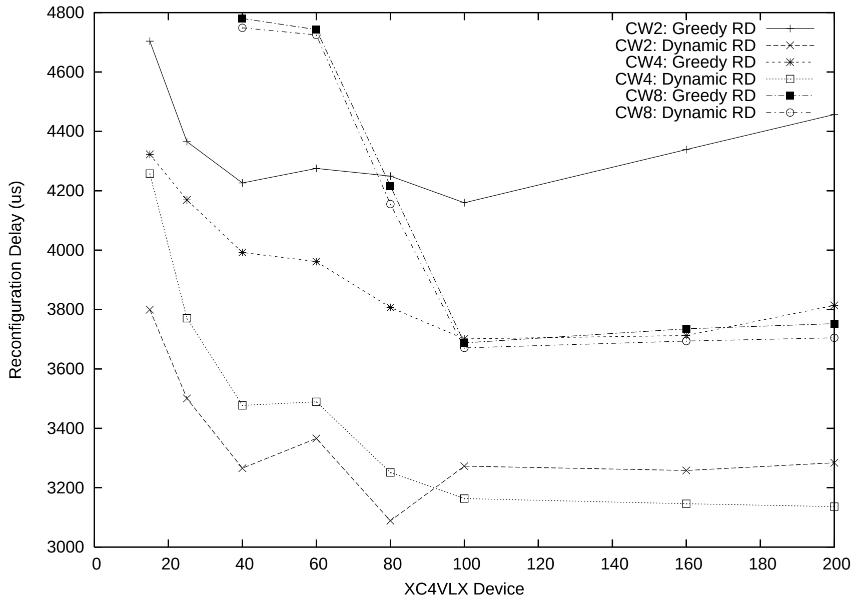


**Figure A.7:** Reconfiguration delays: 120 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

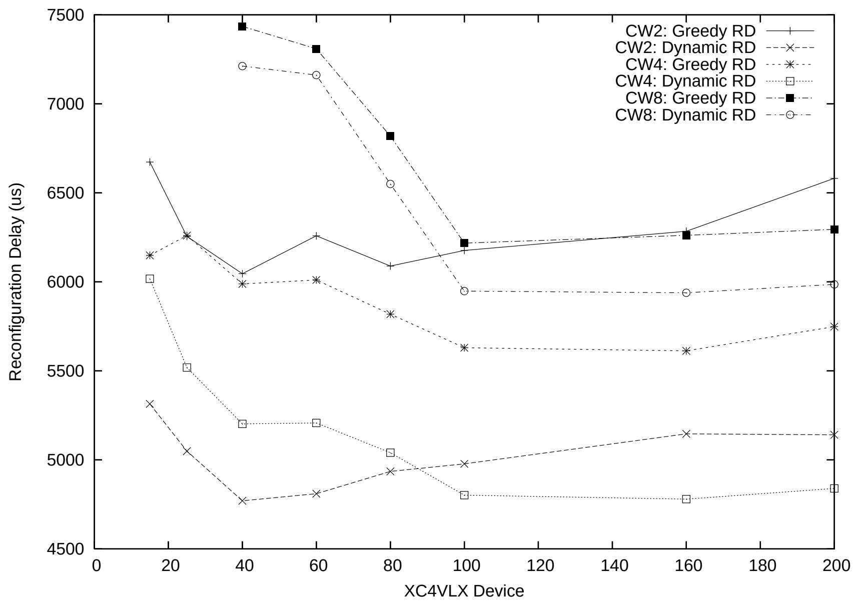


**Figure A.8:** Reconfiguration delays: 160 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

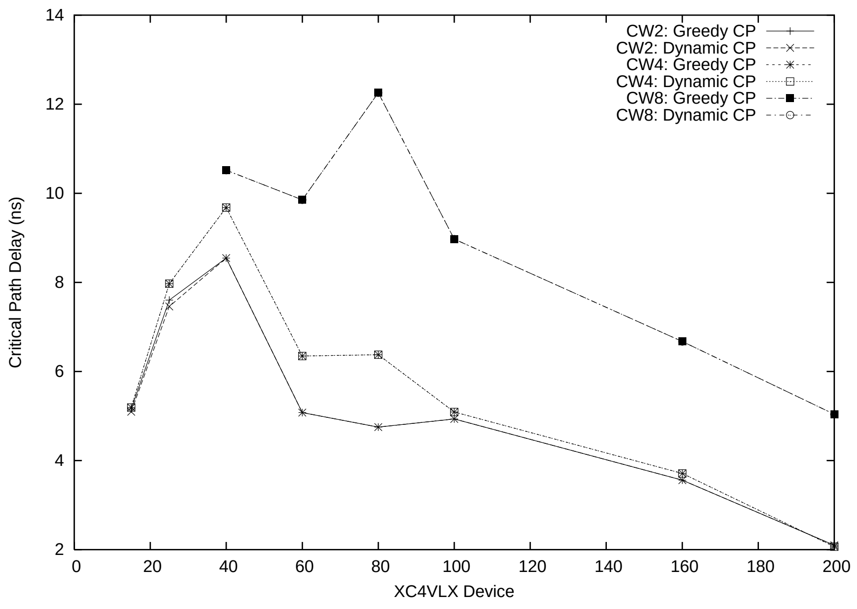


**Figure A.9:** Critical path delays: 40 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

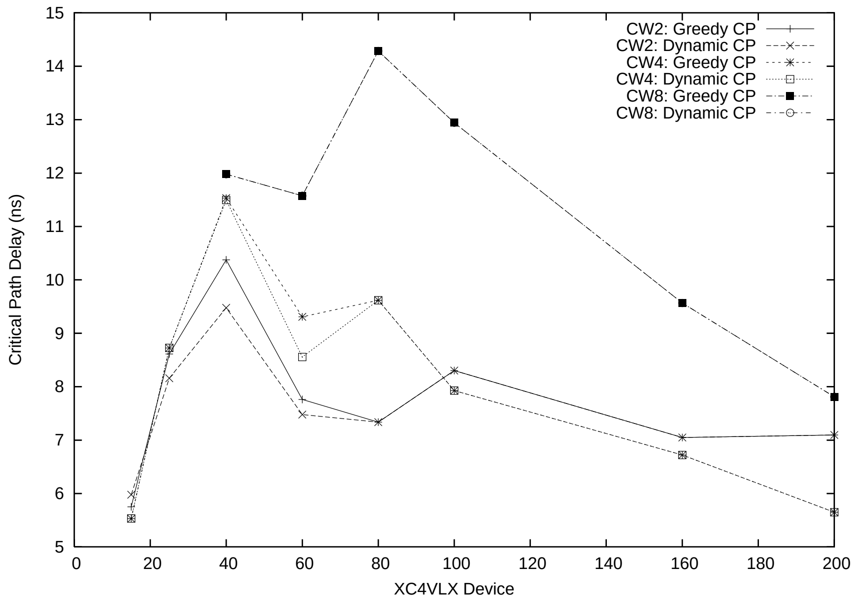


**Figure A.10:** Critical path delays: 80 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

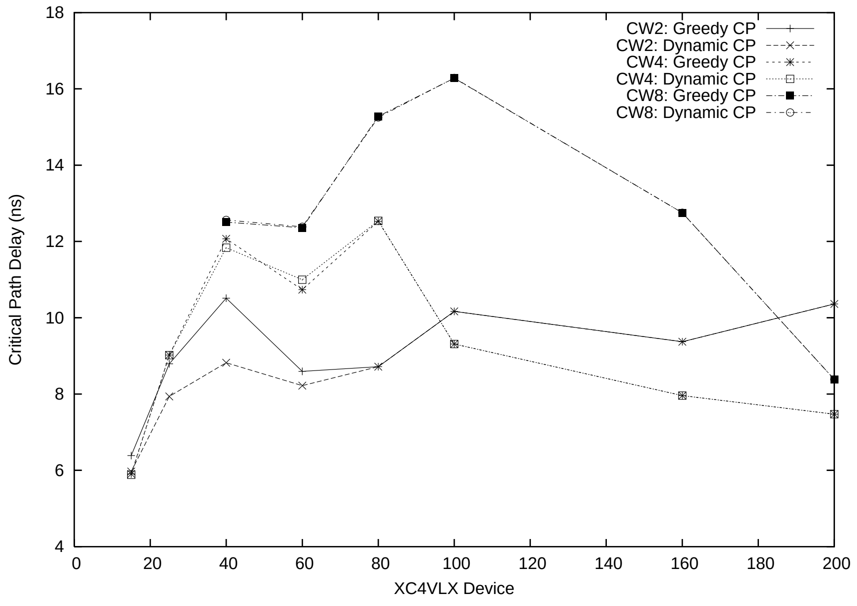


**Figure A.11:** Critical path delays: 120 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

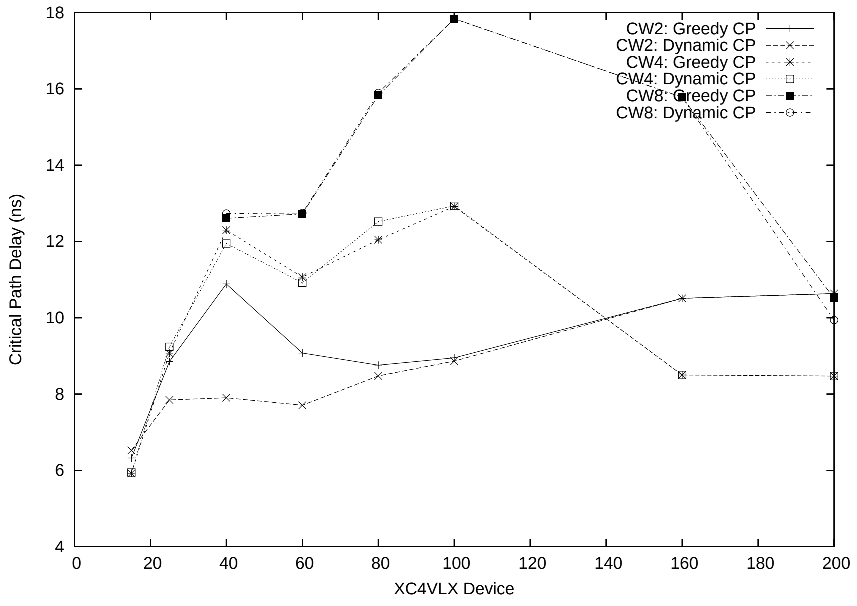


**Figure A.12:** Critical path delays: 160 modules, 20% type variation, 60 CLB exact module size, 120 runs, clustered

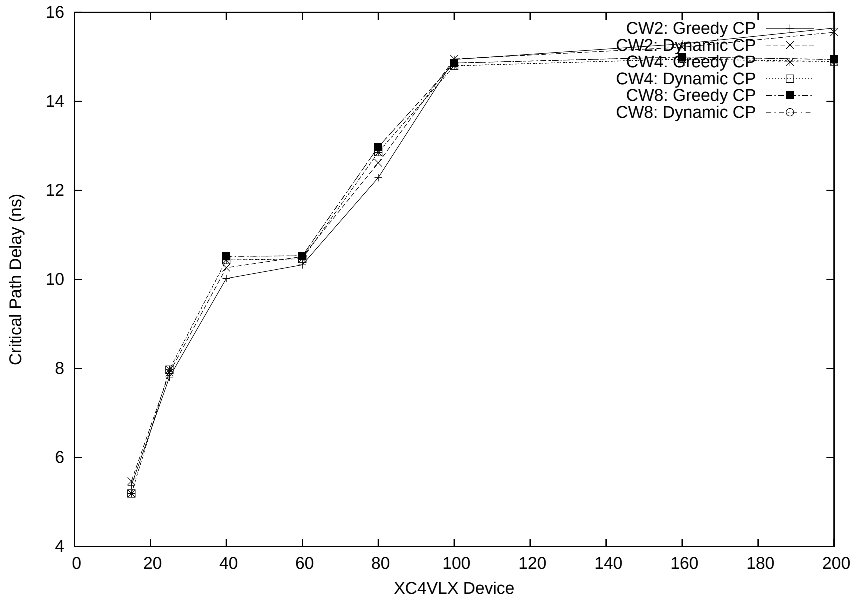


**Figure A.13:** Critical path delays: 40 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

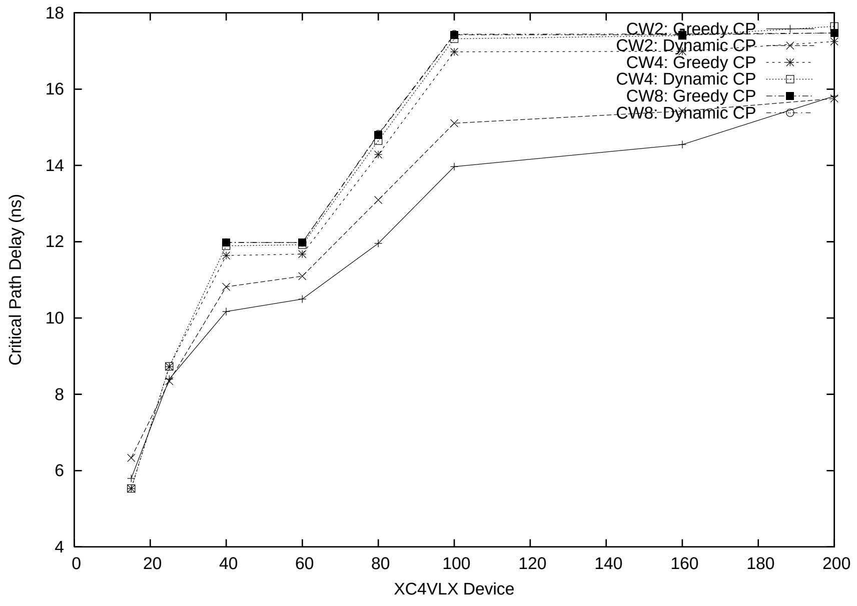


**Figure A.14:** Critical path delays: 80 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

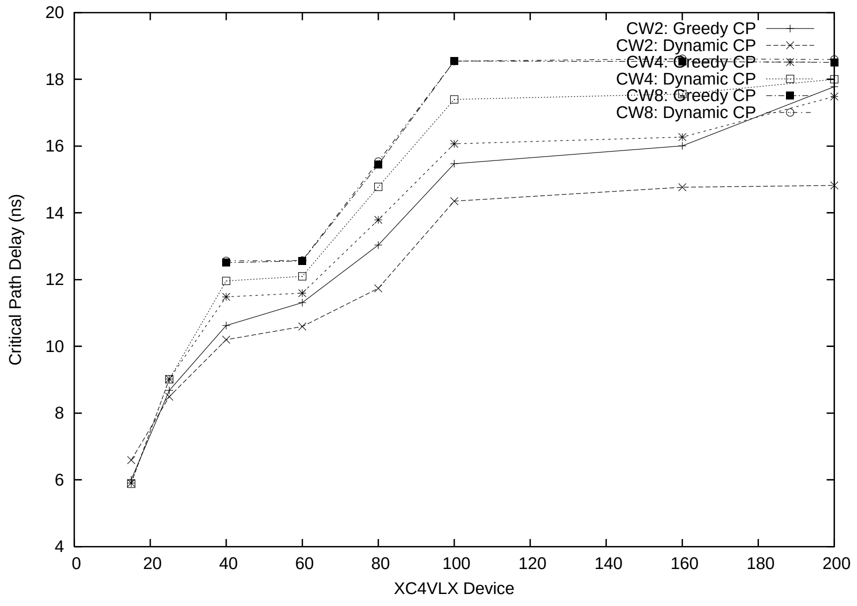


**Figure A.15:** Critical path delays: 120 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

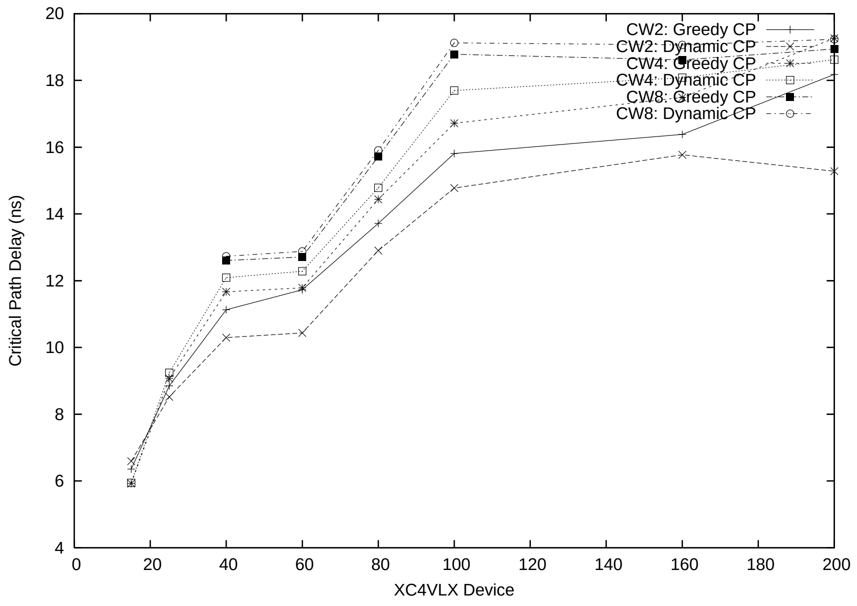


**Figure A.16:** Critical path delays: 160 modules, 20% type variation, 60 CLB exact module size, 120 runs, non-clustered

# Bibliography

[1] Ahmadinia, A., Bobda, C., Ding, J., Majer, M. and Teich, J. A practical approach for circuit routing on dynamic reconfigurable devices. In *International Workshop on Rapid System Prototyping*, pages 84–90, Montréal, Canada, 2005.

[2] Ali, F.M. and Das, A.S. Hardware-software co-synthesis of hard real-time systems with reconfigurable FPGAs. *Computers and Electrical Engineering*, 30(7):471–489, 2004.

[3] Balachandran, S. and Bhatia, D. A priori wirelength and interconnect estimation based on circuit characteristics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1054–1065, 2005.

[4] Balachandran, S., Kannan, P. and Bhatia, D. On routing demand and congestion estimation for FPGAs. In *Asia and South Pacific Design Automation Conference and the International Conference on VLSI Design*, pages 639–646, Bangalore, India, 2002.

[5] Banerjee, S., Bozorgzadeh, E. and Dutt, N. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic recon-

figuration. In *Design Automation Conference*, pages 335–340, Anaheim, California, USA, 2005. ACM.

[6] Becker, T., Luk, W. and Cheung, P.Y.K. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In *International Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, California, USA, 2007.

[7] Betz, V. and Rose, J. FPGA routing architecture: segmentation and buffering to optimize speed and density. In *International Symposium on Field Programmable Gate Arrays* , pages 59–68, Monterey, California, USA, 1999. ACM.

[8] Blodget, B., Bobda, C., Hübner, M. and Niyonkuru, A. Partial and dynamically reconfiguration of Xilinx Virtex-II FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 801–810, Antwerp, Belgium, 2004.

[9] Bobda, C. Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement (Dr rer. nat. thesis). *University of Paderborn*, 2003.

[10] Bobda, C., Ahmadinia, A., Majer, M., Teich, J., Fekete, S. and Veen, J.v.d. DyNoC: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *International Conference on Field Programmable Logic and Applications*, pages 153–158, Tampere, Finland, 2005.

[11] Bobda, C., Majer, M., Koch, D., Ahmadinia, A. and Teich, J. A dynamic NoC approach for communication in reconfigurable devices. In *International Conference on Field-Programmable Logic and Applications*, pages 1032–1036, Antwerp, Belgium, 2004.

[12] Bondalapati, K. Modeling and mapping for dynamically reconfigurable hybrid architectures. *PhD Thesis*, 2001.

[13] Bondalapati, K. and Prasanna, V.K. Dynamic precision management for loop computations on reconfigurable architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, California, 1999. IEEE.

[14] Brebner, G. The Swappable Logic Unit: A paradigm for virtual hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, Napa Valley, California, USA, 1997.

[15] Bruhn, A.e.s., Weickert, J., Feddern, C., Kohlberger, T. and Schnörr, C. Real-time optic flow computation with variational methods. In *Computer Analysis of Images and Patterns*, volume 2756, pages 222–229. Springer Berlin / Heidelberg, 2003.

[16] Chan, J. and Parameswaran, S. NoCOUT : NoC topology generation with mixed packet-switched and point-to-point networks. In *Asia and South Pacific Design Automation Conference*, COEX, Seoul, Korea, 2008.

[17] Chaouat, L., Garin, S., Vachoux, A. and Mlynek, D. Rapid prototyping of hardware systems via model reuse. In *IEEE International Work-*

*shop on Rapid System Prototyping*, pages 150–156, Chapel Hill, North Carolina, USA, 1997.

[18] Chatha, K.S. and Vemuri, R. Hardware-software codesign for dynamically reconfigurable architectures. In *International Conference on Field-Programmable Logic*, pages 175–185, Glasgow, UK, 1999.

[19] Compton, K. and Hauck, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, 2002.

[20] Cormen, T.H., Leiserson, C.E. and Rivest, R.L. Introduction to Algorithms. *The MIT Press*, 1990.

[21] Dally, W.J. and Seitz, C.L. The torus routing chip. *Distributed Computing*, 1:187–196, 1986.

[22] Edwards, M. and Green, P. Run-time support for dynamically reconfigurable computing systems. *Journal of Systems Architecture*, 49(4–6):267–281, 2003.

[23] Eles, P., Kuchcinski, K., Peng, Z., Doboli, A. and Pop, P. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Design, Automation and Test in Europe*, pages 132–139, Paris, France, 1998.

[24] Elgindy, H., Schröder, H., Spray, A., Somani, A.K. and Schmeck, H. RMB — a reconfigurable multiple bus network. In *International Symposium on High-Performance Computer Architecture*, pages 108–117, San Jose, CA, USA, 1996. IEEE.

[25] Esquiagola, J., Ozari, G., Teruya, M., Strum, M. and Chau, W. A dynamically reconfigurable Bluetooth base band unit. In *International Conference on Field Programmable Logic and Applications*, pages 148–152, Tampere, Finland, 2005.

[26] Fekete, S.P., Köhler, E. and Teich, J. Optimal FPGA module placement with temporal precedence constraints. In *Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, 2001. IEEE.

[27] Ganesan, S. and Vemuri, R. An integrated temporal partitioning and partial reconfiguration technique for design latency improvement. In *Design Automation and Test in Europe*, pages 320–325, Paris, France, 2000. ACM.

[28] Ghiasi, S., Nahapetian, A. and Sarrafzadeh, M. An optimal algorithm for minimizing run-time reconfiguration delay. *ACM Transactions on Embedded Computing Systems*, 3(2):237–256, 2004.

[29] Guccione, S. and Levi, D. Run-time parameterizable cores. In *International Conference on Field-Programmable Logic and Applications*, pages 215–222, Glasgow, UK, 1999. Springer-Verlag.

[30] Guccione, S., Levi, D. and Sundararajan, P. JBits — Java based interface for reconfigurable computing. In *Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, John Hopkins University, Maryland, USA, 1999.

[31] Hagemeyer, J., Kettelhoit, B., Köster, M. and Porrmann, M. A design methodology for communication infrastructures on partially recon-

figurable FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 331–338, Amsterdam, The Netherlands, 2007. IEEE.

[32] Hart, P.E., Nillson, N.J. and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 2:100–107, 1968.

[33] Heron, J.-P. and Woods, R.F. Accelerating run-time reconfiguration on FCCMs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 260–261, Napa Valley, CA, 1999.

[34] Hübner, M., Becker, T. and Becker, J. Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration. In *Symposium on Integrated Circuits and Systems Design*, pages 28–32, Lafayette, Los Angeles, USA, 2004.

[35] Hübner, M., Schuck, C., Kühnle, M. and Becker, J. New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits . In *IEEE Computer Society Annual Symposium on VLSI: Emerging VLSI Technologies and Architectures*, pages 97–102, Karlsruhe, Germany, 2006.

[36] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Standard P1076 2004-10*, 2004.

[37] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Standard 1364 -2005*, 2006.

[38] Janac, G., Poltronetti, T., Herbert, A. and RuDusky, D. IP supply chain-the design reuse paradigm comes of age. *Integrated System Design*, 13(141):66–70, 2001.

[39] Kalte, H. and Porrmann, M. REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Conference on Computing Frontiers*, pages 403–412, Ischia, Italy, 2006.

[40] Kalte, H., Lee, G., Porrmann, M. and Rückert, U. Study on column wise design compaction for reconfigurable systems. In *IEEE International Conference on Field-Programmable Technology*, pages 413–416, Brisbane, Australia, 2004.

[41] Kalte, H., Lee, G., Porrmann, M. and Rueckert, U. REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In *IEEE International Parallel and Distributed Processing Symposium*, page 151b, Denver, Colorado, USA, 2005.

[42] Kalte, H., Porrmann, M. and Rückert, U. A prototyping platform for dynamically reconfigurable system on chip design. In *IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip*, Hamburg, Germany, 2002.

[43] Kalte, H., Porrmann, M. and Rückert, U. System-on-Programmable-Chip approach enabling online fine-grained 1D-placement. In *International Parallel and Distributed Processing Symposium*, pages 141–148, Santa Fe, New Mexico, USA, 2004.

[44] Kannan, P., Balachandran, S. and Bhatia, D. On metrics for comparing routability estimation methods for FPGAs. In *Design Automation Conference*, pages 70–75, New Orleans, Louisiana, USA, 2002.

[45] Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[46] Katajainen, J., Pasanen, T. and Teuhola, J. Practical in-place mergesort. *Nordic Journal of Computing*, 3(1):27–40, 1996.

[47] Kernighan, B.W. and Lin, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[48] Kim, J., Ha, D.S. and Reed, J.H. A new reconfigurable modem architecture for 3G multi-standard wireless communication systems. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 1051–1054, Kobe, Japan, 2005.

[49] Koh, S. and Diessel, O. COMMA: a communications methodology for dynamic module-based reconfiguration of FPGAs. In *International Conference on Architecture of Computing Systems, Dynamically Reconfigurable Systems Workshop Proceedings*, pages 173–182, Frankfurt, Germany, 2006.

[50] Koh, S. and Diessel, O. COMMA: a communications methodology for dynamic module reconfiguration in FPGAs. In *IEEE Symposium on*

*Field-Programmable Custom Computing Machines*, pages 273–274, Napa Valley, California, 2006.

[51] Koh, S. and Diessel, O. Communications infrastructure generation for modular FPGA reconfiguration. In *IEEE International Conference on Field Programmable Technology*, pages 321–324, Bangkok, Thailand, 2006. IEEE.

[52] Koh, S. and Diessel, O. Module graph merging and placement to reduce reconfiguration overheads in paged FPGA devices. In *International Conference on Field Programmable Logic and Applications*, pages 293–298, Amsterdam, The Netherlands, 2007. IEEE.

[53] Koh, S. and Diessel, O. The effectiveness of configuration merging in point-to-point networks for module-based FPGA reconfiguration. In *IEEE Symposium on Field-Programmable Custom Computing Machines (in submission)*, Palo Alto, California, 2008. IEEE.

[54] Köster, M., Porrmann, M. and Kalte, H. Relocation and defragmentation for heterogeneous reconfigurable systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 70–76, Volos, Greece, 2006.

[55] Lou, J., Thakur, S., Krishnamoorthy, S. and Sheng, H.S. Estimating routing congestion using probabilistic analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(1):32–41, 2002.

[56] Lysaght, P., Blodget, B., Mason, J., Young, J. and Bridgford, B. Invited paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 1–6, Madrid, Spain, 2006.

[57] Majer, M., Teich, J., Ahmadinia, A. and Bobda, C. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing*, 47(1), 2007.

[58] Malik, U. Configuration encoding techniques for fast FPGA reconfiguration. *PhD Thesis*, 2006.

[59] Manohararajah, V., Chiu, G.R., Singh, D.P. and Brown, S.D. Difficulty of predicting interconnect delay in a timing driven FPGA CAD flow. In *International Workshop on System-Level Interconnect Prediction*, pages 3–8, San Diego, California, USA, 2006.

[60] Marescaux, T., Bartic, A., Verkest, D., Vernalde, S. and Lauwereins, R. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 741–763, Montpellier, France, 2002.

[61] McMurchie, L. and Ebeling, C. PathFinder: A negotiation-based performance-driven router for FPGAs. In *International ACM Symposium on Field-Programmable Gate Arrays*, pages 111–117, Monterey, California, USA, 1995.

[62] Morris, K. FPGA BASE jump — Partial reconfiguration for SDR. *FPGA and Structured ASIC Journal*, 17(9), 2007.

[63] Murphy, C.W. and Harvey, D.M. Reconfigurable hardware implementation of BinDCT. *Electronics Letters*, 38:1012–1013, 2002.

[64] Noguera, J. and Basia, R.M. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *ACM Transactions on Embedded Computing Systems*, 3(2):385–406, 2004.

[65] Oliveira, A. and Sklyarov, V. Implementation of virtual control circuits in dynamically reconfigurable FPGAs. In *IEEE International Conference on Electronics, Circuits and Systems*, pages 217–220, Pafos, Cyprus, 1999.

[66] Ouaiss, I., Govindarajan, S., Srinivasan, V., Kaul, M. and Vemuri, R. An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures. In *Reconfigurable Architectures Workshop*, pages 31–36, Orlando, Florida, USA, 1998.

[67] Pang, V. Real-time optical-flow computation using FPGAs. *Computer Engineering Undergraduate Thesis, School of Computer Science and Engineering, The University of New South Wales*, 2005.

[68] Parthasarathy, G., Marek-Sadowska, M., Mukherjee, A. and Singh, A. Interconnect complexity-aware FPGA placement using Rent's rule. In *International Workshop on System-Level Interconnect Prediction*, pages 115–121, Sonoma, California, United States, 2001.

[69] Purna, K.M.G. and Bhatia, D. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, 1999.

[70] Quinn, H., King, L.A.S., Leeser, M. and Meleis, W. Runtime assignment of reconfigurable hardware components for image processing pipelines. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 173–182, Napa, California, 2003.

[71] Roy, J.A., Papa, D.A., Adya, S.N., Chan, H.H., Ng, A.N., Lu, J.F. and Markov, I.L. Capo: Robust and scalable open-source min-cut floorplacer. In *International Symposium on Physical Design*, pages 224–226, San Francisco, California, USA, 2005. IEEE.

[72] Sarkar, V. Partitioning and scheduling parallel programs for multiprocessors. *MIT Press*, 1989.

[73] Sedcole, P., Blodget, B., Anderson, J., Lysaght, P. and Becker, T. Modular partial reconfiguration in Virtex FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 211–216, Tampere, Finland, 2005.

[74] Singh, A. and Marek-Sadowska, M. FPGA interconnect planning. In *International Workshop on System-Level Interconnect Prediction*, pages 23–30, San Diego, California, USA, 2002.

[75] Sklyarov, V., Lau, N., Oliveira, A., Melo, A., Kondratjuk, K., Ferrari, A., Monteiro, R. and Skliarova, I. Synthesis tools and design environ-

ment for dynamically reconfigurable FPGAs. In *Brazilian Symposium on Integrated Circuit Design*, pages 46–49, Rio de Janeiro, 1998.

[76] Synplicity. Synplify and Synplify Pro. *http://www.synplicity.com/products/synplifypro/*, 2008.

[77] Tanougast, C., Berviller, Y., Brunet, P., Weber, S. and Rabah, H. Temporal partitioning methodology optimizing FPGA resources for dynamically reconfigurable embedded real-time system. *Microprocessors and Microsystems*, 27(3):115–130, 2003.

[78] Tessier, R.G. Fast place and route approaches for FPGAs. *PhD Thesis*, Massachusetts Institute of Technology, 1999.

[79] Uhm, M. Software-defined radio: the new architectural paradigm. *DSP magazine*, 1(1):40–42, 2005.

[80] Ullmann, M., Hübner, M., Grimm, B. and Becker, J. On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities. In *International Conference on Field Programmable Logic and Applications*, pages 454–463, Leuven, Belgium, 2004.

[81] Villasenor, J., Jones, C. and Schoner, B. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):565–567, 1995.

[82] Xie, Y. and Wolf, W. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Design, Automation and Test in Europe*, pages 620–625, Munich, Germany, 2001.

[83] Xilinx. XC6200 Field Programmable Gate Arrays datasheet. *XC6200 Reconfigurable Programmable Logic Family v1.10*, 1997.

[84] Xilinx. Virtex$^{\mathrm{TM}}$ 2.5 V Field-Programmable Gate Arrays. *Datasheet DS003-1*, 2001.

[85] Xilinx. Configuration and readback of Virtex FPGAs using (JTAG) boundary scan. *Xilinx Application Note 139*, 2003.

[86] Xilinx. Two flows for partial reconfiguration: Module based or difference based. *Xilinx Application Note 290*, 2003.

[87] Xilinx. Virtex-4 family overview. *Datasheet DS112*, 2004.

[88] Xilinx. Virtex series configuration architecture user guide. *Application note 151*, 2004.

[89] Xilinx. Virtex-4 configuration guide. *User Guide UG071*, 2005.

[90] Xilinx. Virtex-II platform FPGA user guide. *User Guide UG002*, 2005.

[91] Xilinx. XtremeDSP design considerations user guide. *User Guide UG073*, 2005.

[92] Xilinx. Development System Reference Guide. *Xilinx ISE 8.2i Documentation*, 2006.

[93] Xilinx. Xilinx and ISR Technologies announce sotware-defined radio kit supporting partial reconfiguration and SCA-enabled SoC. *Xilinx Press Release 0626*, 2006.

[94] Xilinx. XST user guide. *Xilinx ISE 8.2i Documentation*, 2006.

[95] Xilinx. Early access partial reconfiguration user guide. *User Guide UG208*, 2007.

[96] Xilinx. PlanAhead user guide. *Xilinx PlanAhead Version 9.2*, 2007.

[97] Xilinx. Virtex-5 family overview: LX, LXT and SXT platforms. *Xilinx Datasheet DS100*, 2007.

[98] Xilinx, I. Partial reconfiguration workshop. In *International Conference on Field Programmable Logic and Applications*, Technische Universiteit Delft, The Netherlands, 2007.