To the Graduate Council:

I am submitting herewith a dissertation written by Nabil Kerkiz entitled "Development and Experimental Evaluation of Partitioning Algorithms for Adaptive Computing Systems." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Donald W. Bouldin

Dr. Donald W. Bouldin, Major Professor

We have read this dissertation and recommend its acceptance:

Dr. Mike Langston

Dr. Danny Newport

Dr. Dan Koch

Dr. Chandra Tan

Accepted for the Council:

Interim Vice Provost and
Dean of The Graduate School

Development and Experimental Evaluation of Partitioning Algorithms for Adaptive Computing Systems

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Nabil Kerkiz
December 2000

Acknowledgement

I would like to extend my sincere gratitude to my advisor, Dr. Don Bouldin, for his support and guidance in this project. Without him, this work could never have been completed. Special thanks to Dr. Chandra Tan for his effort and assistance which contributed greatly to this work. I would also like to thank Dr. Mike Langston, Dr. Danny Newport, and Dr. Dan Koch for serving as members of my thesis committee. I also acknowledge the Defense Advanced Research Projects Agency for its support of this research under grant F33615-97-C-1124.

Much appreciates and love is extended to all of my family and friends. I would like to especially acknowledge my parents, Fouad and Ganimah, and my wife, Huda, for their continued support and love throughout this work.

Abstract

Multi-FPGA systems offer the potential to deliver higher performance solutions than traditional computers for some low-level computing tasks. This requires a flexible hardware substrate and an automated mapping system. CHAMPION is an automated mapping system for implementing image processing applications in multi-FPGA systems under development at the University of Tennessee. CHAMPION will map applications in the Khoros Cantata graphical programming environment to hardware.

The work described in this dissertation involves the automation of the CHAMPION back-end design flow, which includes the partitioning problem, netlist to structural VHDL conversion, synthesis and placement and routing, and host code generation. The primary goal is to investigate the development and evaluation of three different k-way partitioning approaches. In the first and the second approaches, we discuss the development and implementation of two existing algorithms. The first approach is a hierarchical partitioning method based on topological ordering (HP). The second approach is a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic (RP). We extend these algorithms to handle the multiple constraints imposed by adaptive computing systems. We also introduce a new recursive partitioning method based on topological ordering and levelization (RPL). In addition to handling the partitioning constraints, the new approach efficiently addresses the problem of minimizing the number of FPGAs used and the amount of computation, thereby overcoming some of the weaknesses of the HP and RP algorithms.

Table of Contents

1. Introduction **1**

1.1 CHAMPION design flow 2

1.2 Field Programmable Gate Arrays 7

1.3 Hardware Architecture 10

1.3.1 Wildforce-XL Board 10

1.3.2 MSP Board 13

1.3.3 SLAAC Board 13

1.4 Khoros Cantata 15

1.5 Placement and Routing 18

2. Research Motivation and Background **21**

2.1 Partitioning Methods for Multi-PE System 21

2.1.1 Bi-Partitioning Methods 21

2.1.2 Multiway Partitioning Methods 28

2.1.3 Benefit Function 31

2.3 Partitioning Constraints 32

3. Partitioning Algorithms **41**

3.1 Hierarchical Partitioning Based on Topological Ordering (HP) 43

3.2 Recursive Algorithm Based on Fiduccia and Mattheyses Bipartitioning Heuristic (RP) 45

3.3 A New Recursive Partitioning Method Based on Topological Ordering and Levelization (RPL) 49

3.3.1 Level Construction step 51

3.3.3 Partitioning Step 54

4. Experimental Results and Analysis 59

4.1 HP Algorithm 67

4.3 RP Algorithm 76

4.1 RPL Algorithm 84

4.4 Comparison Between RPL, HP, and RP Algorithms 92

4.5 Processing Time for Three Different Applications by Targeting Different Hardware Architectures. 105

5. Conclusion 121

References 128

VITA 134

List of Tables

Table 4.1 Partitioning Netlists 61

Table 4.2 Partitioning results for Wildforce-XL using HP algorithm 69

Table 4.3 Partitioning results for Wildforce-XL1 using HP algorithm 70

Table 4.4 Partitioning results for MSP1 board using HP algorithm. 71

Table 4.5 Partitioning results for MSP2 board using HP algorithm. 72

Table 4.6 Partitioning results for SLAAC-1V board using HP algorithm 73

Table 4.7 Partitioning results for SLAAC-1P board using HP algorithm 74

Table 4.8 Partitioning results for Wildforce-XL using RP algorithm 77

Table 4.9 Partitioning results for Wildforce-XL1 using RP algorithm 78

Table 4.10 Partitioning results for MSP1 board using RP algorithm 79

Table 4.11 Partitioning results for MSP2 board using RP algorithm 80

Table 4.12 Partitioning results for SLAAC-1V board using RP algorithm 81

Table 4.13 Partitioning results for SLAAC-1P board using RP algorithm 82

Table 4.14 Partitioning results for Wildforce-XL using RPL algorithm 86

Table 4.15 Partitioning results for Wildforce-XL1 using RPL algorithm 87

Table 4.16 Partitioning results for MSP1 board using RPL algorithm. 88

Table 4.17 Partitioning results for MSP2 board using RPL algorithm. 89

Table 4.18 Partitioning results for SLAAC-1V board using RPL algorithm 90

Table 4.19 Partitioning results for SLAAC-1P board using RPL algorithm 91

Table 4.20 RPL Partitioning results for the ATR by targeting the Wildforce-XL board. 98

Table 4.21 RP Partitioning results for the ATR by targeting the Wildforce-XL board. ... 99

Table 4.22 HP Partitioning results for the ATR by targeting the Wildforce-XL 103

Table 4.23 Partitioning Results for the Wildforce-XL Board 107

Table 4.24 Partitioning Results for the Wildforce-XL1 Board 108

Table 4.25 Partitioning Results for the MSP1 Board 109

Table 4.26 Partitioning Results for the MSP2 Board 110

Table 4.27 Partitioning Results for the SLAAC-1V Board 111

Table 4.28 Partitioning Results for the SLAAC-1P Board 112

Table 4.29 Processing time for the ATR algorithm 114

Table 4.30 Processing time for different hardware architectures 120

List of Figures

Figure 1.1 Design Flow of CHAMPION 3

Figure 1.2 Back-End Flow of CHAMPION 5

Figure 1.3 General structure of FPGAS 8

Figure 1.4. Basic Wildforce-XL Block Diagram 11

Figure 1.5. Wildforce-XL Board As Used 12

Figure 1.6 MSP Board as Used. 14

Figure 1.7 SLAAC Boards as Used. 16

Figure 1.8 Hipass Filter 18

Figure 2.1 An example of KL Heuristic 24

Figure 2.2 Illustration of the gain concept for FM Heuristic. 26

Figure 2.3 The concept of the benefit function 33

Figure 2.4 Two Possible cuts 36

Figure 2.5 Multiple board configuration 38

Figure 2.6 Asyclic constraint 40

Figure 3.1 Initial Phase of HP. 44

Figure 3.2 Illustration of the Partitioning using HP 46

Figure 3.3 Pseudo-Code for the HP Algorithm 46

Figure 3.4 Recursive algorithm based on FM algorithm 48

Figure 3.5 Pseudo-Code for the RP Algorithm 49

Figure 3.6 Reduced form after level construction. 50

Figure 3.7 Pseudo-Code for the RPL Algorithm 52

Figure 3.8 Illustrative example for RPL algorithm 53

Figure 3.9 RAM access conflict 55

Figure 3.10 Illustration of the partitioning step 56

Figure 3.11 Illustration of the partitioning step 57

Figure 4.1 Partitioning Configuration File for Wildforce-XL 64

Figure 4.2 Partitioning Configuration File for Wildforce-XL1 65

Figure 4.3 Partitioning Configuration File for SLAAC-1V 66

Figure 4.4 M29 Netlist 95

Figure 4.5 Run time during the ATR partitioning process 101

Figure 4.6 Run time of the HP algorithm during the ATR partitioning process 104

Figure 4.7 Partitioning results for the Wildforce-XL 106

Figure 4.8 Image processing time for the ATR application 115

1. Introduction

In recent years, developments in the area of Field Programmable Gate Arrays (FPGAs) have allowed the concept of reconfigurable computing machines to become reality. Advances in fabrication technology have allowed multiple FPGAs of sufficient capacity to be fabricated for this purpose. FPGAs owe much of their potential to their reconfigurability. They can be reconfigured many times so that design faults can be corrected simply by reconfiguration.

Due to the short testing cycle and time to implement, FPGAs have long been used for the prototyping of ASICs. Sometimes FPGAs are used to emulate other component architectures because of their versatility. These are also used as hardware accelerators for some applications that would otherwise take longer to process on a general-purpose CPU.

At the University of Tennessee, research is currently underway to develop an automated system for mapping image processing applications in a graphical programming environment called Khoros Cantata to configurable computing hardware. It is expected that this system, called CHAMPION, will allow new applications to be implemented in much less time than is required now, since many portions of application mapping that must currently be done manually will be automated. It is also expected that the system will make the power of configurable hardware more accessible to users who lack digital hardware design experience.

1.1 CHAMPION design flow

The design flow for a CHAMPION application being implemented using multiple FPGAs is shown in Figure 1.1. The work of the author is a part of the overall research being conducted which includes the partitioning problem, netlist to structural VHDL conversion, synthesis and placement and routing, and host code automation. The design flow consists of several steps and a brief discussion of each step is provided here.

The first step is to insert the application into the Cantata workspace and convert it into an intermediate form for use by CHAMPION. Each glyph in the Cantata workspace must be replaced by its hardware equivalent. Any application in the Cantata workspace can be modeled as a directed acyclic hypergraph. In the hypergraph, nodes represent the hardware glyphs and the interconnections between nodes are represented by directed edges.

The second task is to convert the directed hypergraph model to a netlist. The netlist representation includes all information about the application hypergraph such as node size, edge width, source and destination of each node.

The third step is the data width matching and data synchronization. If data width matching is needed, a pad glyph must be inserted between two glyphs that differ in edge size. In data synchronization, data must be synchronized because all of the hardware is synchronous and data is processed every rising clock edge. If a glyph has two inputs and

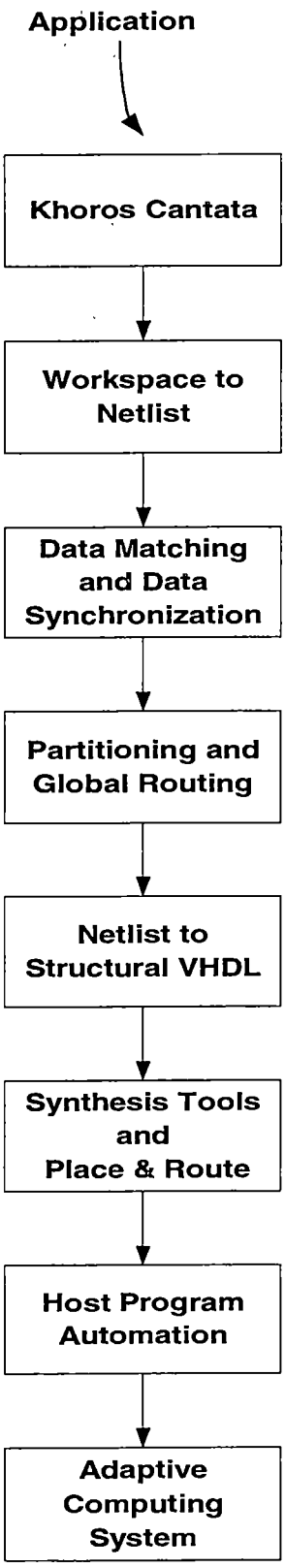


Figure 1.1 Design Flow of CHAMPION

data are not available to both inputs at the same time, then a delay buffer needs to be inserted before one of the inputs to fix the problem.

The fourth task is the partitioning problem. At this point in the design flow, the Cantata workspace has been converted into a directed hardware hypergraph. The netlist representation of the directed hardware hypergraph has the necessary information to specify the glyphs and the connections between them. The hardware graph is a set of vertices representing the hardware resources and a set of directed edges representing the connections between them. Each vertex has a number representing the size of the hardware resources available in the vertex (measured in CLBs for Xilinx FPGAs). Each edge has a number representing the width of the connections. If the application graph does not fit in a single processing element (PE) or FPGA, then the hardware graph must be partitioned into sub-graphs. In most cases, the size of the entire graph is larger than the size of a single PE. Therefore, there exists a need for a multiple partitioning algorithm. This step will be explained in more detail since the research of the author involves the development of multiple partitioning algorithms, which meet the constraints imposed by the hardware architecture.

In the fifth task, each sub-netlist resulting from the partitioning step, as shown in Figure 1.2, must be converted to a structural VHDL file representing the hardware resources desired for each FPGA. A PERL script file written by the author is used to generate the structural VHDL file. The script file identifies the glyphs used in the sub-netlist, the connections between glyphs, and the connections between glyphs and the

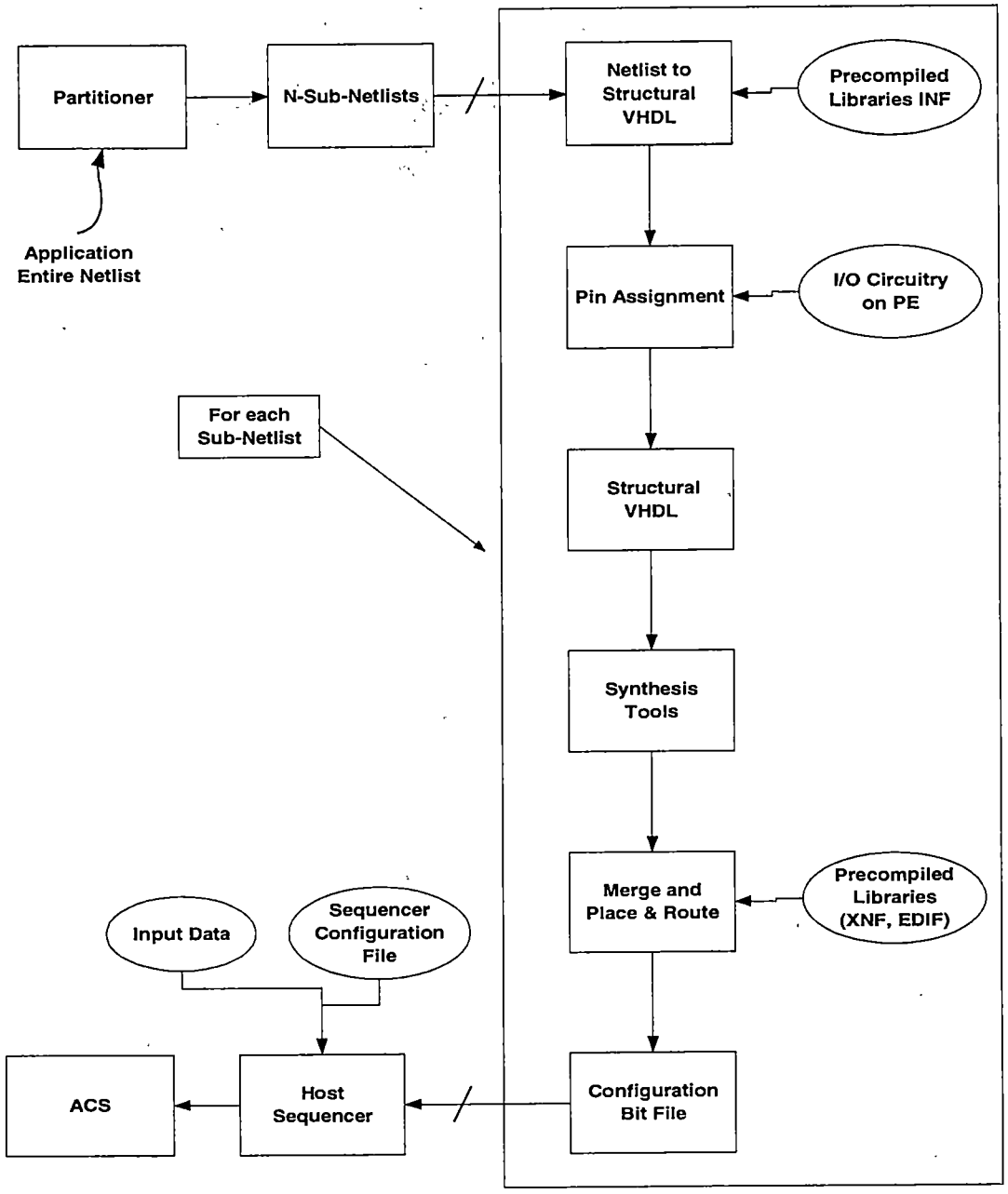


Figure 1.2 Back-End Flow of CHAMPION

other FPGAs. Furthermore, the script file accesses precompiled VHDL files to extract the port map information for each glyph. There are other VHDL files that specify the board architecture, internal interface logic for each FPGA, and global signals on the board. The script file accesses these files for pin assignment.

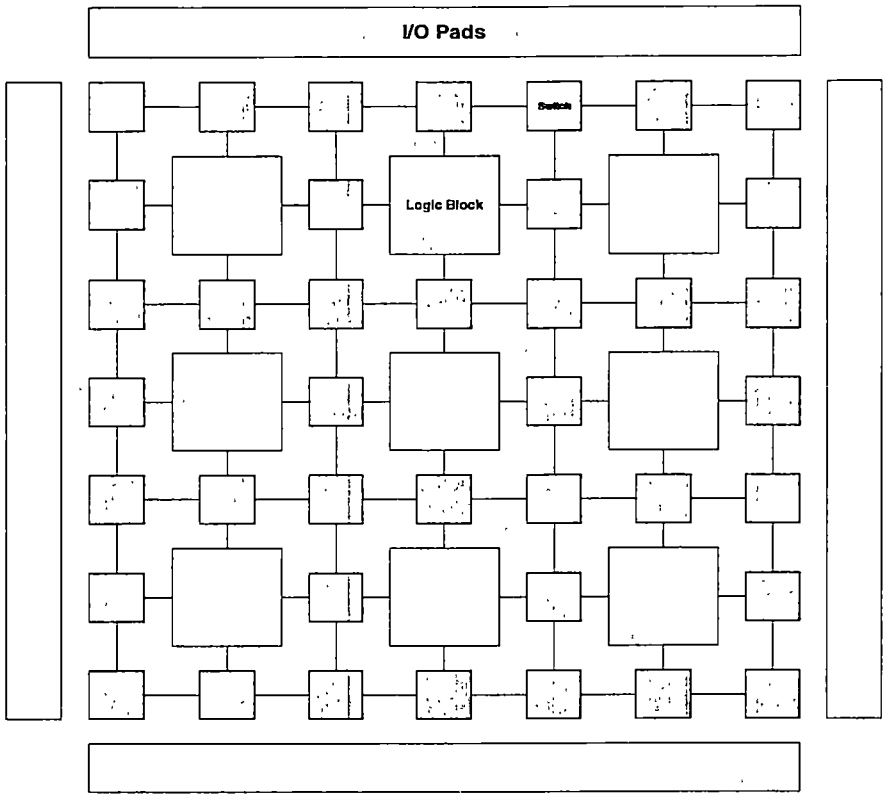
The sixth task deals with the physical design phase. Once the structural VHDL for each Sub-netlist is complete, it is necessary to create the programming files to actually implement the desired hardware in the FPGAs on the ACS board. All of the behavioral information is in the pre-synthesized files for each glyph. The synthesis tools to generate the synthesized file for a sub-netlist, which is required to configure each PE, can access the pre-synthesized files for each glyph. The place and route software tools are then used to map the hardware description in the sub-netlist file to specific resources available in each PE. This results in a programming bit file, which specifies the configuration of all the function generators and storage units in the CLBs, as well as the configuration of all of the programmable interconnections in the PE. The bit file can then be downloaded to the PE to specify its behavior. One programming bit file is needed for every configuration of each PE.

The final step is the host code generation. A program written in C takes care of certain functions necessary to enable the ACS (e.g. Wildforce board) to be used. A set of function calls to communicate with the board is provided by the manufacturer of the ACS. These function calls must be used to create the host program. This host program must initialize the ACS board and download the programming bit files for each PE. The host program reads image files from the workstation hard drive to be used as input to the

application, and writes the application results back to the hard drive. The automated host code uses a configuration file produced by a PERL script, which accesses the resulting sub-netlists and extracts the configuration data. This configuration file is used by the host code to determine the number of configurations, the name and location for each programming bit file, if a specific PE needs to access the SRAM, and where to write the result after each configuration. If multiple configurations of the ACS board are needed, then the data resulting from each board configuration is written to the hard drive and supplied to the next configuration. After each board configuration, the user can look at the resulting data from each board configuration and compare it with the expected result to detect any error that may occur during implementation. In addition to the function calls provided by the manufacturer, the automated host code uses a dynamic data structure that grows or shrinks since different applications may have a different number of configurations.

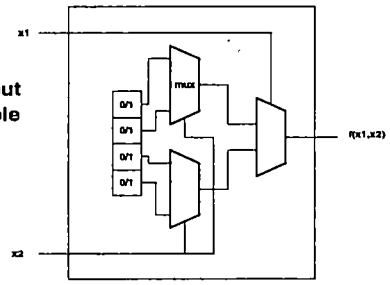
1.2 Field Programmable Gate Arrays

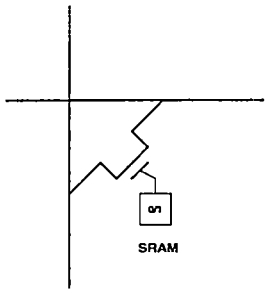
A field programmable gate array (FPGA) is a programmable logic device that supports implementation of a logic circuit containing thousands of gates and interconnections. FPGAs are quite different from PLDs (programmable logic devices) and CPLDs (complex programmable logic devices) because FPGAs do not contain AND or OR planes. Instead, FPGAs provide programmable logic blocks for implementing the required logic functions [28]. Figure 1.3 shows a general structural of a FPGA [14]. In a FPGA, the logic blocks are arranged in a two-dimensional array, and the interconnection



(a) General structure of FPGAs

(b) Two input look-up table (LUT)





(c) Pass-transistor switch in FPGAs

Figure 1.3 General structure of FPGAS

wires are organized as horizontal and vertical routing channels between rows and columns of logic blocks. The routing channels contain wires and programmable switches that allow the logic blocks to be connected in many ways. Each programmable logic block in a FPGA typically has a small number of inputs (say four) and one output. The most commonly used logic is a lookup table (LUT), which contains storage cells that are used to implement a small logic function. The storage cell holds a single logic value, either 0 or 1. Figure 1.3b shows the structure of a small logic block capable of implementing any logic function of two variables. In FPGAs, a switch can be implemented by using an NMOS transistor, with its gate controlled by an SRAM cell. This type of switch is known as a pass-transistor switch. The NMOS switch is turned off if a 0 is stored in the SRAM cell. But if a 1 is stored in the SRAM cell, then the NMOS switch is turned on. In this case, the NMOS switch forms the connection between the two wires attached to its source and drain terminal. Figure 1.3c shows the structure of the NMOS pass-transistor.

FPGAs are high density devices, which are commercially available at low cost. The programmability features and the short production times of these devices enable changes to be incorporated immediately. These features make FPGAs suitable for prototyping applications, and implementation of applications formerly targeted to ASICs. The main disadvantage of FPGAs is the lower speed of operation. The programmable switches and the associated programming circuitry require a large a mount of the chip area. The switches have significant resistance and capacitance, which account for the low speed of operation [3].

1.3 Hardware Architecture

In this section we will describe the hardware architectures of the three different adaptive computing systems (ACS), which were chosen to implement the CHAMPION applications.

1.3.1 Wildforce-XL Board

The Wildforce-XL board from Annapolis Micro Systems was chosen as the first architecture used in the CHAMPION project. The Wildforce-XL board uses Xilinx XC4000 series FPGAs. It is a PCI-bus card, which uses five Xilinx XC4000XL FPGAs for processing elements. The specific version of the board used had one XC4036XL FPGA and four XC4013XL FPGAs available for processing. Figure 1.4 shows the basic Wildforce-XL block diagram.

Annapolis Microsystems refers to the FPGAs on the Wildforce board as processing elements (PEs). The XC4036XL FPGA is called a control processing element and given the designation CPE0. It differs from the other FPGAs in that it is larger, and also in that it has control lines available for various resources on the board, such as the external I/O interface and crossbar configuration register, that are not available to the other FPGAs. The four XC4013XL FPGAs are given the designations PE1, PE2, PE3, and PE4. These four processing elements are connected together in a linear array by a 36-bit systolic bus. All five FPGAs can be connected by the 36-bit crossbar, which selectively allows

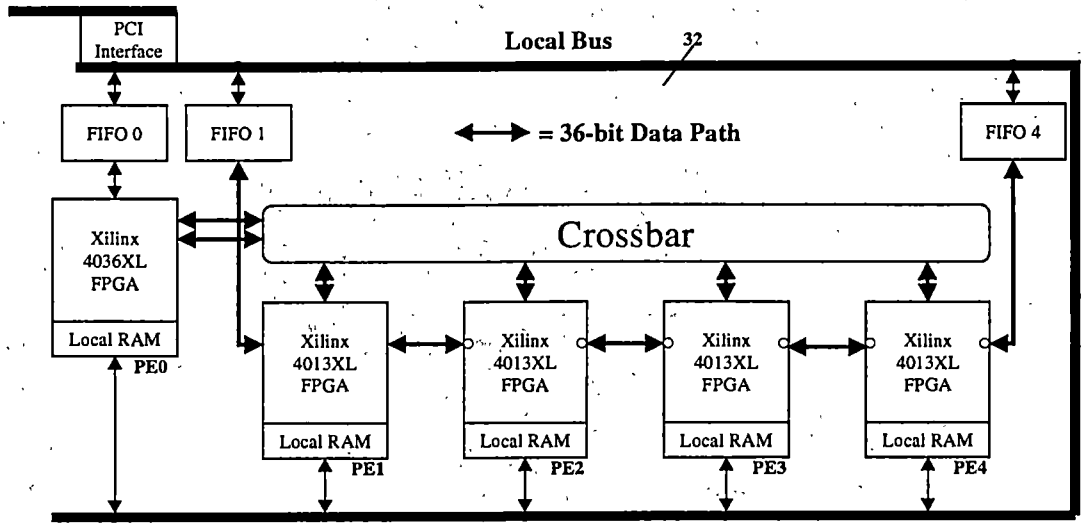


Figure 1.4. Basic Wildforce-XL Block Diagram [4].

connections between any of the processing elements. CPE0 can connect to the other processing elements only through the crossbar.

Each FPGA on the board has a small daughter-board associated with it, which can be populated with memory or a Digital Signal Processing (DSP) chip. Each of the FPGAs on the board used in this project had 32 KB of 32-bit SRAM on its daughter-board. Each daughter-board has a dual-port memory controller such that both the FPGA and the host computer can access the SRAM. The motherboard also contains a PCI interface for communicating with the host computer, and several FIFO registers to facilitate data transfer across the PCI bus.

Since there are many resourses available on the Wildforce board and many configurations of the crossbar and other components, it was decided to use a constrained configuration of the board for the automatic implementations. This reduces the problem

complexity to a more manageable level. The constrained configuration of the board used in this project does not use any of the FIFOs. All communication with the host is done through the SRAM associated with each processing element. The crossbar is used only to provide a 36-bit path from CPE0 to PE1. The connections between processing elements are normally bidirectional. For the constrained implementation, however, it was decided that the direction of all connections between processing elements would be fixed so that all signals would pass in one direction only. The board topology became a linear array, with all signals starting in CPE0 passing to PE1. No signals can run from PE1 back to CPE0. Similarly, all signals from PE1 run to PE2, with no signals allowed to pass back from PE2 to PE1. A diagram showing the configuration of the Wildforce-XL board as used in this project is shown in Figure 1.5.

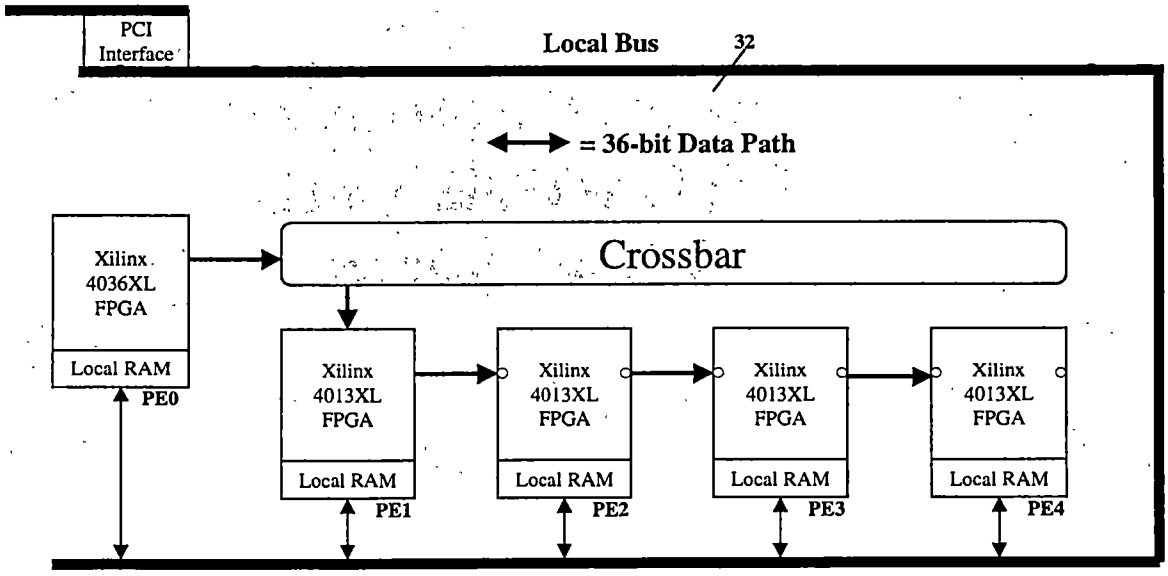


Figure 1.5. Wildforce-XL Board As Used [4].

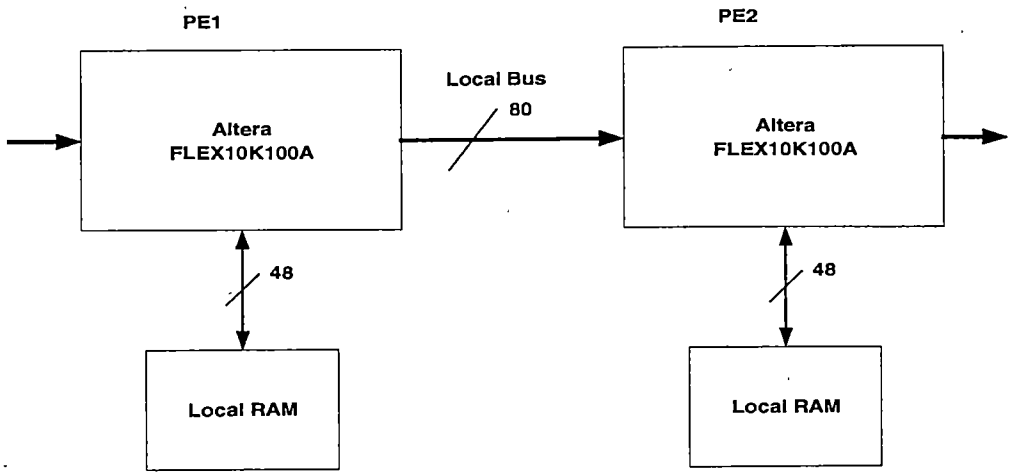
1.3.2 MSP Board

The MSP board uses Altera FLEX10K series FPGAs. It is a PCI-bus card, which uses two FLEX10K100A FPGAs for processing elements. Since there are many resourses available on the MSP board and many configurations of the crossbar and other components, it was decided to use a constrained configuration of the board for the automatic implementations. A diagram showing the configuration of the MSP board as used in this project is shown in Figure 1.6. The two FLEX10K100A FPGAs are given the designations PE1 and PE2. These two processing elements are connected together in a linear array by a 80-bit systolic bus. The board topology became a linear array. The capacity of each FPGA is very large, where each FPGA comes with 100K logic gates.

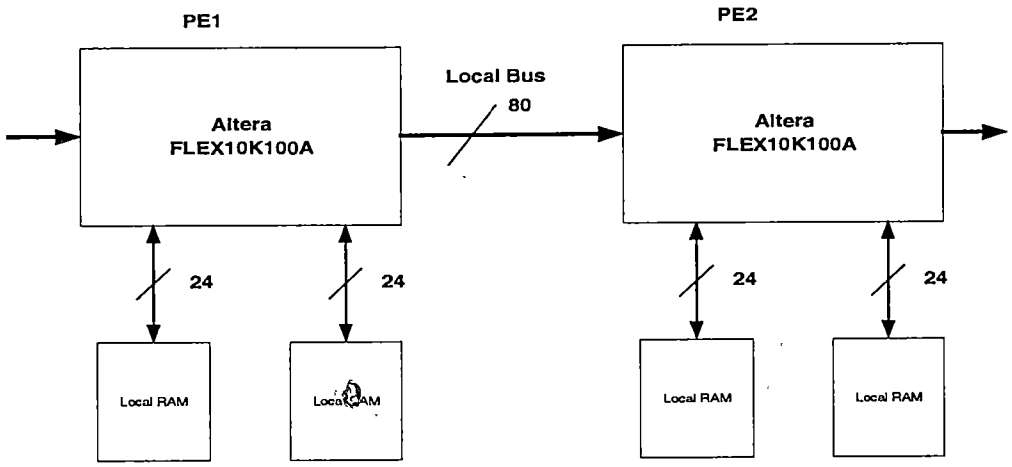
Each of the FPGAs on the MSP board used in this project has 512 KB SRAM on its daughter-board. The 512KB RAM can be organized as one bank of 512kX48 bits or two banks of 512kX24 bits Each daughter-board has a dual-port memory controller such that both the FPGA and the host computer can access the SRAM.

1.3.3 SLAAC Board

In this research project we consider two versions of the SLAAC board, the SLAAC-1V and the SLAAC-1P. The SLAAC-1V board uses Virtex series FPGAs while the SLAAC-1P uses Xilinx XC40150 FPGAs. The SLAAC board has many resources. Similar to the Wildforce-XL and MSP boards, it was decided to use a constrained configuration of the SLAAC board for the automatic implementations.



A. RAM organized as one Bank of 512kX48 bits



B. RAM organized as two Banks of 512kX24 bits

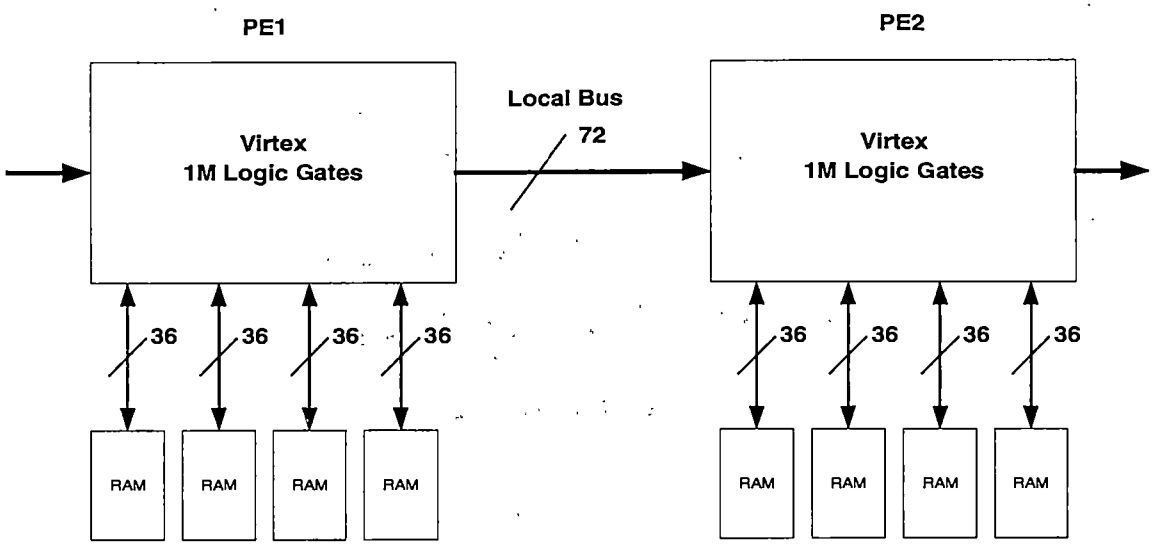
Figure 1.6 MSP Board as Used.

A diagram showing the configuration of the SLAAC board as used in this project is shown in Figure 1.7. The SLAAC-1V board uses very large FPGAs where each FPGA has 1M logic gates. The SLAAC-1P board uses FPGAs where each FPGA has 750K logic gates. For both SLAAC-1V and SLAAC-1P boards, the two FPGAs are given the designations PE1 and PE2. These two processing elements are connected together in a linear array by a 72-bit systolic bus.

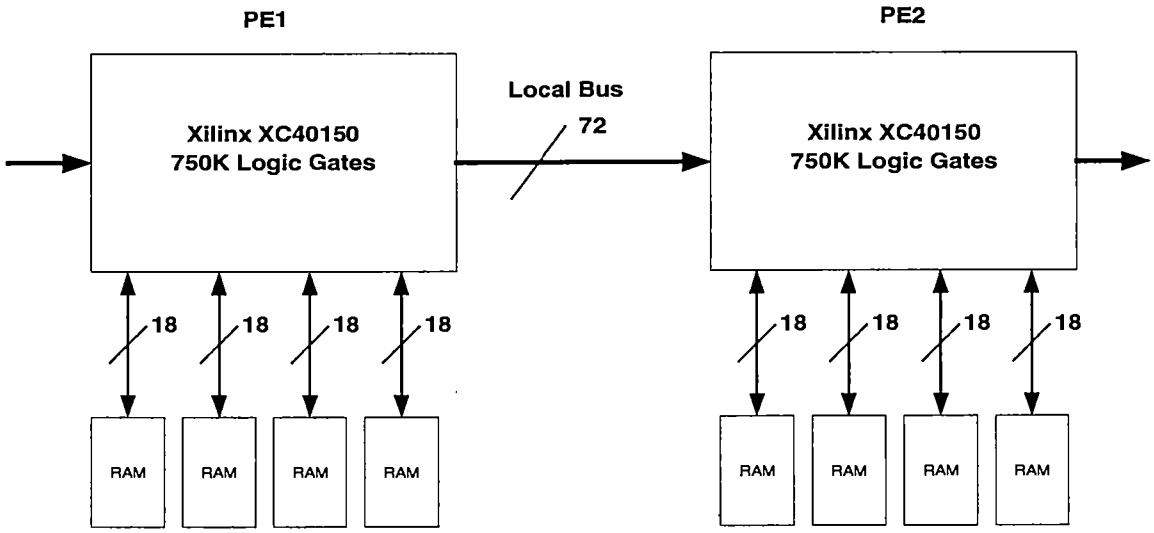
Each of the FPGAs on the SLAAC-1V board used in this project has four 256KX36 bits SRAM on its daughter-board. In the SLAAC-1P, each FPGA uses four 256KX18 bits SRAM on its daughter-board. Each daughter-board has a dual-port memory controller such that both the FPGA and the host computer can access the SRAM.

1.4 Khoros Cantata

Khoros is a software system from Khoral Research Incorporated (KRI). Khoros has a set of toolboxes containing over 300 operators [4]. Many functions can be implemented using these operators such as arithmetic operations, image and signal processing functions, and data visualization. The Khoros operators can be run as stand alone programs from the command line, or as functions called by C code. Cantata is a graphical programming environment used to run Khoros functions. In Cantata, the user can draw a graphical representation of an application and run it. Each function in the Khoros toolboxes is represented in the Cantata workspace by a small icon called a glyph.



A. SLAAC-1V Board



A. SLAAC-1P Board

Figure 1.7 SLAAC Boards as Used.

Each glyph has an input corresponding to each of the possible inputs to the function and output terminals for each of the outputs. Furthermore, each glyph has a pane, which is a set of interface objects that allow the user to set options for a function. In Cantata, the user does not need to be concerned about the type of input data. Cantata finds out if a data conversion between types is necessary and takes care of it.

As mentioned before, the objectives of CHAMPION are to parse a netlist of Khoros glyphs and automatically map the design captured by the workspace into a form that can be executed on a multi-FPGA platform. Therefore, the development of an equivalent hardware library was necessary to complete the mapping. The Cantata glyphs used in CHAMPION are called hardware equivalent glyphs and are written in fixed-point C. These glyphs, chosen to be used in CHAMPION, have to operate in Cantata in a manner equivalent to the way the hardware glyphs would operate. These equivalent hardware glyphs are collected together into a library, which characterizes the glyphs by size, delay, and I/O count [5]. To aid the mapping procedure, an attempt was made to make the equivalent hardware glyphs function as similar as possible to the traditional Khoros glyphs. By doing so, the mapping procedure will yield an almost one to one correspondence between the Khoros workspace design and the mapped hardware implementation. Figure 1.8 shows an implementation of a Hipass Filter using the CHAMPION equivalent hardware glyphs. A hardware glyph had to be developed for every hardware-equivalent glyph used in CHAMPION. These hardware glyphs were developed in VHDL.

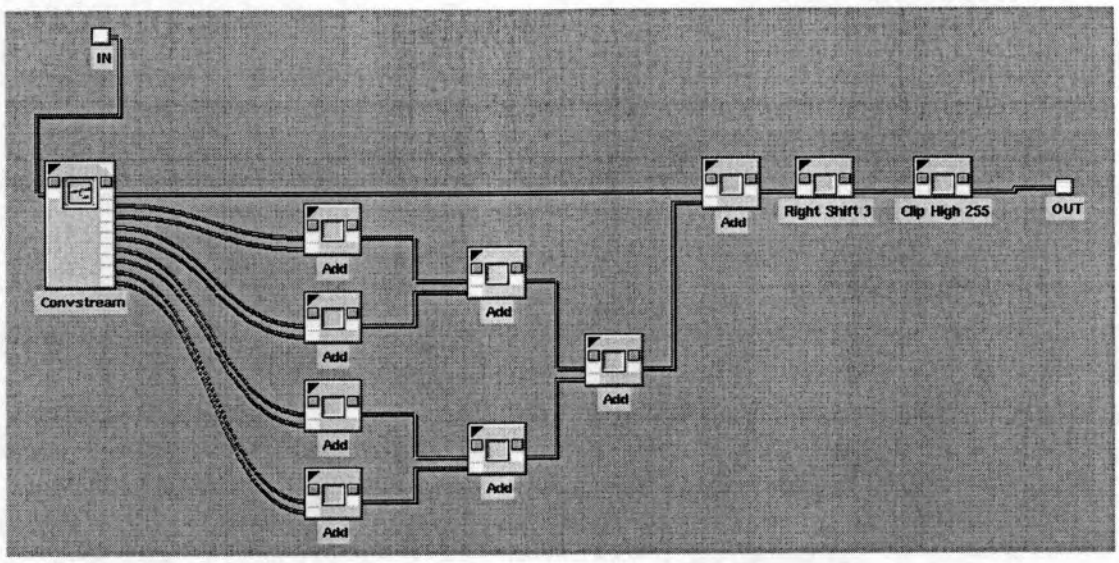


Figure 1.8 Hipass Filter [4]

The hardware glyphs were developed in a parameterized manner, meaning that various characteristics of the glyphs, such as the number of bits in the input, were specified as generic that could be changed as needed. This allows the hardware to be used in an efficient way. Each specific version of each hardware glyph was synthesized and stored in a specific format. Since Xilinx FPGAs were used on the Wildforce board, the hardware glyphs were synthesized into Xilinx Netlist Format (XNF). Glyphs needed by CHAMPION applications are pre-synthesized to speed up the mapping process.

1.5 Placement and Routing

In this section, the Placement and Routing step is given some attention. Placement takes the logic functions formed by technology mapping and assigns them to specific logic blocks in the FPGA. This process can have a large impact on the capacity and performance of the FPGA. Specifically, routing between distant points in a FPGA

requires a significant amount of routing resources. Thus, this path will be much slower, and use many resources. For this reason, the primary goal of the placement process is to minimize the length of signal wires in the FPGA. In this case, logic blocks that communicate with one another must be placed together as close as possible [3]. Placement is a complex process since logic circuits tend to have a significant amount of connectivity with many different functions communicating together. In this case, many functions may want to be placed together to minimize the number and length of signal wires. Because of this, the placement tools must find out which logic blocks are most important to place together, and minimize the total wiring in the system.

There exist many software tools for performing the placement step for FPGAs. The most common technique used in the industry is simulated annealing. Simulated annealing solves the optimization problem by using a cost function. A cost function could be the total wire length in the design. Once a cost function is defined, the placement tool picks a random starting point. The algorithm then repeatedly applies optimization steps to find a new solution with a lower cost than the current solution.

The routing process for FPGAs is the process of finding out exactly which routing resources will be used to wire the communication signals. Since FPGAs have prefabricated routing channels, a FPGA router must work within the framework of the architecture resources. In deciding which channels and wires to use, and how to connect through the switchboxes, the router must ensure that there are enough resources to carry the signal in the chosen routing regions, as well as leaving enough resources to route the

other signals in the system. The most common technique used for performing this routing step is presented in [9]. The algorithm divides the routing process into global and detailed routing. In global routing, the algorithm decides which routing regions the signal will move through. Thus, it will select the routing channels used to carry a signal, as well as the switchboxes it will move through. The detailed router decides which specific wires to use to carry the signal. It finds a connected series of wires, in the channels and switchboxes chosen by the global router, which connects from the source to all the destinations. The algorithm avoids congestion inside a channel, making sure that all signals can be routed successfully, as well as minimizing wire length and capacitance on the path.

For successful placement and routing, a full capacity utilization of a FPGA is avoided. Therefore, a 90% utilization of one PE is a typical value to perform placement and routing successfully.

2. Research Motivation and Background

As mentioned in the previous chapter, there exists a need for a k-way partitioning algorithm to subdivide the CHAMPION netlist into multiple sub-netlists. This is done when a large application cannot fit in a single device. Each resulting sub-netlist must be written in a form suitable for further processing. The resulting sub-netlist is the input required for structural VHDL conversion. An application, for our purposes, is equivalent to a graph, and more specifically a directed acyclic graph (DAG). A single device is equivalent to a single processing element (PE) or a single FPGA. Both expressions are used in our project. The resulting sub-netlist is considered a partition. A gate or a macro is the same as a node and a net is used to refer to an edge.

2.1 Partitioning Methods for Multi-PE System

In order for a circuit to be implemented across multiple PEs, the circuit must be cut into pieces such that multiple constraints are met. The following sections review several existing algorithms developed for multi-PE partitioning.

2.1.1 Bi-Partitioning Methods

In bi-partitioning, the method begins with a graph G with weighted edges E and weighted nodes N. The graph is split randomly into two halves A and B. Nodes are then moved across A and B to find a valid partitioning result with a minimum cut set. The cut

set is defined to be the sum of all weighted edges interconnecting nodes in both subsets A and B. The overall goal in circuit partitioning is to minimize the number of nets that are cut. The bi-partitioning method can be used with multi-PE systems by repeatedly applying the technique until each partition meets the PE constraints. However, this repeated application locks in an initial solution that may be poor in a multi-PE system. Also, the basic technique does not consider PE to PE interconnect limitations even though it does attempt to minimize interconnect usage [10]. Below, we discuss a few bi-partitioning algorithms, which have contributed to the development of partitioning algorithms.

A. KL Heuristic

In 1970, Kernighan and Lin introduced an iterative improvement algorithm that has become known as the foundation of partitioning algorithms [2]. The algorithm is a bi-partitioning algorithm that begins with an initial partition A and B and iterates to improve the cut set size. The algorithm uses a pair-swap structure and proceeds in a series of passes. During each pass, every node is moved exactly once, either from A to B or from B to A. At the beginning of the pass, each node is unlocked, meaning that it is free to be moved across A and B. A node becomes locked after swapping. The KL algorithm iteratively swaps the pair of unlocked nodes **a** and **b** with the highest gain, where the gain of swapping $a \in A$ with $b \in B$ is given by [3]:

$$gain(a,b) = D_a + D_b - 2c_{ab} \qquad (1)$$

$$D_a = E_a - I_a$$

$$D_b = E_b - I_b$$

where

E_a= number of edges incident to node **a** that connect to a node in subset B

E_a= number of edges incident to node **b** that connect to a node in subset A

I_a= number of edges incident to node **a** that remain internal to subset A

I_b= number of edges incident to node **b** that remain internal to subset B

c_{ab}= sum of the weights of the edges that connect the nodes **a** and **b**

The swapping process is iterated until all nodes become locked, and the lowest cost bisection observed during the pass is returned. Another pass is then executed by using this partition as its initial solution. The algorithm terminates when a pass fails to find a solution with lower cost than its starting solution. Figure 2.1 shows an example of the KL algorithm [6]. A pass of KL is implemented in $O(n^2 \log n)$. The KL algorithm was modified by Schweikert and Kernighan to handle multi-pin nets [7].

B. FM Heuristic

In 1982, Fiduccia and Mattheyess presented a KL-inspired algorithm that takes into consideration multi-pin nets and reduces the running time of a pass. The main difference between KL and FM is the neighborhood structure. In the FM algorithm, a single node is moved either from A to B or from B to A. Therefore, the FM algorithm was designed to handle imbalance.

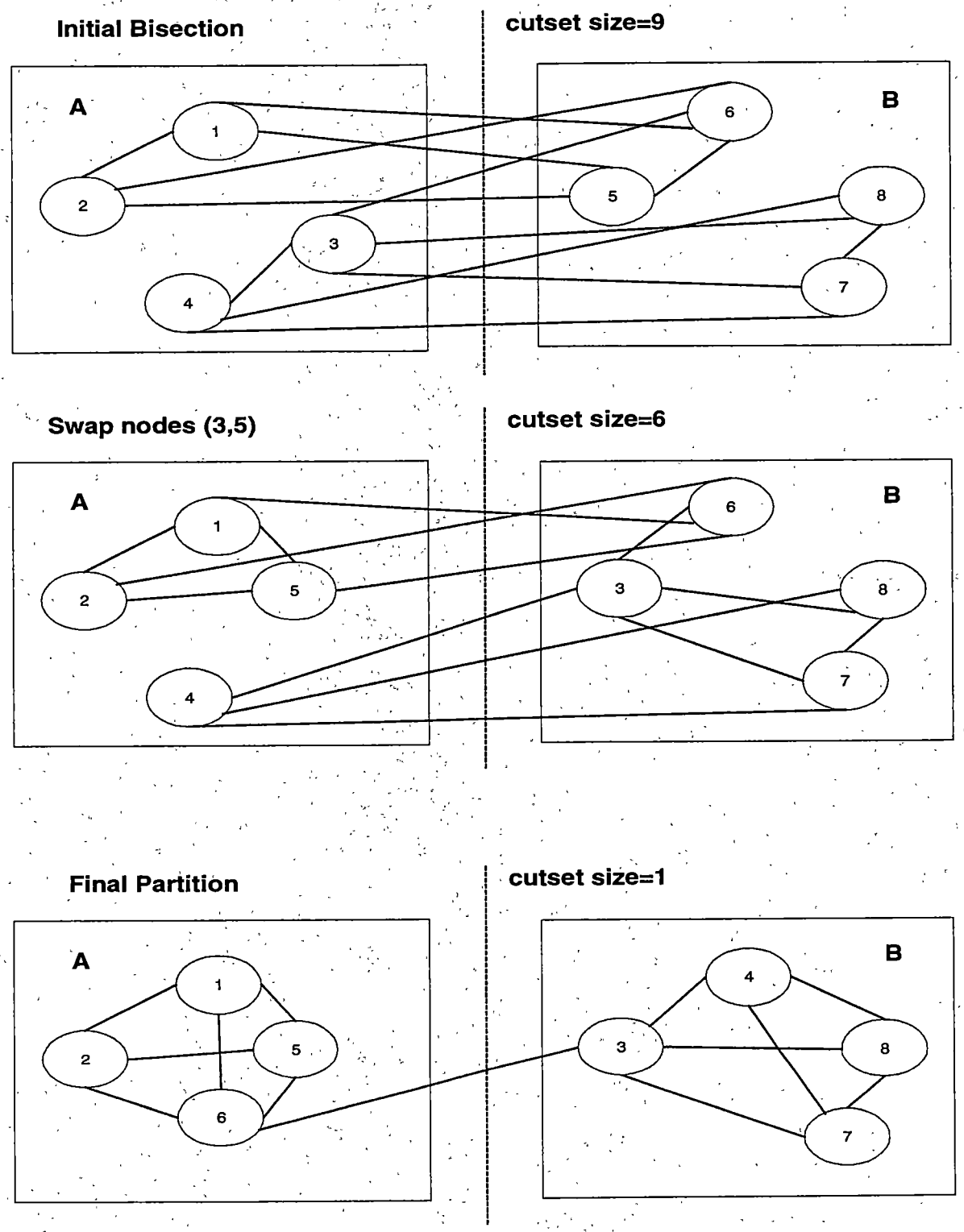


Figure 2.1 An example of KL Heuristic

Similar to the KL heuristic, the gain due to the movement of a single node from one block to another is computed instead of the gain due to the swapping of two nodes [3]. Moving individual nodes can result in unbalanced blocks. To avoid having all nodes migrate to one block, a balance criterion is maintained by the definition:

$$r|V| - w_{max} \leq |A| \leq r|V| + w_{max} \qquad (2)$$

where

$$|A| + |B| = |V|$$

$$r = \frac{|A|}{|A| + |B|}$$

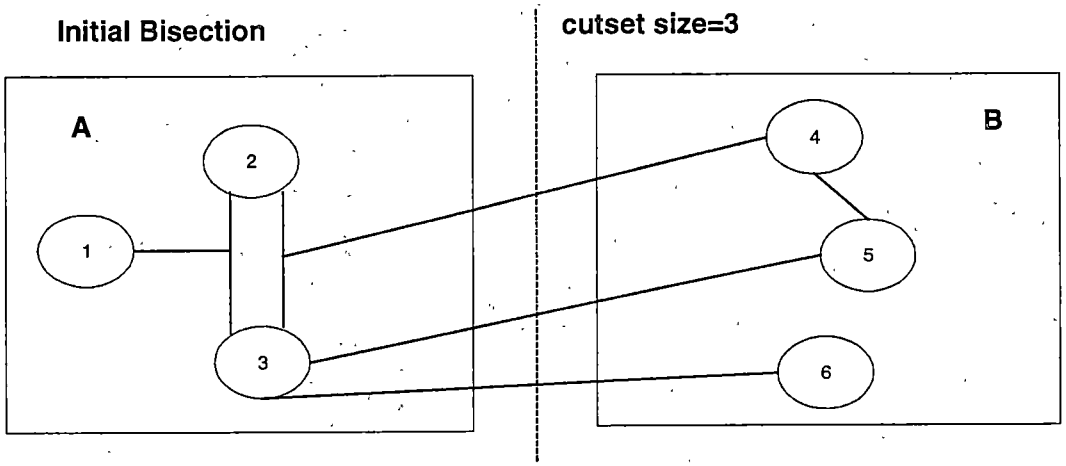
w_{max} = the node with the maximum weight

$|A|$ =size of the partitioned block A

$|B|$ =size of the partitioned block B

$$0 < r < 1$$

The initials gains are computed for all free nodes. The gain, g(i), of a node is the number of nets that would result in a decrease or increase in the cutset if the node were moved to the opposite block [8]. A single node, called the *best* node, is selected for moving based on its gain as well as on the balance criterion. Figure 2.2 shows the concept of node gain for the FM heuristic [3]. In this example, moving node 1 from A to B would increase the cutset by 1, therefore, g(1)= -1. Moving node 2 from A to B would decrease the cutset by 1. Moving node 3 from A to B would increase the cutset by 1. However,



Initial Gains

Node	Gain
1	-1
2	1
3	-1
4	0
5	0
6	1

Figure 2.2 Illustration of the gain concept for FM Heuristic.

there will be no change in the cutset if the nodes (4,5) are moved to their complementary blocks, therefore g(4)= 0 and g(5)= 0. If node 6 were moved from B to A, the cutset would decrease by 1, therefore g(6)= 1. A negative gain might be accepted during a pass to allow the algorithm to climb out of a local minimum. After each move, the *best* node is locked in its new block for the remainder of the pass and the gains of the affected nodes are updated. The algorithm considers all nodes for movement. Afterwards, the best partition encountered during the pass is taken as initial solution to the following pass.

C. Simulated Annealing (SA)

Simulated Annealing belongs to the class of non-deterministic algorithms. The heuristic was first introduced by Kirkpatrick, Gelatt and Vecchi in 1983 [12]. The algorithm has been applied to almost all known CAD problems, including partitioning [3]. SA begins with an initial partition A and B and iterates to minimize the cutset while maintaining balance between partitions. A pair of nodes is moved across the cut and the new partition after the movement is evaluated. If the cut improves, then the move is accepted. Otherwise, the move is accepted with a certain probability. This allows the algorithm to climb out of a local minimum.

The SA is analogous to the annealing process, in which the process starts at a high temperature. This corresponds to a large number of moves being accepted in an attempt to minimize the cost function. The process continues to lower the temperature until a freezing point is reached. The lower the temperature, fewer moves are accepted that do

not improve upon the cost function. Even though the SA algorithm is noted for achieving excellent partitioning results, the use of it for multi-PE partitioning is not feasible because of the excessive computation time [13].

2.1.2 Multiway Partitioning Methods

In the previous section, we mentioned that the bi-partitioning method can be used with multi-PE systems by repeatedly applying the technique until each partition meets the PE constraints. The bi-partitioning approach has been extended by some researchers to a k-way approach because of the increased complexity of VLSI designs as well as applications requiring a fixed number of partitions. In this section, we discuss a few multi-way partitioning algorithms.

A. Multiple-way Algorithm

In 1989, Sanchis extended the FM algorithm to multi-way partitioning [16]. The detailed explanation of this approach has led to wide use of the algorithm in industry. The algorithm begins with k blocks and iterates to minimize the total number of interconnections between the blocks. The algorithm uses the level gain concept, mentioned in the previous section, to compute the gain of a moving node. Sanchis extended the ith level gain of a node to include the gain of moving a node from the originating block to all other possible blocks. A pass consists of moving free nodes that have the highest gain that satisfy the balance requirement to its block. Moves continue

until all nodes are locked and the resulting partition becomes the initial partition for the next pass. The algorithm uses a specified number of blocks k to start the partitioning process.

B. MP2 Algorithm

The MP2 algorithm is a k-way extension of the FM heuristic. It is an iterative improvement algorithm that begins with a random initial partition of k blocks, where k is specified [29]. The capacity constraint and the I/O limitations of each PE device must be satisfied during a pass. For moving a free node from the originating block to another block, a benefit for the node is calculated with respect to an incident net. The algorithm uses a look-ahead technique by using a secondary benefit for each free node. During a pass, the I/O and the capacity constraints of the destination block are checked for violation. If no violations occur, the node is moved and the total number of interconnections between the k blocks is recorded. After each pass, the resulting partition is checked to make sure no block constraints are violated. If the constraints are satisfied, the algorithm returns the current partition as the solution. The algorithm requires that the number of the total partitions k must be specified before starting the partitioning process.

A. PROP Algorithm

The PROP algorithm makes use of a recursive paradigm using functional replication [15]. For a given netlist, the algorithm applies a bi-partitioning procedure to extract the first feasible partition, and then repeats the process on the remainder netlist until the

remainder fits on one PE. The bi-partitioning strategy can be viewed as an extreme case of asymmetrical recursive bi-partitioning. The main advantage of this strategy is the immediate possibility of evaluating at least one of the subsets produced in each bi-partitioning stage. Another advantage is that the algorithm does not require the total number of partitions k to be specified before running the partitioning process. In our proposed methods, we will make use of this strategy and develop the algorithm to meet the constraints injected by CHAMPION applications. Therefore, the details of this approach will be given in the next chapter.

D. Hierarchical Multiway Partitioning Strategy (HPS)

In 1997, Stanley developed a new k-way partitioning algorithm to incorporate the architecture configuration of a hardware emulator into the partitioning process [8]. The HPS considers the interconnect limitations of the hardware and the upper limit on the number of partitions during the partitioning process. The algorithm was initially developed using a random selection of nodes or clusters to move. All clusters or nodes are initially moved into a virtual partition VP and all blocks are initially empty. An initial node or cluster with an incident external I/O is randomly selected from VP and moved into the first partition, P0. The algorithm continues to move nodes or clusters with the highest benefit into P0 until the capacity or external I/O constraint of the current partition is violated. Once a constraint is violated, the algorithm begins an evaluation step. If the interconnect constraint is violated, the algorithm "rolls back" or reverses the moves until the constraint is met. Similar to the PROP algorithm, the HPS does not require the total

number of partitions k to be specified before running the partitioning process. In addition, the HPS strategy evaluates the partition produced in each stage. In our proposed methods, we will make use of this strategy and develop the algorithm to meet the CHAMPION application constraints. More details will be explored in the next chapter.

In addition to these approaches, several other partitioning algorithms were developed to handle multiple-PE systems [17, 18,19,20,21]. Since no general model exists to handle different partitioning problems, most of the recent approaches were developed to target specific hardware structures. To the best of our knowledge, no approach exists to handle the CHAMPION netlist.

2.1.3 Benefit Function

In this section, the definition of the benefit function is given some attention. The benefit of a node or cluster is determined by its connectivity to the current partition [8]. Stanley defined the benefit function for a cluster of nodes. In this work, we modify this definition slightly to consider nodes instead of clusters. It is another way to find the best node that can be moved across a cut. A benefit is calculated for each node n based on the following:

$$ben(n) = \mathrm{int}(n) - ext(n) \qquad (3)$$

where

int(n) = The number of nets incident to node *n* that will no longer be incident to the current partition after the move.

ext(n) = The number of nets incident to node *n* that will become incident to the current partition after the move.

Figure 2.3 shows a simple example for calculating the benefit of nodes. If the node **n2** is moved from partition A to partition B, by using equation 3, then ben(n2)=1-1=0. In this case, **net2** is removed from the cutset but an additional net is added. Therefore, the benefit of moving node **n2** from A to B is 0. If the node **n3** is moved from partition A to partition B, then ben(n3)=1-3=-2. Moving the node **n3** from A to B results in a negative benefit since two additional nets are added to the cutset. Comparing the computed benefits for **n2** and **n3**, the node with the highest benefit is **n2**. Therefore, we consider the node **n2** to be moved from partition A to partition B. We will make use of the benefit function to calculate the best node in our proposed approaches.

2.3 Partitioning Constraints

The k-way partitioning algorithm must meet the constraints of our hardware architecture used to implement an application. In order to assure the results generated meet these constraints, it is necessary for the algorithm to have knowledge of the hardware architecture. Most existing partitioning algorithms for multiple FPGA systems use the following constraints:

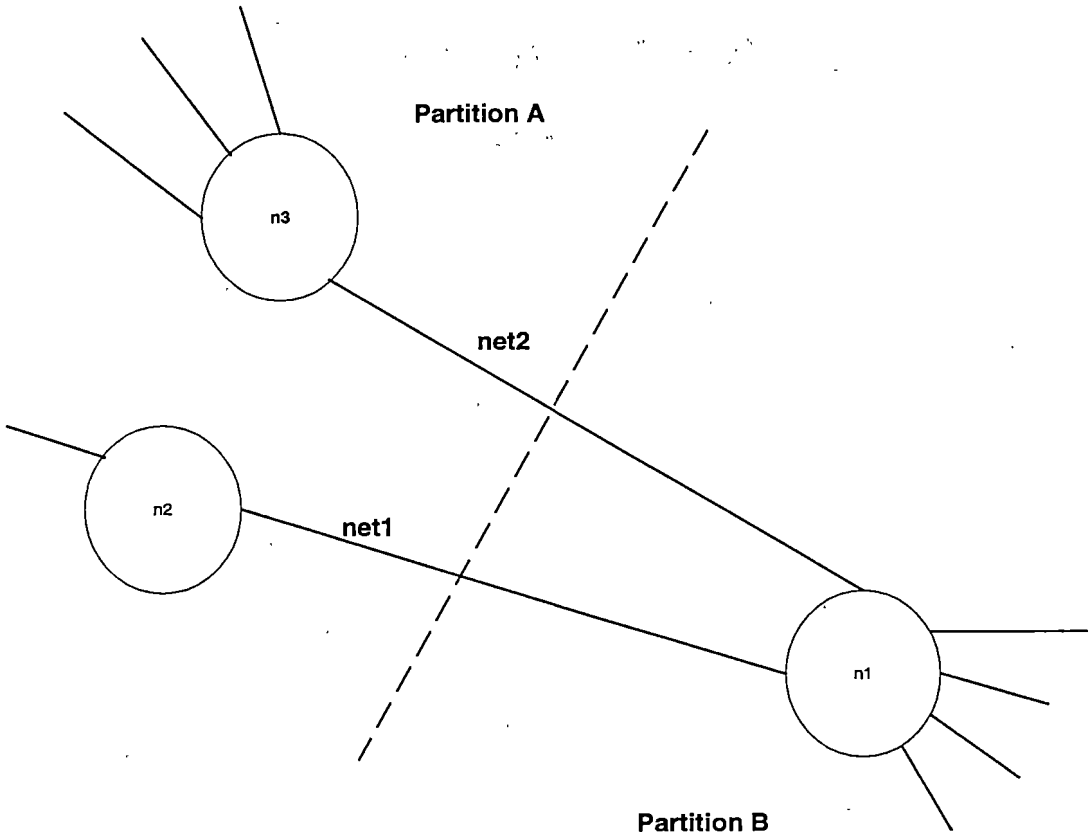


Figure 2.3 The concept of the benefit function [8]

- Capacity per partition
- Number of I/O pins per partition

Targeting a particular hardware architecture injects additional constraints that must be considered to generate a resulting partition that can be successfully placed and routed.

The objective of this research is to develop three different k-way partitioning algorithms based on the target architecture of the boards used in this research project. The partitioning strategy is based on the following constraints followed by a brief discussion of each constraint:

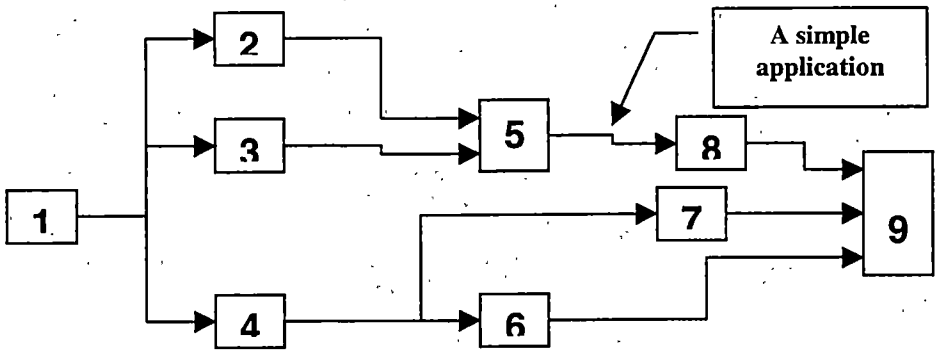
1. Capacity per partition
2. Number of I/O pins per partition
3. Each partition can only have one RAM access module
4. Input module and output module must be placed in the first partition and in the last partition.
5. Temporal partitioning constraint: For multiple board configurations, storage of intermediate results between board configurations is needed.
6. Maintaining the acyclic constraint so that all edges point the same way (from left to right).

The first two constraints are used to meet the limitations of a single PE device. The number of interconnections between partitions becomes an issue due to the limited

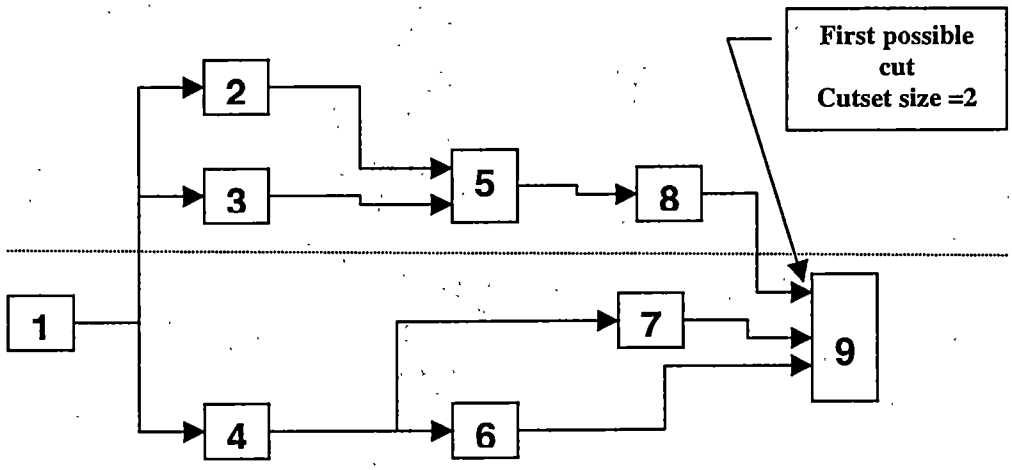
number of connections available between PEs. A successful routing will not occur if this constraint is violated. An implementation is not possible if the size of a partition exceeds the available capacity for a single PE. Both constraints must be met for a successful implementation.

The third constraint deals with the memory access for each PE. The architecture of the ACS imposes that a local SRAM to each FPGA is available for data writing and date reading. A single PE can only access its local SRAM. This means, the SRAM available to each PE can be used either for writing the data to the output or for reading the input data. Therefore, a partition can contain only one RAM access module. To explain this point in more detail, we consider a very simple example shown in Figure 2.4.

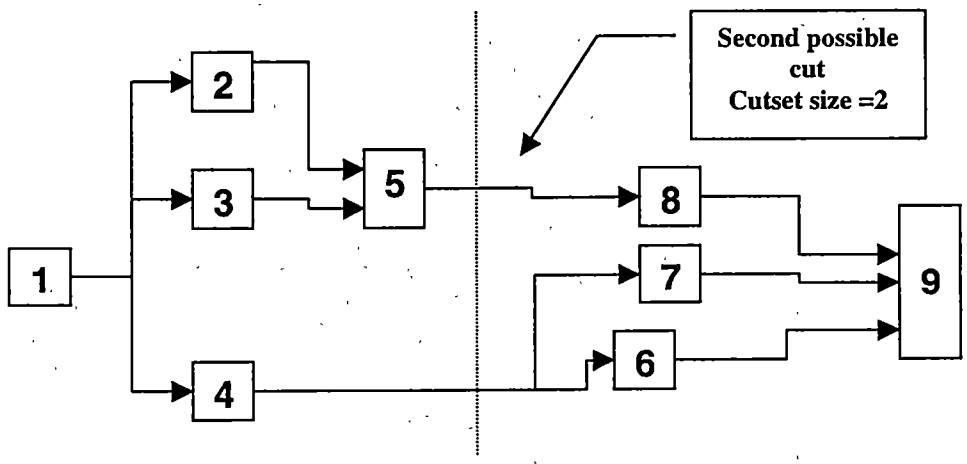
We consider a network with 9 nodes and 7 edges each of equal size. We assume that the node pair (1,9) represents two RAM access modules. Node 1 is used to read the input data and node 9 is used to write the output data. This implies that at least two partitions and only one board configuration are needed. For illustration, we further assume that the entire application fits in two PEs. Two possible partitions are shown in Figure 2.4a where both have equal cutset. The first possible partition showing in 2.4b violates the RAM access constraint since the partitioner places the node pair (1,9) in the same PE. Therefore, this partitioning cannot be implemented. The second possible partition showing in 2.4c satisfies the capacity constraint, I/O constraint, RAM access constraint, input and output module constraint, and the acyclic constraint. Therefore, the application can be implemented successfully.



a. A simple example



b. First possible cut



c. Second possible cut

Figure 2.4 Two Possible cuts

The fourth constraint deals with reading and writing the data. The input data is read first and supplied to the rest of the glyphs. The resulting data must be written to the hard disk via the output module. Therefore, the input module must be located in the first partition and the output module in the last partition. Both the input and output nodes must use RAM access modules. Even though an entire application might fit in one PE, this limitation imposes that at least two partitions are needed for successful implementation. This means that the partitioning process is always required in the CHAMPION design flow.

The fifth constraint deals with temporal partitioning of the ACS board. A single configuration of the board is the same as the configuration of all available PEs. If the entire application cannot fit in one board configuration, then multiple configurations of the board are necessary. Figure 2.5 shows a simplified version of the Wildforce board. When multiple configurations are used, storage of intermediate results between board configurations is needed. In this case, one RAM-read hardware glyph must be added at the beginning of each configuration and one RAM-write hardware glyph must be added at the end of each configuration. This process makes the partitioning problem more complicated since these two RAM hardware glyphs do not exist in the original application. In this case, the partitioner deals with a modified version of the original application netlist. Furthermore, the capacity of the first and the last PE is utilized by a certain amount for the RAM-read and RAM-write modules insertion.

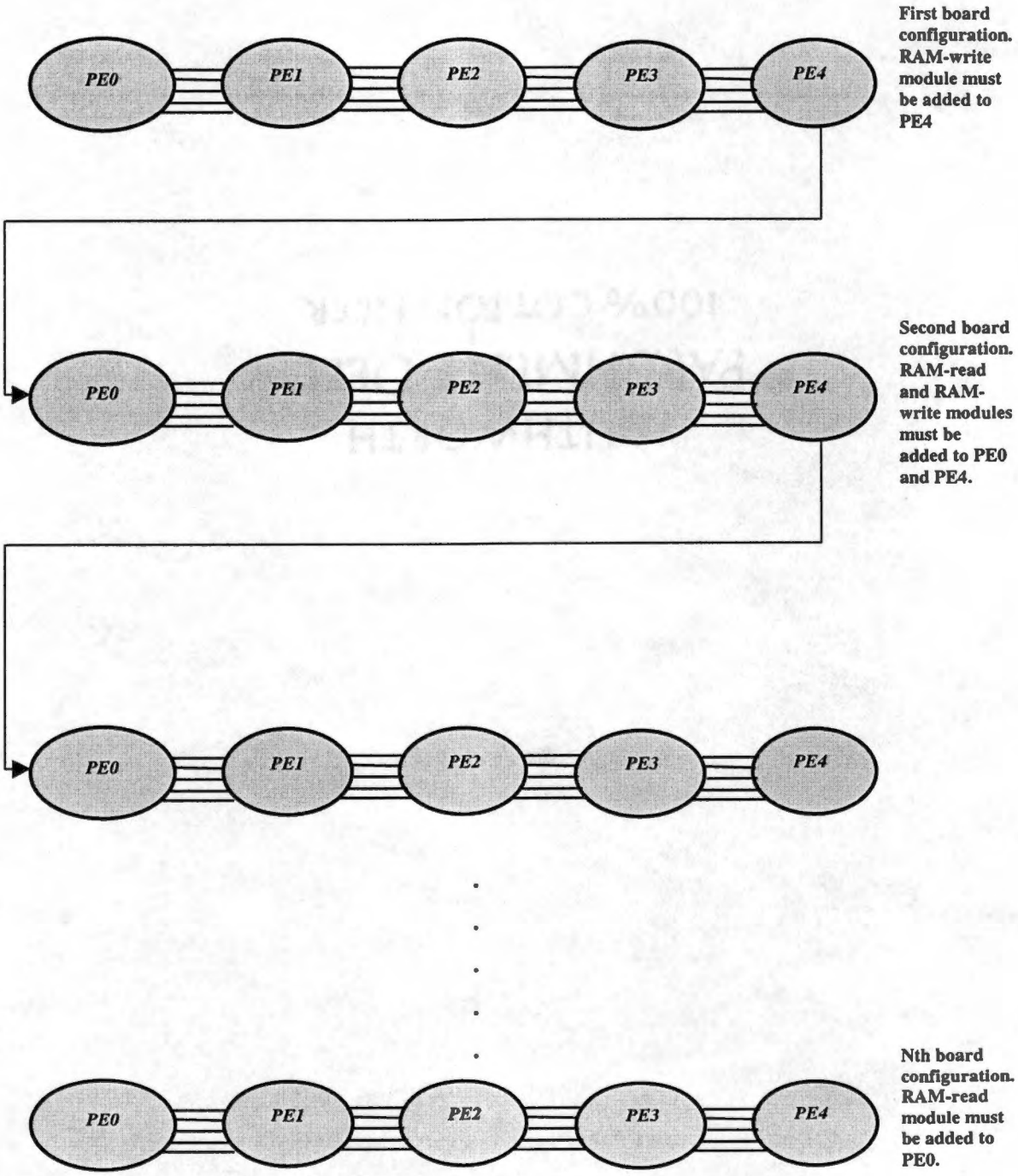
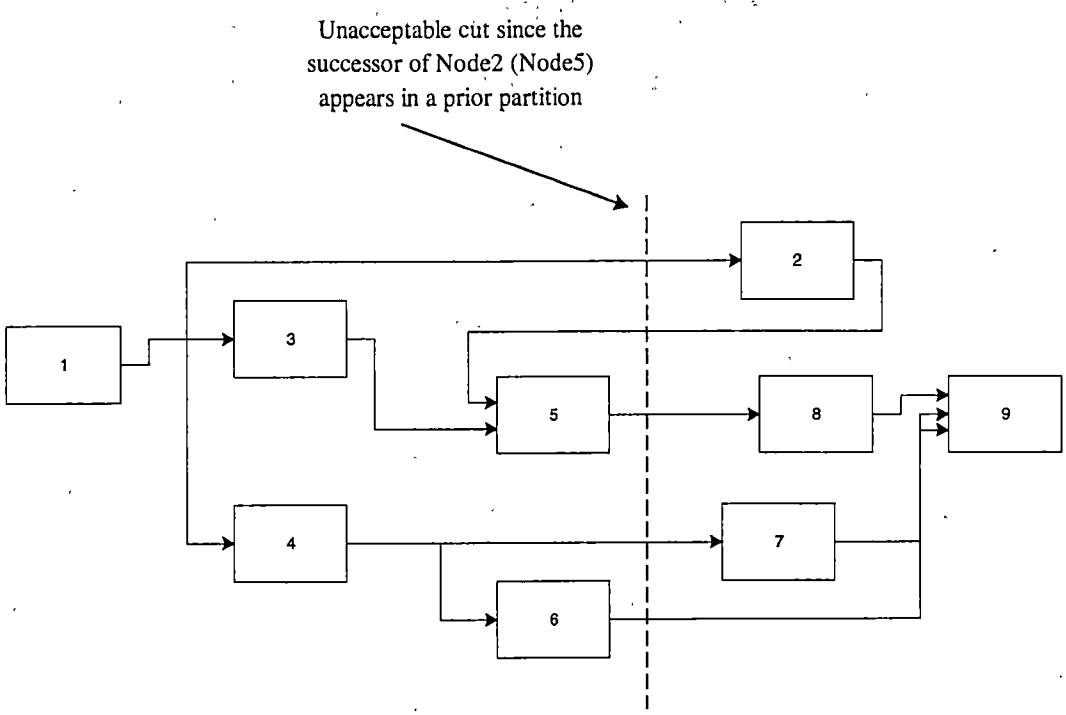


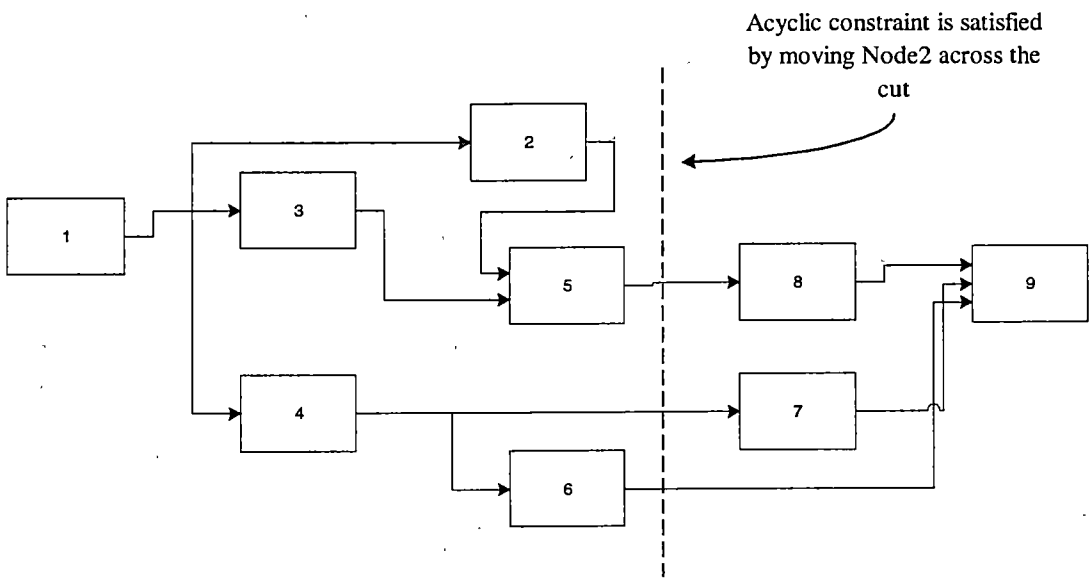
Figure 2.5 Multiple board configuration

The sixth, and final, constraint requires maintaining the direction of the hypergraph so that all edges are pointing the same way. The resulting sub-netlists $P = \{P_1, P_2, \ldots, P_k\}$ must maintain the acyclic constraint so that nodes in partition P_i must appear before the nodes in partition P_j. To illustrate the acyclic constraint, we consider the example from Figure 2.4. In Figure 2.6, we show two possible cuts for this network. The first cut showing in 2.6a violates the acyclic constraint since **node 2** appears in a prior partition. Moving **node 2** or **node 5** across the cut can satisfy the acyclic constraint as shown in Figure 2.6b.

A partitioning result is a theoretical solution that could possibly be implemented onto the given board architecture. Violation of any constraint discussed above will result in an unsuccessful mapping. In this case, the entire application must be repartitioned until a practical solution is found. Thus, our partitioning approaches must guarantee that all partitions will meet these constraints.



a. First possible cut



b. Second possible cut

Figure 2.6 Asyclic constraint

3. Partitioning Algorithms

In this chapter, we consider the problem of partitioning a large netlist into a collection of sub-netlists such that each sub-netlist will fit into one of the PEs where each PE is characterized by its capacity, I/O count, and RAM access. In addition to finding a feasible partitioning solution that meets the partitioning constraints, our objective includes the minimizing of the total number of the PEs used to implement a particular application.

In the previous chapter, we mentioned that there exist many algorithms dealing with the partitioning problem, but no general model exists that handles an arbitrary partitioning problem. For this reason, one algorithm might be developed or a new approach must be created to target a specific hardware structure. To the best of our knowledge, there exists no approach that can take the CHAMPION netlist and produce a valid partitioning result without development. Therefore, it was necessary to study some of the existing approaches and pick up one or two that can be adapted easily. Two main points were considered when we surveyed the existing work. The first point is that we do not know in advance how many partitions or PEs are needed to implement the CHAMPION netlist. An estimate can always be made based on the application size and the RAM access constraint. In this case, an algorithm such as MP2 cannot be easily adapted since this algorithm requires the number of blocks k to be specified in advance. In the second point, the immediate evaluation of the produced feasible solution in each partitioning stage is always preferred for a partitioning problem with several constraints.

In this case, an algorithm such as the multiple-way approach cannot be used since this approach seeks to minimize the total number of interconnections between the blocks. The minimization of the total number of interconnections between the PEs does not assure a feasible solution for an individual PE. Because of this, the hierarchical partitioning methods and the recursive algorithms were the best candidates among the existing work. They fulfill our two consideration points.

For solving the partitioning problem, three different approaches are investigated in this work. In the first and the second approaches, we discuss the development and implementation of two existing algorithms. The first approach is a hierarchical partitioning method based on topological ordering (HP). The second approach is a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic (RP). We extend these algorithms to handle the RAM access constraint, the acyclic constraint, and the temporal partitioning constraint. We shall describe the details of these two algorithms, including modifications and extensions.

We also introduce a new recursive partitioning method based on topological ordering and levelization (RPL). The details of this approach shall be described and explained by an illustrative example. In addition to handling the partitioning constraints, the new approach efficiently addresses the problem of minimizing the amount of computation thereby overcoming the weaknesses of the HP and RP algorithms. All three algorithms start with a topological sorting solution of the given application netlist. This solution will

assure that no node is processed before any node that points to it. This step is necessary to maintain the acyclic constraint so that all the nodes point the same way from left to right.

3.1 Hierarchical Partitioning Based on Topological Ordering (HP)

A topological sorting solution of a given network is a linear ordering L of all nodes N, such that node i appears before node j if the output of i is an input of j. This means, no node is processed before any node that points to it. The breadth-first search (BFS) algorithm is used to generate a topological sorting solution L. The BFS is a natural way to visit every node and check every edge in the graph systematically. This step is necessary to maintain the acyclic constraint so that all the nodes point the same way from left to right. Given a topological sorting solution L of all nodes, we can partition the list L from left to right into K Sub-Netlists $P = \{P_1, P_2, \ldots, P_k\}$ such that the constraints mentioned above are not violated.

Initially all nodes are in the linear ordering L and the first block P_1 is empty. Figure 3.1 shows the initial step of the HP algorithm. At each step, we select a node i from L and put it into P_1. The algorithm moves nodes into P_1 until the capacity constraint or the RAM access constraint of the partition is violated. The RAM access constraint is checked first. Each partition can have only one RAM access module. The capacity constraint is checked next. Once one of these constraints is violated, the algorithm checks if the interconnect constraint of the current PE is satisfied. If this interconnect constraint is violated, the algorithm rolls back the moves until the constraint is met. Rolling back

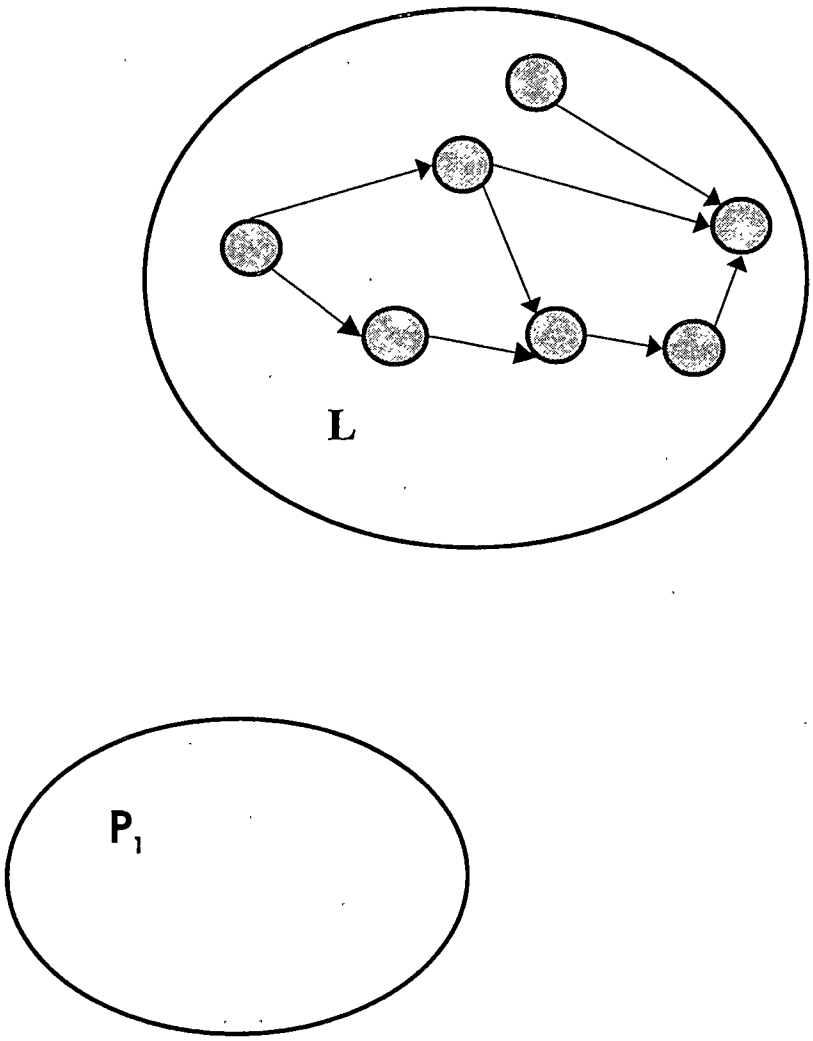


Figure 3.1 Initial Phase of HP.

refers to moving nodes back to L. If there exists more than one candidate node for back rolling, the algorithm starts an optimization step. This situation arises when for example two nodes with different sizes will lead to the same cut set if one of them is moved back to L. In this case, it is intuitive to keep the node with the maximum size in the current partition. This optimization step finds the best node which maximizes the capacity of the current partition and meets the interconnect constraint. The optimization strategy uses the benefit function to find the node with the highest benefit and leaves it in the current partition.

We repeat this process by creating a new block P_2 and applying the same procedure to the remainder of L. Figure 3.2 shows some of the nodes moved to the first and second partitions. The process stops when the list L becomes empty and all nodes are in $P = \{P_1, P_2, \ldots, P_k\}$. The algorithm fails to find a solution to the partitioning problem if one of the constraints is violated. Figure 3.3 shows the pseudo code for the HP algorithm.

3.2 Recursive Algorithm Based on Fiduccia and Mattheyses Bipartitioning Heuristic (RP)

The second approach to this problem is a recursive algorithm based on the Fiduccia and Mattheyses (FM) bipartitioning heuristic. The FM algorithm starts with initial bipartition A and B and iterates to improve it by reducing the cutset size. For some applications, a random method is used to generate an initial bipartition.

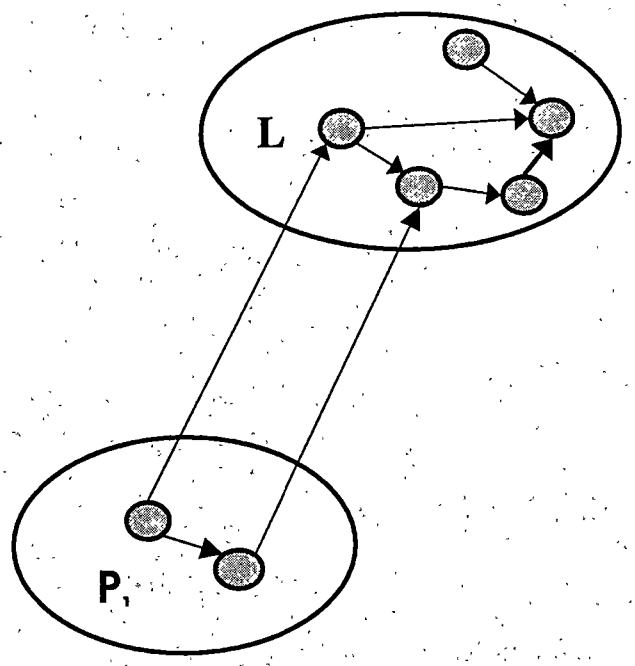


Figure 3.2 Illustration of the Partitioning using HP

```
Input: G(N,E), D=devices;
Output: P1, P2, P3 ,.., Pk
Create a linear ordering solution L; k=0;
Create a new partition, Pk;
While( emptyL =0) begin
        while(violation=False) begin
          move nodes into Pk; record size, I/O count;
         check constraints of Pk;
       end while;
       if(violation=True) then begin
          if there is more than one candidate then
             Optimize Pk;
             Reverse move until violation=False;
       end
       record final partition size, I/O count;
       if L is empty then
          emptyL=1;
       else begin
          k=k+1; Create a new partition, Pk;
       end
end while;
```

Figure 3.3 Pseudo-Code for the HP Algorithm

An arbitrary random bipartition can no longer be used here since it may violate the acyclic constraint. An efficient way of generating an acyclic initial bipartition is to use the linear ordering approach mentioned above. To create an initial solution, nodes are moved from the linear ordering array L into the current partition P_i until the capacity constraint of P_1 has been met. The recursive bipartitioning strategy, as illustrated in Figure 3.4, can be viewed as an extreme case of unbalanced bipartitioning. This means, the size of the current partition is much smaller than the remainder size of L during the first partitioning stages. In this process, each application of the bipartitioning procedure produces one feasible sub-netlist and the remainder netlist. In the first iteration, the entire netlist R_0 is partitioned into one feasible solution, P_1, that meets the constraints of the first PE on the board, and the remainder partition, R_1. In this case, all the N nodes in the current partition P_1 and a remainder R_1 are unlocked and involved in the partitioning process. When the partitioner finds a feasible solution for the current partition P1, the nodes in that partition become locked. In other words, the nodes in P1 are no more involved in the subsequent iterations. The run time of the RP algorithm is a function of the total number of nodes. During the partitioning process, the run time decreases as the number of locked nodes increases. Subsequent iterations apply the same procedure to the remainder until all resulting partitions meet the constraints. Figure 3.5 shows the pseudo code for the RP algorithm.

The RP algorithm uses the FM concept of moving a single node either from A to B or from B to A. As mentioned before, the FM algorithm was designed to handle imbalance.

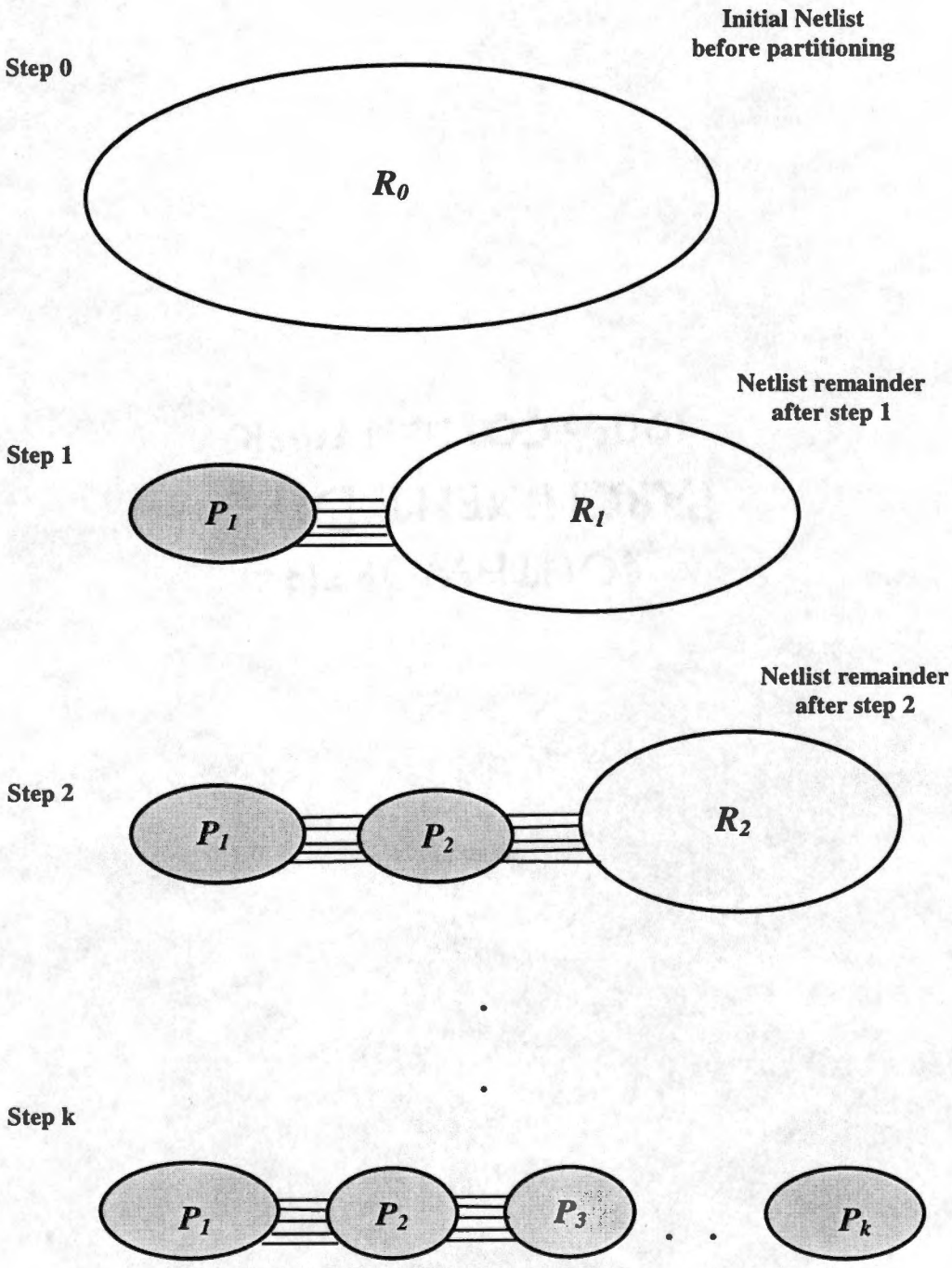


Figure 3.4 Recursive algorithm based on FM algorithm

```
Input: G(N,E), D = devices;
Output: P1, P2, P3 ,.., Pk
Create a linear ordering solution L;
k=0;
R0 =L;
Proceed=True;
While (proceed) begin
        Create initial bipartition {Pk , Rk};
        Optimize bipartition {Pk , Rk} with FM heuristic;
                Subject to the current device Dk ;
         Record final partition size, I/O count;
        if (Rk = Dk+1)   /* Rk fit into the next device Dk+1 */
        proceed=False;
        else
           k=k+1;
end while
```

Figure 3.5 Pseudo-Code for the RP Algorithm

To avoid having all nodes migrate to one block, the balance criterion defined in section 2.1.2 must be maintained.

3.3 A New Recursive Partitioning Method Based on Topological Ordering and Levelization (RPL)

In this section, we present our new approach for solving the partitioning problem for a CHAMPION netlist. This algorithm applies existing ideas to graph partitioning; however, they have not been used in this manner previously. The algorithm strategy is based on two steps: the level construction step and the partitioning step. Below, we describe the procedure to partition the CHAMPION netlist with this approach. In the following two sections, we demonstrate the level construction step and the partitioning step in more detail with an illustrative example.

The algorithm starts with a linear ordering solution for all the N nodes. Given a topological sorting solution of all nodes, we can construct multi levels $L = \{L_1, L_2, \ldots, L_n\}$ of nodes N with a modified version of breadth-first search (BFS), such that nodes in level L_i appear before nodes in level L_j and the nodes in level L_j must be successors of the nodes in level L_i. Afterwards, the resulting flow is reduced to a form shown in Figure 3.6. Each level consists of a subset of nodes and no level can have more than one RAM access node. Level construction is a onetime step and is denoted as a preprocessing step. Since the RAM access constraint is a very challenging one, we expect the preprocessing step to solve any conflicts associated with this constraint before moving to the

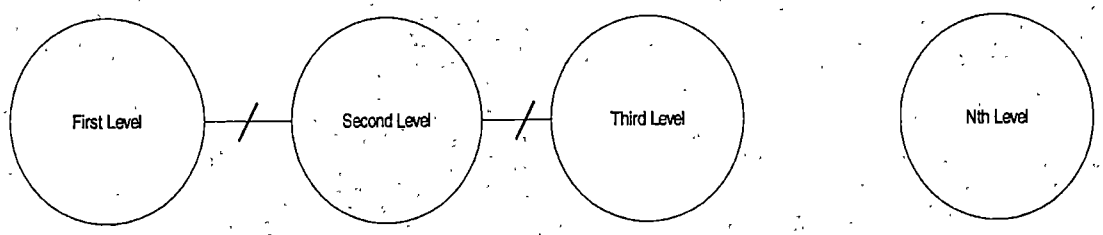


Figure 3.6 Reduced form after level construction.

partitioning step. Given a levelization solution $L = \{L_1, L_2, \ldots, L_n\}$, we can partition L from left to right into K Sub-Netlists $P= \{P_1, P_2, \ldots, P_k\}$. Initially all nodes are in L and the first block P_1 is empty. First, we start the process by moving n levels from L and put them into P_1 until P_1 has met the capacity constraint. At this moment, the last level moved to P_1 is marked as L_i . Then we start an optimization step by moving nodes across the marked level L_i and its successor level L_{i+1} until the rest of the constraints are satisfied. The optimization step is based on the benefit function discussed above. Since only two levels are involved in the optimization step, the number of nodes involved in the

optimization process and the computation amount are reduced significantly. If the algorithm fails to find a valid solution, then we reduce the size of P_1 by removing the last level moved to P_1 and the optimization step is repeated. We repeat this process by creating a new block P_2 and applying the same procedure to the remainder of L. The process stops when L becomes empty and all nodes are in $P = \{P_1, P_2, \ldots, P_k\}$. Figure 3.7 shows the pseudo code for the RPL algorithm.

3.3.1 Level Construction step

In this section, we demonstrate the level construction step in more detail. The construction step is very crucial to our proposed method and helps the RPL to reduce the weaknesses inherent in the other two approaches. This step is best demonstrated by considering a very simple example, shown in Figure 3.8, with two sources S_1 and S_2 and two destinations $D_1$and D_2. This process uses a modified version of breadth-first search (BFS) for constructing levels. To search the nodes of the graph systematically, we begin with the first source S_1 as a starting point, all others nodes are unseen. The BFS completely covers the area close to the starting point, moving farther away only when everything close has been looked at. The source node S_1 and its successors (2,5) will construct the first level. Before visiting any successors of (2,5) we check out to see if any of the nodes (2,5) has to wait for other nodes which are not included in the current level. In this particular example, we see that node 5 has to wait for node 8. We force the algorithm to add node 8 to the current level.

```
Input: G(N,E), D = devices;
Output: P1, P2, P3 ,.., Pk
Create levels L = { L1, L2, . . ., Ln};
k=0; j=1;
Proceed=True;
violation=True;

While (Proceed) begin
        Create new partition Pk;
        while(violation=False) begin
                move level Lj to Pk;
                check constraints;
                mark the last level moved to Pk as Lj and its successor Lj+1;
                j=j+1;
         end while
        Optimize Pk by moving nodes across {Lj, Lj+1}
                Subject to the current device Dk ;
         Record final partition size, I/O count;
        if (L= Dk+1)   /* if the remained of L fit into the next device Dk+1 */
        Proceed=False;
        else
           k=k+1;
end while
```

Figure 3.7 Pseudo-Code for the RPL Algorithm

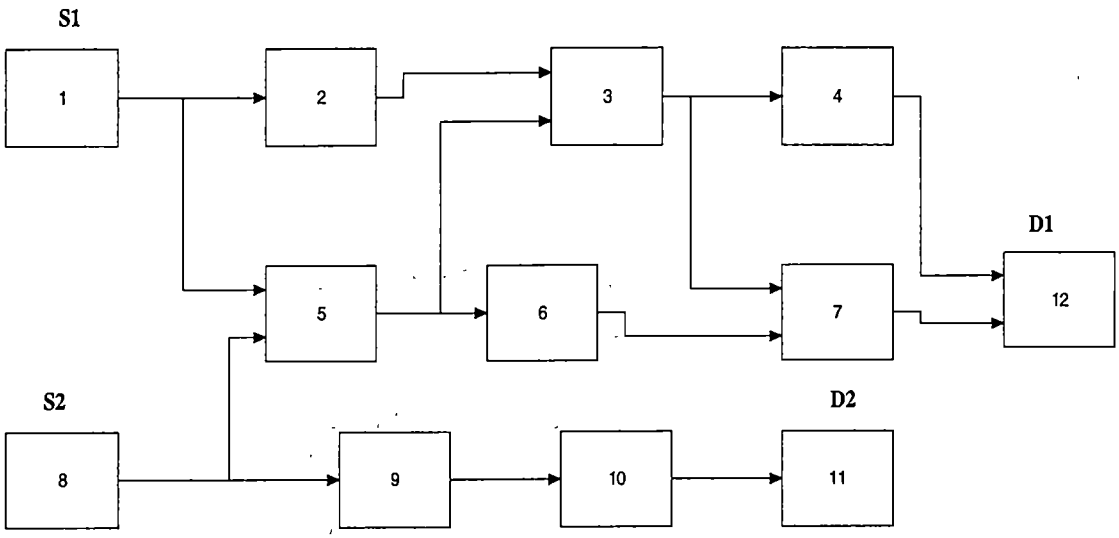
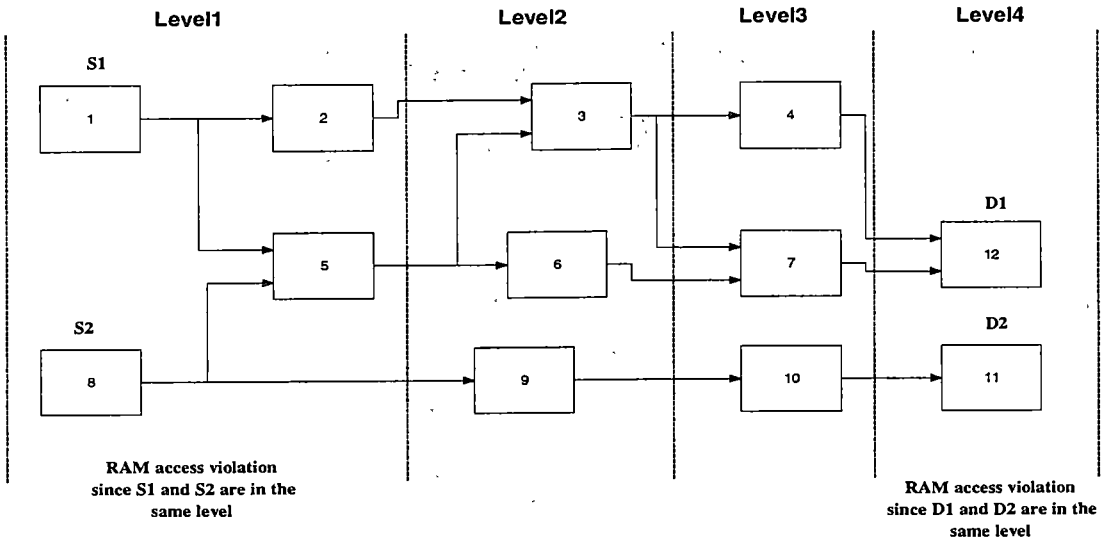


Figure 3.8 Illustrative example for RPL algorithm

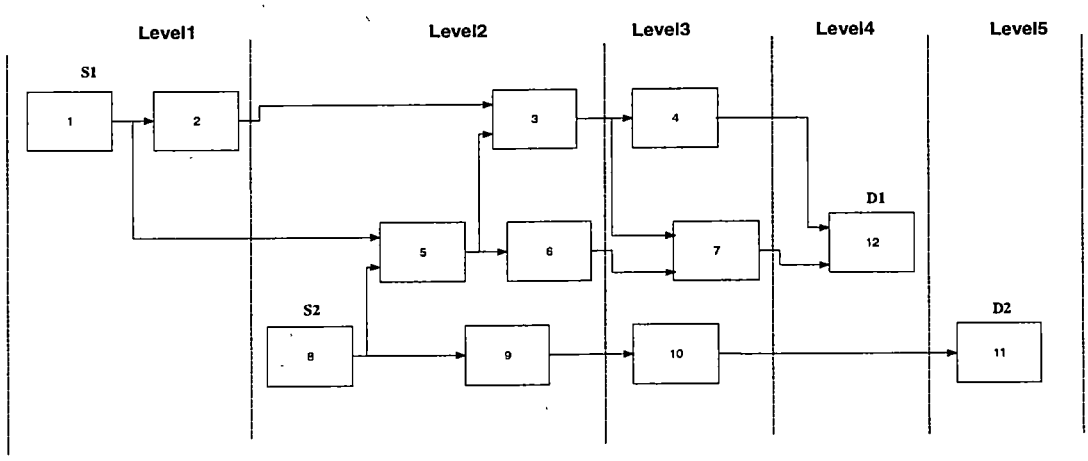
The set (1,2,5,8) now constructs the first level. We proceed with the construction step for finding the second level by visiting the successors of the nodes (2,5,8). The successors of those nodes are (3,6,9) which build up the second level. We repeat the process until all the N nodes have been visited. The resulting levels are shown in Figure 3.9. The next step of level construction is to check if there is a RAM access conflict inside each level. Figure 3.9a shows a RAM access conflict inside the first and the fourth level. To remove this conflict, we move node S_2 and its successors to the second level. In the fourth level, depending on the cut set, either D_1 or D_2 has to be moved to a new level. For this example, we move D_2 to the fifth level since the cut set created after the move is lower. The final step is shown in Figure 3.9b.

3.3.3 Partitioning Step

In this section we demonstrate the partitioning step in more detail by considering the simple network from the previous section. At this point, we assume that the levelization step has been done and the solution $L = \{L_1, L_2, \ldots, L_n\}$ is available to the partitioner. Initially all nodes are in L and the first block P_1 is empty. First, we start the process by moving n levels from L and put them into P_1 until P_1 has met either the capacity constraint or the RAM access constraint. For this particular example, as shown in Figure 3.10 and Figure 3.11, the first level is moved to P_1. A RAM access conflict occurred when the algorithm tries to move the second level L_2 to P_1 since and S_1 and S_2 are RAM access nodes. Before creating a new block, the algorithm starts to optimize the capacity utilization of P_1 by moving nodes across the first and second levels.

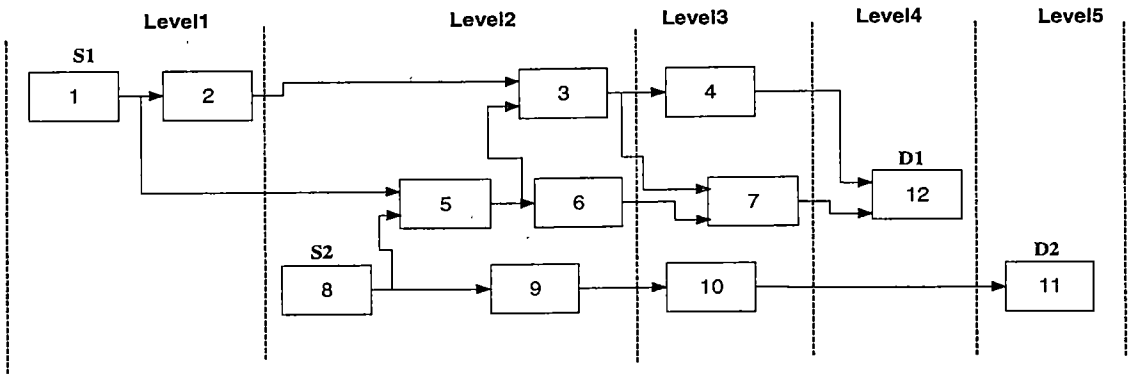


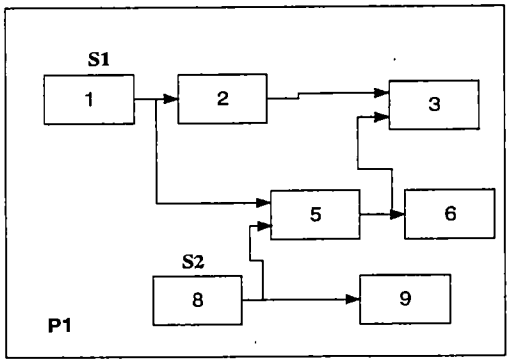
a. RAM access conflict



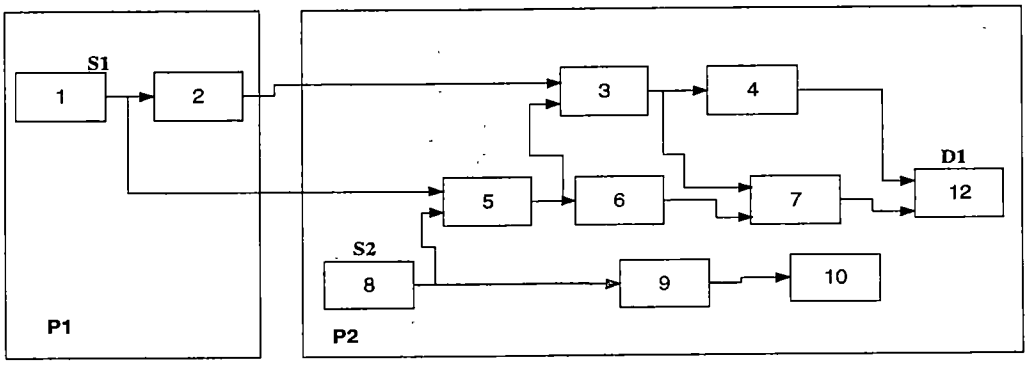
b. Removing the RAM access conflict

Figure 3.9 RAM access conflict





RAM access violation since S1 and S2 are in the same PE



RAM access violation since S2 and D1 are in the same PE

Figure 3.10 Illustration of the partitioning step

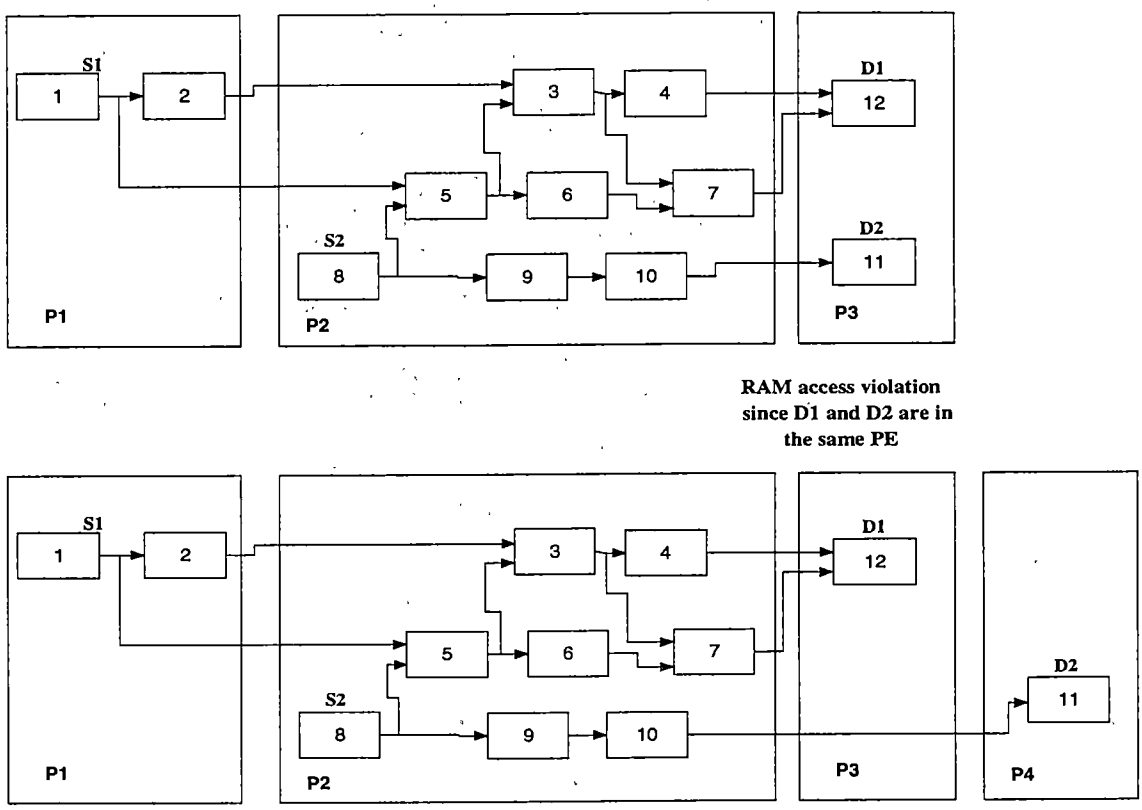


Figure 3.11 Illustration of the partitioning step

The algorithm fails to move any node from the second level to the current partition since any node will violate the acyclic constraint. Therefore, the current partition P_1 consists only of the nodes (1,2). If the total size of the nodes (1,2) is much smaller than the available capacity of P_1, then the current PE is very poorly utilized. This case shows how the RAM access constraints can limit a full utilization of one PE. The algorithm proceeds by creating a new block P_2. Because of a RAM access conflict, only the second and the third levels are moved to the current partition. Only two levels remain in L where each of these requires access to the local RAM. Therefore, another two new blocks are created P_3 and P_4. The partitioning process stops since L became empty.

4. Experimental Results and Analysis

In this chapter, we discuss the partitioning results for the three different approaches in which we focused on the development of two different existing partitioning algorithms and the development of a new partitioning approach. The HP and the RP algorithms were developed to consider our partitioning constraints. Our new idea, the RPL approach, was developed to cope with the weaknesses of the HP and RP algorithms.

The three partitioning approaches were implemented successfully in the C++ language and are currently used to partition the CHAMPION netlists. These algorithms were run on several netlists targeting different hardware architectures. Some of these netlists were generated randomly by using a random netlist generator developed in this research to test these algorithms.

Partitioning for Hardware Architectures

In the following sections, we will discuss the partitioning results for each algorithm by targeting six different hardware architectures. The first hardware architecture is the Wildforce-XL board from Annapolis Micro Systems, discussed in section 1.2, which uses five Xilinx FPGAs. In the second hardware architecture, we consider another version of the Wildforce-XL. We assume that the board has the same structure as the previous one but comes with larger size FPGAs (10 times bigger), bigger RAMs for each FPGA (two RAM modules for each FPGA), and larger I/O count (72-bit

systolic bus). We denote this version as Wildforce-XL1. In the third and fourth hardware architectures, we consider the MSP board from, which uses two Altera FLEX10k FPGAs. In this board, RAM modules can be organized as one RAM bank or two RAM banks. In the fifth hardware architecture, we consider the SLAAC-1V board. This board uses two Virtex FPGAs. The sixth hardware architecture deals with the SLAAC-1P which uses two Xilinx XC4000 FPGAs.

Partitioning Netlists

Different netlists are used in this research to determine the performance of the three partitioning algorithms. The partitioning algorithms are run on identical netlists in order to compare the algorithms against each other. The comparison is based on the number of PEs required for implementing the netlist and the running time. Table 4.1 shows the different netlists used in this research. The first three netlists were automatically mapped and implemented successfully from the Cantata workspace to the Wildforce-XL platform. The automatic target recognition (ATR) is relatively a complex netlist. The ATR was first implemented manually by Ben Levine to assist in the development of function libraries for use in the CHAMPION system [4]. The mapping techniques used were developed in such a way that they could serve as the basis for the automated system. The ATR netlist was used to test our three different partitioning approaches. This netlist consists of 101 nodes and 234 hyper nets. Among the 101 nodes, 14 nodes are used for accessing the local RAMs. The M29 netlist is a very challenge one

Table 4.1 Partitioning Netlists

Netlists	Size	RAM Modules	Nodes #	Nets #
Hipass Filter	458	2	17	48
NVL	549	2	45	71
ATR	4885	14	101	234
M29	29	7	29	28
R300	7845	11	301	311
R400	10130	7	406	421
R500	12845	12	504	493
R600	15320	9	601	571
R700	17640	10	702	714
R800	19690	12	807	809
R900	23041	18	906	881
R1000	24533	23	1005	1041
R1100	29685	23	1101	1091
R1200	31420	22	1202	1231
R1300	33120	25	1303	1243
R1400	36975	29	1401	1412
R1500	41453	34	1502	1497

which utilizes 7 RAM modules. This netlist was generated manually by utilizing a significant number of RAM nodes to challenge the three partitioning algorithms. To the best of our knowledge, there exist no benchmarks that represent our partitioning constraints. For this reason, a random netlist generator to produce benchmarks was developed in this research to investigate the performance of the three partitioning algorithms. In addition, this will enable us to investigate the running time of each approach.

The netlists R300-1500 were produced randomly with a random netlist generator. The random netlist generator uses a random number generator seeded by the time of the system to generate the nodes and to map the connection between these nodes. Initially, nodes are placed into n blocks where each block has a certain number of nodes. The number of nodes for the ith block B_i is selected randomly between 1 and 10. For example, if the number chosen were 3, this would generate a block with 3 nodes. For each node in the block B_i a random number between (1 and # of nodes in block B_{i+1}) is selected to map the nodes in block B_i to the nodes in block B_{i+1}. The nodes in block B_{i+1} that mapped to nodes in block B_i are selected randomly. For example, we consider two blocks B_3 and B_4 with 2 nodes and 5 nodes in each block respectively. In this case, random numbers between 1 and 5 will be selected for each node in B_3. For example, if the number chosen were 2 for the first node in block B_3, this would map this node to two nodes in block B_4. In addition, the random netlist generator uses a random number selected between 1 and 100 to assign weights to the nets and nodes. Finally, the random netlist generator selects the RAM nodes randomly by using a random number between 1

and 100. For each node, if the random number chosen lies between 45 and 55, the node is assigned for RAM access.

Partitioning Configuration File

As mentioned before, one of the major goals in this research is to target different hardware architectures. The partitioning algorithms are extended to be dynamic. In this case, the partitioner reads the particular hardware structure data from a partitioning configuration file. This enables the user to change some specific information in the configuration file instead of working inside the partitioner. This gives the user the flexibility of switching from one hardware architecture to another. The hardware architecture data includes the number of PEs available in the ACS, the capacity of each PE, the I/O pins between PEs, the I/O pins between each PE and its local RAM, and the number of RAMs available for each PE. In addition, the user can specify the data that is required to configure the memory used by the partitioner. This includes the maximum number of nodes, the maximum number of nets, the maximum number of levels, and the maximum number of the produced partitions. Figure 4.1-4.3 show the partitioning configuration files for the Wildforce-XL, Wildforce-XL1, and SLAAC-1V boards respectively. Using this approach, a user can adopt the partitioner to a new ACS in only few minutes. This is valuable information which may be used to select an existing hardware platform or to consider tradeoffs in the design of a new ACS. However, the steps involved in connecting partition blocks to PE I/O and communicating with the host CPU may take several days to accomplish.

```
/* Partitioner Configuration File For The Wildforce-XL Board */

# define  NumberOfPE      5
/* Number of PEs or FPGAs available on the ACS board */
# define  MaxNodeSize  3000      /* Maximum Number of Nodes */
# define  MaxLevelSize  700
/* Maximum Number of Levels. This is required for the RPL Algorithm */
# define  MaxNetSize  4000       /* Maximum Number of Nets */
# define  MaxPartNumber  300     /* Maximum number of partitions */
# define  PE0  1290              /* Capacity of PE0 */
# define  PE1  570               /* Capacity of PE1 */
# define  PE2  570               /* Capacity of PE2 */
# define  PE3  570               /* Capacity of PE3 */
# define  PE4  570               /* Capacity of PE4 */
# define  UF   0.9               /* utilization factor for each PE */
# define  TotalNumOfCLB  3550    /* Total CLBs available on the ACS board */
# define  PE2MEMCutset  32
/* Maximum I/O count between one PE and the local RAM */
# define  PE2PECutset     36
/* Maximum I/O count between one PE and another PE */
# define  PEMemNum  1            /* Number of local RAMs available for each PE */
```

Figure 4.1 Partitioning Configuration File for Wildforce-XL

```
/* Partitioner Configuration File For The Wildforce-XL1 Board */

# define  NumberOfPE      5
/* Number of PEs or FPGAs available on the ACS board */
# define  MaxNodeSize  3000        /* Maximum Number of Nodes */
# define  MaxLevelSize  700
/* Maximum Number of Levels. This is required for the RPL Algorithm */
# define  MaxNetSize  4000         /* Maximum Number of Nets */
# define  MaxPartNumber  300       /* Maximum number of partitions */
# define  PE0  1290*10                    /* Capacity of PE0 */
# define  PE1  570*10                     /* Capacity of PE1 */
# define  PE2  570*10                     /* Capacity of PE2 */
# define  PE3  570*10                     /* Capacity of PE3 */
# define  PE4  570*10                     /* Capacity of PE4 */
# define  UF    0.9                 /* utilization factor for each PE */
# define  TotalNumOfCLB  35500  /* Total CLBs available on the ACS board */
# define  PE2MEMCutset  32 *2
/* Maximum I/O count between one PE and the local RAM */
# define  PE2PECutset     36*2
/* Maximum I/O count between one PE and another PE */
# define  PEMemNum  2              /* Number of local RAMs available for each PE */
```

Figure 4.2 Partitioning Configuration File for Wildforce-XL1

```
/* Partitioner Configuration File For The SLAAC-1V Board */

# define  NumberOfPE        2
/* Number of PEs or FPGAs available on the ACS board */
# define  MaxNodeSize  3000        /* Maximum Number of Nodes */
# define  MaxLevelSize  700
/* Maximum Number of Levels. This is required for the RPL Algorithm */
# define  MaxNetSize  4000         /* Maximum Number of Nets */
# define  MaxPartNumber  300       /* Maximum number of partitions */
# define  PE0  1000000                    /* Capacity of PE0 */
# define  PE1  1000000                    /* Capacity of PE1 */
# define  UF   0.9                  /* utilization factor for each PE */
# define  TotalNumOfCLB  2000000    /* Total CLBs available on the ACS board */
# define  PE2MEMCutset  36
/* Maximum I/O count between one PE and the local RAM */
# define  PE2PECutset   72
/* Maximum I/O count between one PE and another PE */
# define  PEMemNum  4              /* Number of local RAMs available for each PE */
```

Figure 4.3 Partitioning Configuration File for SLAAC-1V

4.1 HP Algorithm

In this section we discuss the partitioning results of the HP algorithm for the netlist shown in Table 4.1. In 1997, Stanley developed this algorithm targeting a particular hardware emulator [8]. The HP algorithm was developed and extended in this research to handle the CHAMPION netlists. The algorithm is relatively a simple idea compared with the RPL and RP algorithms. As mentioned in section 3.1, the algorithm starts with a linear ordering L of all nodes and with an empty block P_1. At each step, the algorithm selects a node i from L and put it into P_1. The algorithm moves nodes into P_1 until the capacity constraint or the RAM access constraint of the partition is violated. The RAM access constraint is checked first. Each partition can have only a certain number of RAM access modules based on the selected hardware board. The capacity constraint is checked next. Once one of these constraints is violated, the algorithm checks if the interconnect constraint of the current PE is satisfied. If this interconnect constraint is violated, the algorithm rolls back the moves until the constraint is met. Rolling back refers to moving nodes back to L. If there exists more than one candidate node for back rolling, the algorithm starts an optimization step. This situation arises when for example two nodes with different sizes will lead to the same cut set if one of them is moved back to L. In this case, it is intuitive to keep the node with the maximum size in the current partition. This optimization step finds the best node which maximizes the capacity of the current partition and meets the interconnect constraint. The optimization strategy uses the benefit function discussed in chapter 2 to find the node with the highest benefit and

leaves it in the current partition. We repeat this process by creating a new block P_2 and applying the same procedure to the remainder of L.

As mentioned above, the HP algorithm uses a linear ordering array to access the nodes and move theme across partitions. Therefore, the running time of the HP algorithm depends on the linear ordering array size, the number of the RAM access modules, and the selected hardware architecture. A complexity analysis for this algorithm was not undertaken in the existing work from Stanley [8]. In this research, for each hardware the running time for each netlist is presented. We will show how the running time varies with the selected hardware architectures and the netlist size.

Tables 4.2-4.7 show the partitioning results for the six hardware architectures. For each netlist the tables show the size of the netlist, the RAM modules count, the nodes size, the nets size, the partitions number, and the running time of partitioning process. We assume that the RAM modules require external implementation. The reported run times are for a 300MHz Pentium II CPU.

As expected, the partitions number produced and the running time for the partitioning process vary with the selected hardware board and the netlist size. Referring to Table 4.2, the HP algorithm was not able to produce valid partitioning results for the ATR, M29, R700, and R1500 when we targeted the Wildforce-XL board. The reason for this is the limited external I/O per PE. For the ATR netlist, a non-valid partitioning result was produced after we relaxed the I/O count from 36 to 50 for the same Wildforce-XL.

Table 4.2 Partitioning results for Wildforce-XL using HP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	Not feasible	2
M29	519	7	29	28	Not feasible	<1s
R300	7845	11	301	311	25	4
R400	10130	7	406	421	23	10
R500	12845	12	504	493	33	19
R600	15320	9	601	571	29	34
R700	17640	10	702	714	Not feasible	761
R800	19690	12	807	809	42	89
R900	23041	18	906	881	52	132
R1000	24533	23	1005	1041	58	184
R1100	29685	23	1101	1091	61	257
R1200	31420	22	1202	1231	67	336
R1300	33120	25	1303	1243	72	424
R1400	36975	29	1401	1412	81	571
R1500	41453	34	1502	1497	Not feasible	1129

Table 4.3 Partitioning results for Wildforce-XL1 using HP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	7	2
M29	519	7	29	28	4	<1s
R300	7845	11	301	311	6	3
R400	10130	7	406	421	4	8
R500	12845	12	504	493	6	13
R600	15320	9	601	571	5	25
R700	17640	10	702	714	6	41
R800	19690	12	807	809	7	71
R900	23041	18	906	881	10	107
R1000	24533	23	1005	1041	13	161
R1100	29685	23	1101	1091	13	221
R1200	31420	22	1202	1231	13	291
R1300	33120	25	1303	1243	15	383
R1400	36975	29	1401	1412	18	431
R1500	41453	34	1502	1497	21	521

Table 4.4 Partitioning results for MSP1 board using HP algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	14	1
M29	519	7	29	28	6	<1s
R300	7845	11	301	311	10	3
R400	10130	7	406	421	5	9
R500	12845	12	504	493	10	15
R600	15320	9	601	571	7	29
R700	17640	10	702	714	9	47
R800	19690	12	807	809	11	81
R900	23041	18	906	881	17	107
R1000	24533	23	1005	1041	22	172
R1100	29685	23	1101	1091	21	234
R1200	31420	22	1202	1231	21	313
R1300	33120	25	1303	1243	24	401
R1400	36975	29	1401	1412	27	453
R1500	41453	34	1502	1497	34	503

Table 4.5 Partitioning results for MSP2 board using HP algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	Not feasible	<1s
M29	519	7	29	28	Not feasible	<1s
R300	7845	11	301	311	Not feasible	<1s
R400	10130	7	406	421	Not feasible	<1s
R500	12845	12	504	493	Not feasible	<1s
R600	15320	9	601	571	Not feasible	<1s
R700	17640	10	702	714	Not feasible	<1s
R800	19690	12	807	809	Not feasible	<1s
R900	23041	18	906	881	Not feasible	<1s
R1000	24533	23	1005	1041	Not feasible	<1s
R1100	29685	23	1101	1091	Not feasible	<1s
R1200	31420	22	1202	1231	Not feasible	<1s
R1300	33120	25	1303	1243	Not feasible	<1s
R1400	36975	29	1401	1412	Not feasible	<1s
R1500	41453	34	1502	1497	Not feasible	<1s

Table 4.6 Partitioning results for SLAAC-1V board using HP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	4	1
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	2
R400	10130	7	406	421	2	6
R500	12845	12	504	493	4	9
R600	15320	9	601	571	3	19
R700	17640	10	702	714	4	37
R800	19690	12	807	809	4	62
R900	23041	18	906	881	6	89
R1000	24533	23	1005	1041	8	131
R1100	29685	23	1101	1091	8	192
R1200	31420	22	1202	1231	8	216
R1300	33120	25	1303	1243	8	289
R1400	36975	29	1401	1412	10	305
R1500	41453	34	1502	1497	11	351

Table 4.7 Partitioning results for SLAAC-1P board using HP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	Not feasible	2
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	2
R400	10130	7	406	421	2	7
R500	12845	12	504	493	4	20
R600	15320	9	601	571	Not feasible	14
R700	17640	10	702	714	4	41
R800	19690	12	807	809	4	66
R900	23041	18	906	881	6	98
R1000	24533	23	1005	1041	10	143
R1100	29685	23	1101	1091	8	203
R1200	31420	22	1202	1231	8	227
R1300	33120	25	1303	1243	9	297
R1400	36975	29	1401	1412	Not feasible	21
R1500	41453	34	1502	1497	Not feasible	27

This partitioning result cannot be implemented by targeting the Wildforce-XL since the I/O limitation is violated.

Targeting the Wildforce-XL1, the HP algorithm was able to produce valid partitioning results for all the netlists considered in this research. To compare the running times and the number of partitions produced by targeting the Wildforce-XL and the Wildforce-XL1, we consider the R1400 netlist. For the Wildforce-XL, 77 PEs are required to implement the R1400 netlist and the running time for the partitioner is 571 seconds. At the other side, only 17 PEs are required to implement the R1400 by targeting the Wildforce-XL1 while the running time was reduced to 431 seconds. This result is presented in Table 4.3. Targeting the SLACC-1V board and R1400 netlist, the HP produced 10 partitions within 305 seconds. This shows how the performance of the HP algorithm depends on the selected hardware architecture.

Referring to Table 4.5 the partitioning results are presented for the MSP board where the local RAM for each PE is organized as one bank of 512kX48 bits. In this case, only one RAM module is available for each PE. The HP failed to produce valid partitioning results for all netlists, which require multiple configuration of the board. A single configuration of the board is the same as the configuration of all available PEs. If the entire application cannot fit in one board configuration, then multiple configurations of the board are necessary. When multiple configurations are used, storage of intermediate results between board configurations is needed. In this case, one RAM read hardware glyph must be added at the beginning of each configuration and one RAM write

hardware glyph must be added at the end of each configuration. Using the RAM as one bank of 512kX48 bits will limit the MSP board for a single board configuration only since both RAM modules are always used to store the intermediate results. Organizing the RAM as two banks of 512kX24 bits can solve the problem. These results are shown in Table 4.4.

4.3 RP Algorithm

In this section we discuss the partitioning results for the RP algorithm by targeting the netlists shown in Table 4.1. The RP algorithm was developed and extended in this research to handle the CHAMPION netlists. The partitioning results are presented for the six hardware architectures considered in this research. The FM algorithm starts with a current partition P_1 and a remainder R_1 and iterates to improve it by reducing the cutset size. Subsequent iterations apply the same procedure to the remainder until all resulting partitions meet the constraints. The RP algorithm uses the FM concept, discussed in chapter 2, of moving a single node cross the cut.

Tables 4.8-4.13 show the partitioning results for the six hardware architectures. Referring to Table 4.8, the RP algorithm was not able to produce valid partitioning results for the M29 and the R700 netlists by targeting the Wildforce-XL board. The reason for this is the limited external I/O count per PE. For the ATR netlist, a valid partitioning result was produced with 23 FPGAs. The partitioning results show that the running time of the RP algorithm depends on the netlist size, the number of the RAM access modules,

Table 4.8 Partitioning results for Wildforce-XL using RP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	23	9
M29	519	7	29	28	Not feasible	2
R300	7845	11	301	311	23	26
R400	10130	7	406	421	21	39
R500	12845	12	504	493	32	89
R600	15320	9	601	571	24	138
R700	17640	10	702	714	Not feasible	61
R800	19690	12	807	809	36	171
R900	23041	18	906	881	49	195
R1000	24533	23	1005	1041	58	236
R1100	29685	23	1101	1091	57	292
R1200	31420	22	1202	1231	66	392
R1300	33120	25	1303	1243	66	501
R1400	36975	29	1401	1412	79	693
R1500	41453	34	1502	1497	91	863

Table 4.9 Partitioning results for Wildforce-XL1 using RP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	7	6
M29	519	7	29	28	4	<1s
R300	7845	11	301	311	6	17
R400	10130	7	406	421	4	31
R500	12845	12	504	493	7	62
R600	15320	9	601	571	5	101
R700	17640	10	702	714	7	125
R800	19690	12	807	809	7	143
R900	23041	18	906	881	10	177
R1000	24533	23	1005	1041	13	211
R1100	29685	23	1101	1091	13	276
R1200	31420	22	1202	1231	13	314
R1300	33120	25	1303	1243	15	461
R1400	36975	29	1401	1412	17	515
R1500	41453	34	1502	1497	21	594

Table 4.10 Partitioning results for MSP1 board using RP algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	14	7
M29	519	7	29	28	6	<1s
R300	7845	11	301	311	10	21
R400	10130	7	406	421	5	35
R500	12845	12	504	493	10	71
R600	15320	9	601	571	7	113
R700	17640	10	702	714	9	139
R800	19690	12	807	809	11	151
R900	23041	18	906	881	16	185
R1000	24533	23	1005	1041	21	222
R1100	29685	23	1101	1091	21	281
R1200	31420	22	1202	1231	21	331
R1300	33120	25	1303	1243	24	471
R1400	36975	29	1401	1412	27	523
R1500	41453	34	1502	1497	33	604

Table 4.11 Partitioning results for MSP2 board using RP algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	Not feasible	<1s
M29	519	7	29	28	Not feasible	<1s
R300	7845	11	301	311	Not feasible	<1s
R400	10130	7	406	421	Not feasible	<1s
R500	12845	12	504	493	Not feasible	<1s
R600	15320	9	601	571	Not feasible	<1s
R700	17640	10	702	714	Not feasible	<1s
R800	19690	12	807	809	Not feasible	<1s
R900	23041	18	906	881	Not feasible	<1s
R1000	24533	23	1005	1041	Not feasible	<1s
R1100	29685	23	1101	1091	Not feasible	<1s
R1200	31420	22	1202	1231	Not feasible	<1s
R1300	33120	25	1303	1243	Not feasible	<1s
R1400	36975	29	1401	1412	Not feasible	<1s
R1500	41453	34	1502	1497	Not feasible	<1s

Table 4.12 Partitioning results for SLAAC-1V board using RP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	5	2
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	11
R400	10130	7	406	421	2	17
R500	12845	12	504	493	4	44
R600	15320	9	601	571	3	89
R700	17640	10	702	714	4	101
R800	19690	12	807	809	4	121
R900	23041	18	906	881	6	152
R1000	24533	23	1005	1041	8	174
R1100	29685	23	1101	1091	8	205
R1200	31420	22	1202	1231	8	281
R1300	33120	25	1303	1243	8	379
R1400	36975	29	1401	1412	10	411
R1500	41453	34	1502	1497	11	434

Table 4.13 Partitioning results for SLAAC-1P board using RP algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	Not feasible	11
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	13
R400	10130	7	406	421	2	18
R500	12845	12	504	493	4	47
R600	15320	9	601	571	Not feasible	31
R700	17640	10	702	714	4	108
R800	19690	12	807	809	4	126
R900	23041	18	906	881	6	169
R1000	24533	23	1005	1041	9	178
R1100	29685	23	1101	1091	9	211
R1200	31420	22	1202	1231	8	295
R1300	33120	25	1303	1243	9	386
R1400	36975	29	1401	1412	Not feasible	76
R1500	41453	34	1502	1497	13	451

and the selected hardware architecture. To compare the running times and the number of PEs for the Wildforce-XL and the Wildforce-XL1, we consider the R1500 netlist. For the Wildforce-XL, the RP algorithm produced 99 partitions within 863 seconds. On the other hand, only 21 partitions were produced by targeting the Wildforce-XL1 while the running time was reduced to 594 seconds. This result is presented in Table 4.9. Targeting the SLACC-1V board for the same R1500 netlist, the RP produced 11 partitions within 434 seconds. This shows how the performance of the RP algorithm depends on the selected hardware architecture.

To compare the running times and the number of PEs for the Wildforce-XL and the Wildforce-XL1, we consider the R1500 netlist. For the Wildforce-XL, the RP algorithm produced 99 partitions within 863 seconds. On the other hand, only 21 partitions were produced by targeting the Wildforce-XL1 while the running time was reduced to 594 seconds. This result is presented in Table 4.9. Targeting the SLACC-1V board for the same R1500 netlist, the RP produced 11 partitions within 434 seconds. This shows how the performance of the RP algorithm depends on the selected hardware architecture.

Referring to Table 4.11 the partitioning results are presented for the MSP board where the local RAM for each PE is organized as one bank of 512kX48 bits. Similar to HP algorithm, the RP algorithm failed to produce valid partitioning results for all netlists, which require multiple configurations of the board. Using the RAM as one bank of 512kX48 bits will limit the MSP board for a single board configuration only since both

RAM modules are always used to store the intermediate results. Organizing the RAM as two banks of 512kX24 bits can solve the problem. These results are shown in Table 4.10.

Referring to Table 4.13, the RP algorithm failed to partition the ATR, the R600, and the R1400 netlists when we targeted the SLAAC-1P board. The reason for that is the limited RAM bus size. This problem can be solved by targeting a hardware board with bigger the RAM bus size. Targeting the SLAAC-1V, the RP algorithm was able to produce feasible solutions for all netlists including the ATR, the R600, and the R1400 netlists. Table 4.12 shows the partitioning results for SLAAC-1V where the size of the local RAM bus is 36 bits.

4.1 RPL Algorithm

In this section we discuss the partitioning results for the RPL algorithm by using the netlists shown in Table 4.1. As mentioned in section 3.3, the RPL algorithm was developed to reduce the weaknesses of the HP and RP algorithms. The RPL starts the partitioning process with a levelization solution L of all nodes. Initially all nodes are in L and the first partition P_1 is empty. First, we start the process by moving n levels from L and put them into P_1 until P_1 has met the capacity constraint or the RAM module constraint. At this moment, the last level moved to P_1 is marked as L_i. Then the algorithm starts an optimization step by moving nodes across the marked level L_i and its successor level L_{i+1} until the rest of the constraints are satisfied. If the algorithm fails to find a valid solution, then we reduce the size of P_1 by removing the last level moved to P_1 and

the optimization step is repeated. We repeat this process by creating a new block P_2 and applying the same procedure to the remainder of L. The optimization step is based on the benefit function discussed in chapter 2. For each partition, only two levels L_i and its successor level L_{i+1} are involved in the optimization step. Therefore, the running time of the RPL algorithm and computation amount of the partitioning process is a function of the number of levels involved in the optimization step. In the same time, the number of levels involved in the optimization steps is a function of the partitions number required to implement the netlist application. For this reason, a hardware architecture with bigger PE capacity, bigger I/O count, and bigger RAM banks will reduce the partitioner running time significantly. In addition, minimizing the PEs number will reduce the data processing time on the targeted hardware. The last statement will be supported when we discuss the implementation of the ATR algorithm, section 4.5, on different hardware architectures.

Tables 4.14-4.19 show the partitioning results for the six hardware architectures. As expected, the partitions number produced and the running time for the partitioning process vary with the selected hardware board and the netlist size. For example, we consider the R1500 netlist and compare the performance of the RPL algorithm for both Wildforce-XL and SLACC-1V hardware boards. For the Wildforce-XL, 88 PEs are required to implement the R1500 netlist and the running time for the partitioner is 371 seconds. In the same time, the partitions number required to implement the R1500 netlist is reduced to 11 PEs by targeting the SLACC-1V board. The running time is reduced to 193 seconds. This big change is very significant for the data processing time on the

Table 4.14 Partitioning results for Wildforce-XL using RPL algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	20	2
M29	519	7	29	28	9	<1s
R300	7845	11	301	311	21	3
R400	10130	7	406	421	19	4
R500	12845	12	504	493	32	11
R600	15320	9	601	571	23	21
R700	17640	10	702	714	26	33
R800	19690	12	807	809	34	51
R900	23041	18	906	881	47	71
R1000	24533	23	1005	1041	55	101
R1100	29685	23	1101	1091	55	135
R1200	31420	22	1202	1231	63	178
R1300	33120	25	1303	1243	64	225
R1400	36975	29	1401	1412	77	293
R1500	41453	34	1502	1497	88	371

Table 4.15 Partitioning results for Wildforce-XL1 using RPL algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	7	1
M29	519	7	29	28	4	<1s
R300	7845	11	301	311	6	2
R400	10130	7	406	421	4	5
R500	12845	12	504	493	6	11
R600	15320	9	601	571	5	15
R700	17640	10	702	714	6	26
R800	19690	12	807	809	7	38
R900	23041	18	906	881	10	59
R1000	24533	23	1005	1041	13	71
R1100	29685	23	1101	1091	13	98
R1200	31420	22	1202	1231	13	127
R1300	33120	25	1303	1243	15	165
R1400	36975	29	1401	1412	17	199
R1500	41453	34	1502	1497	20	257

Table 4.16 Partitioning results for MSP1 board using RPL algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	12	1
M29	519	7	29	28	6	<1s
R300	7845	11	301	311	10	3
R400	10130	7	406	421	5	6
R500	12845	12	504	493	10	11
R600	15320	9	601	571	7	21
R700	17640	10	702	714	9	32
R800	19690	12	807	809	11	48
R900	23041	18	906	881	16	68
R1000	24533	23	1005	1041	21	76
R1100	29685	23	1101	1091	21	107
R1200	31420	22	1202	1231	21	133
R1300	33120	25	1303	1243	23	181
R1400	36975	29	1401	1412	27	209
R1500	41453	34	1502	1497	32	273

Table 4.17 Partitioning results for MSP2 board using RPL algorithm.

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	2	<1s
NVL	549	2	45	71	2	<1s
ATR	4885	14	101	234	Not feasible	<1s
M29	519	7	29	28	Not feasible	<1s
R300	7845	11	301	311	Not feasible	<1s
R400	10130	7	406	421	Not feasible	<1s
R500	12845	12	504	493	Not feasible	<1s
R600	15320	9	601	571	Not feasible	<1s
R700	17640	10	702	714	Not feasible	<1s
R800	19690	12	807	809	Not feasible	<1s
R900	23041	18	906	881	Not feasible	<1s
R1000	24533	23	1005	1041	Not feasible	<1s
R1100	29685	23	1101	1091	Not feasible	<1s
R1200	31420	22	1202	1231	Not feasible	<1s
R1300	33120	25	1303	1243	Not feasible	<1s
R1400	36975	29	1401	1412	Not feasible	<1s
R1500	41453	34	1502	1497	Not feasible	<1s

Table 4.18 Partitioning results for SLAAC-1V board using RPL algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	1	<1s
ATR	4885	14	101	234	4	1
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	1
R400	10130	7	406	421	2	3
R500	12845	12	504	493	4	8
R600	15320	9	601	571	3	11
R700	17640	10	702	714	4	20
R800	19690	12	807	809	4	31
R900	23041	18	906	881	6	52
R1000	24533	23	1005	1041	8	63
R1100	29685	23	1101	1091	8	87
R1200	31420	22	1202	1231	8	103
R1300	33120	25	1303	1243	8	127
R1400	36975	29	1401	1412	10	162
R1500	41453	34	1502	1497	11	193

Table 4.19 Partitioning results for SLAAC-1P board using RPL algorithm

Netlists	Size	RAM Modules	Nodes #	Nets #	Partitions #	Time(s)
Hipass Filter	458	2	17	48	1	<1s
NVL	549	2	45	71	Not feasible	<1s
ATR	4885	14	101	234	5	1
M29	519	7	29	28	2	<1s
R300	7845	11	301	311	4	1
R400	10130	7	406	421	2	3
R500	12845	12	504	493	4	9
R600	15320	9	601	571	Not feasible	6
R700	17640	10	702	714	4	23
R800	19690	12	807	809	4	37
R900	23041	18	906	881	6	57
R1000	24533	23	1005	1041	9	67
R1100	29685	23	1101	1091	8	93
R1200	31420	22	1202	1231	8	111
R1300	33120	25	1303	1243	9	136
R1400	36975	29	1401	1412	Not feasible	73
R1500	41453	34	1502	1497	11	203

targeted hardware. Reducing the PEs count required to implement a particular netlist can speed up the data processing significantly.

Referring to Table 4.17 the partitioning results are presented for the MSP board where the local RAM for each PE is organized as one bank of 512kX48 bits. Using the RAM as one bank of 512kX48 bits will limit the MSP board for a single board configuration only since both RAM modules are always used to store the intermediate results. Organizing the RAM as two banks of 512kX24 bits can solve the problem. These results are shown in Table 4.16.

Referring to Table 4.19, the partitioning results for the SLAAC-1P board are illustrated. As mentioned before, four RAM modules each of the size 256KX18 bits are available for each PE. In this case, the systolic bus between the PE and the local RAM is limited to 18 bits. For this reason, the RPL failed to find a valid partitioning result for the NVL, the R600, and R1400 netlists since these netlists need to access the local RAMs by more than 18 bits. Increasing the PE to RAM systolic bus width can solve this problem. Table 4.18 shows the partitioning results for SLAAC-1V where the size of the local RAM is 512KX36 bits and the RAM bus width is 36 bits.

4.4 Comparison Between RPL, HP, and RP Algorithms

In this section, we shall compare the three different partitioning approaches against each other. We will refer to the partitioning results shown in Tables 4.2-4.19.

The ATR netlist is given more attention in this discussion because of the following reasons:

1. The ATR is a very challenging netlist where a high number of RAM nodes are used.
2. There exists a manual partitioning result for the ATR netlist.
3. The ATR was implemented manually from CANTATA workspace to the Wildforce-XL.
4. The ATR netlist has a moderate size of nodes and nets so that the visualization of some problems for the partitioning process is possible.

The partitioning results shows that the HP and the RP approaches have difficulties in finding valid partitioning results for some of the netlists which utilize a high number of RAM access nodes. The problems arise when these RAM access nodes are close to each other in the hypergraph netlist. Therefore, the partitioning result is depending on the number of the RAM access nodes and on how these nodes are distributed in the netlist. The RAM access constraint is equivalent to the locking of a certain number of nodes in a netlist in which these nodes are prevented from moving freely across the cuts. This constraint is not affecting the movement of the particular node only but its neighbors nodes too.

Another major weakness of the HP and RP algorithms is the acyclic constraint. In addition to the original requirements of maintaining the acyclic constraint, the RAM

access constraint adds more difficulties to this constraint. This means, if one RAM node is locked in one partition, then the successors of this node are locked too because the movement of the successor nodes across a cut will violate the acyclic constraint. Therefore, adding these two constraints to the partitioning problem makes it a very challenging one.

To show the strength of our RPL algorithm, we illustrate the results of two different examples. In the first example, we consider a very challenging netlist, M29, shown in Figure 4.4, with 29 nodes and 28 nets. Among the 29 nodes, 7 nodes are utilized for RAM accessing. The shaded nodes represent the sources and destinations for this netlist. This netlist was generated manually where a valid partitioning result exists. At least seven PEs are needed for successful implementation of this netlist on the Wildforce-XL since the netlist utilizes seven RAM access modules. Referring to the Tables 4.2 and 4.8, both HP and RP algorithms failed to produce a valid partitioning result for the M29 netlist when we targeted the Wildforce-XL board. However, the RPL was able to produce a valid partitioning result for this particular netlist. The result is shown in Table 4.14. A total number of seven partitions were produced where nine levels were constructed during the partitioning process. In the M29 netlist, the source nodes (1,8,13) and the destination nodes (23,24,25,26) are used to access the RAMs for reading and writing the data. The source and destination nodes are placed closely to each other in the hypergraph. By removing the RAM access need for the nodes (8,23) and assigning a RAM access need for the nodes (12,27), the RP algorithm was able to produce a valid partitioning result while the HP algorithm still failed.

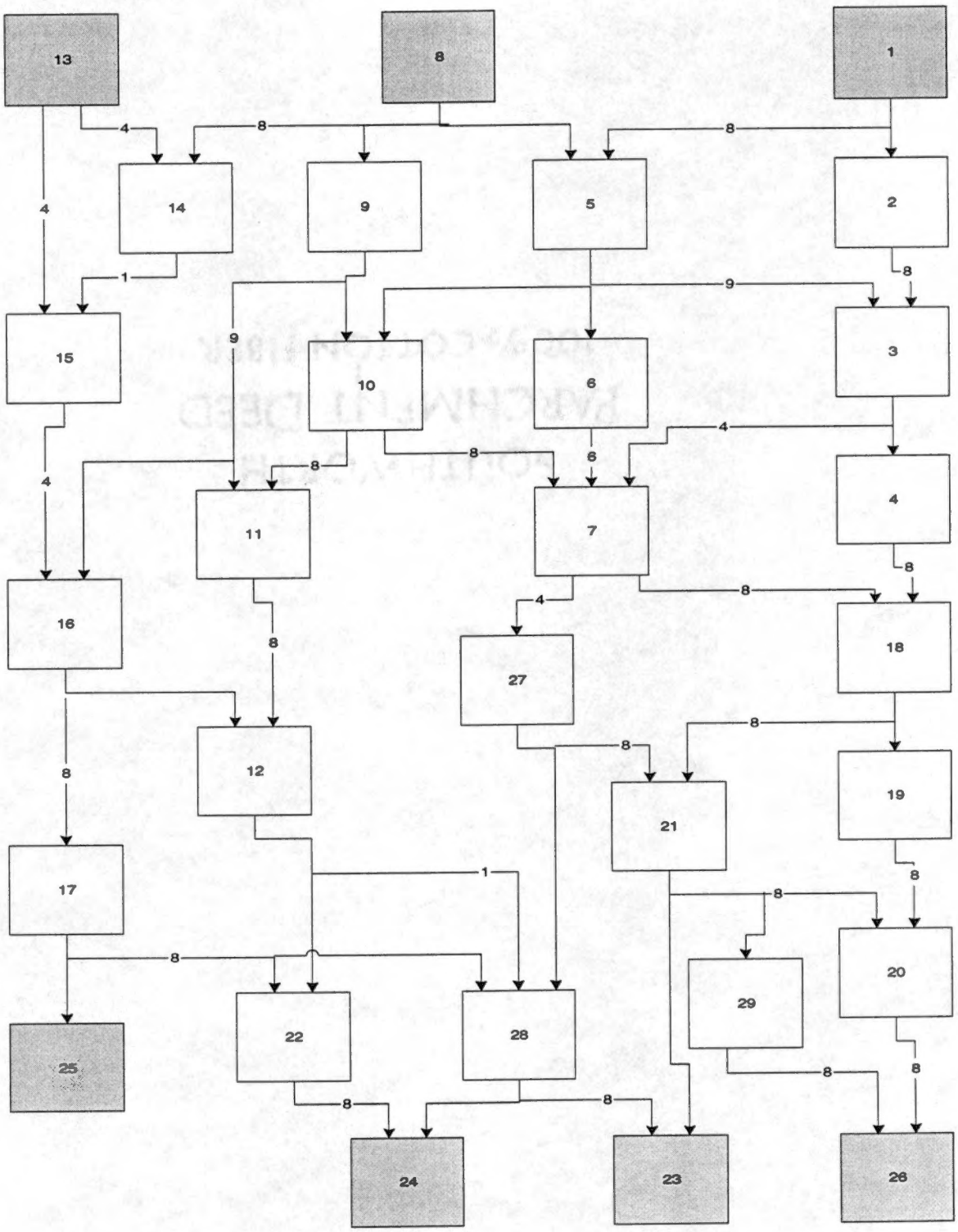


Figure 4.4 M29 Netlist

In other words, the distribution of the RAM nodes in the hypergraph was changed to affect the results produced by the RP and HP algorithms. The RPL was still able to produce a valid partitioning result for this netlist. Several experiments were conducted with the RAM nodes to investigate the performance of the algorithms. The small size of the M29 netlist enabled us to conduct these experiments and to observe the weaknesses of the HP and RP algorithms. The results of these experiments showed that the performance of the RP algorithm is determined by the RAM nodes distribution in the hypergraph. The results showed that the HP performance depends on the number of RAM nodes, the distribution of the RAM nodes in the hypergraph, and the hypergraph density. To obtain a valid partitioning result for the HP algorithm, we had to reduce the number of the RAM nodes and the weights for some of the hyper nets.

An application netlist similar to the M29 netlist can arise in the practice. Sometimes the user tries to collapse the design by using macros instead of cells. Of course this will improve the visualization and the manageability of the design, but it will reduce the granularity in the hypergraph. This means the nodes and nets numbers are reduced. If the design ends up having a structure similar to the M29 netlist and a high number of RAM nodes, then the partitioning results will be affected greatly by the RAM distribution and the density of the hypergraph. Therefore, the user must be aware of the size and I/O of the macros and the RAM distribution.

The automatic target recognition (ATR) application was automatically mapped from the Cantata workspace to the Wildforce platform recently. The ATR was first

implemented manually by the CHAMPION research group to assist in the development of function libraries and hardware for use in the CHAMPION system. The mapping techniques used were developed in such a way that they could serve as the basis for the automated system [4]. The ATR netlist was used to test our three different partitioning approaches. This netlist consists of 101 nodes and 234 hyper nets. Among the 101 nodes, 14 nodes are used for accessing the local RAMs. The ATR application utilizes a high number of macros where the sizes of these macros differ. A total number of 20 FPGAs was needed to implement the ATR netlist manually. Both RPL and RP algorithms were able to produce two different mappable partitioning results by targeting the Wildforce-XL, while the HP algorithm failed. The RPL result was identical to the manual partitioning result in terms of the FPGAs numbers. A total number of 20 partitions were produced where 51 levels were constructed during the partitioning process by using the RPL algorithm. The run time is 2 seconds. In the meantime, the RP algorithm produced 23 partitions within 9 seconds. Tables 4.20-4.21 show the ATR partitioning result for the RPL and RP algorithms respectively. Similar to the manual result, the RPL and RP results showed a poor utilization of the PEs capacity. The RAM access constraint and the macros sizes was the main reason for this poor utilization of the PEs capacity.

The run time for the RP algorithm is relatively longer when compared to the run time of the RPL and HP algorithms. In section 3.2, we mentioned that the run time of the RP algorithm is a function of the unlocked nodes in the current partition P and the remainder R. In the first phase of the RP algorithm, the entire netlist R_0 is partitioned into one feasible solution, P_1, that meets the constraints of the first PE on the board, and the

Table 4.20 RPL Partitioning results for the ATR by targeting the Wildforce-XL board.

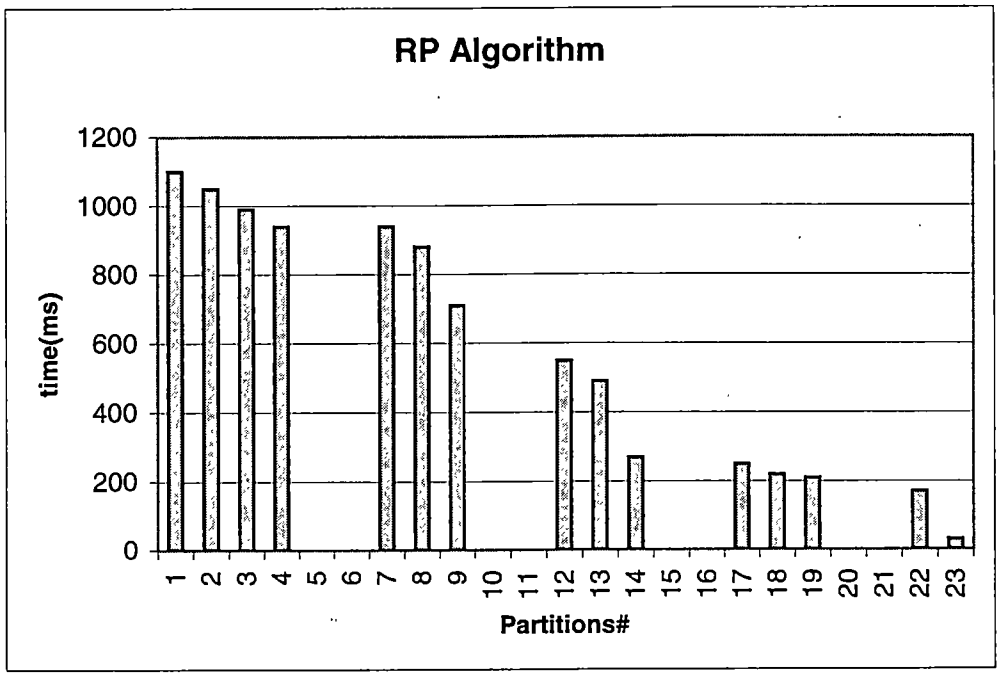
Partition Number	CLB Usage	I/O Count	Nodes #
1	741	24	9
2	462	15	5
3	482	32	12
4	424	28	10
5	0	-	0
6	0	-	0
7	367	28	8
8	345	26	8
9	389	25	7
10	0	-	0
11	0	-	0
12	367	29	9
13	367	28	8
14	389	25	7
15	0	-	0
16	0	-	0
17	389	19	9
18	32	30	1
19	80	11	7
20	52	11	1

Table 4.21 RP Partitioning results for the ATR by targeting the Wildforce-XL board.

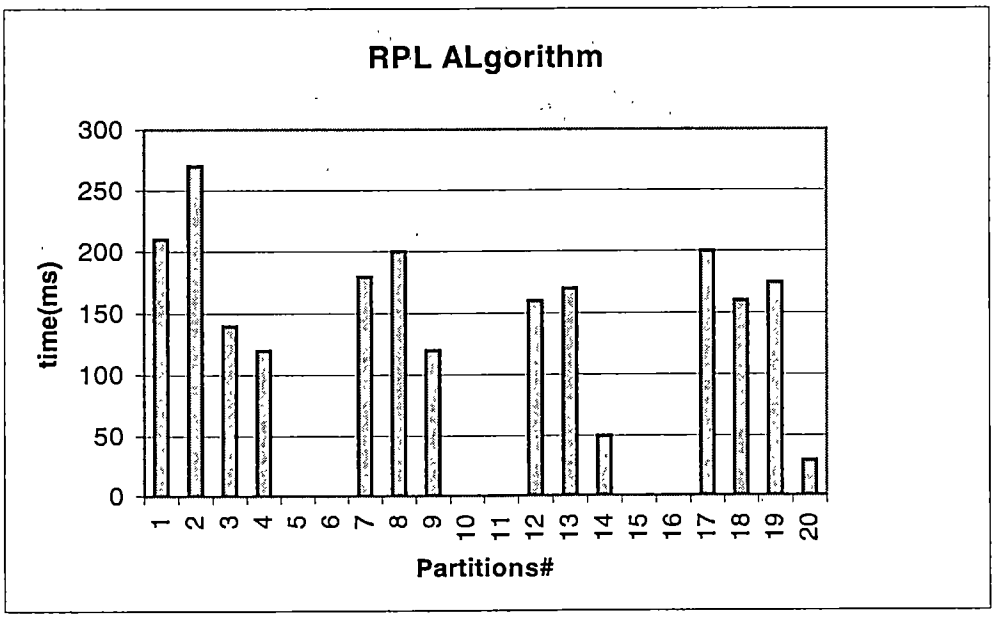
Partition Number	CLB Usage	I/O Count	Nodes #
1	718	33	3
2	400	35	5
3	434	29	11
4	471	9	14
5	0	-	0
6	0	-	0
7	362	10	9
8	367	9	8
9	367	11	8
10	0	-	0
11	0	-	0
12	367	9	8
13	371	9	7
14	367	7	8
15	0	-	0
16	0	-	0
17	421	19	10
18	56	19	1
19	32	30	1
20	0	-	0
21	0	-	0
22	80	11	7
23	52	11	1

remainder partition, R_1. During this phase, all the N nodes in the current partition P_1 and a remainder R_1 are unlocked and involved in the partitioning process. When the partitioner finds a feasible solution for the current partition P_1, the nodes in partition P_1 become locked. Therefore, during the partitioning process the run time decreases as the number of locked nodes increases. Figure 4.5a shows the RP run time during the partitioning of the ATR netlist. In section 4.1, it was mentioned that the run time of the RPL algorithm is a function of the number of levels involved in the partitioning process. For each partition, only two levels L_i and its successor level L_{i+1} are involved in the optimization step. Therefore, the run time for each partition is a function of the nodes size in L_i and its successor level L_{i+1}. Figure 4.5b shows the RPL run time during the partitioning of the ATR netlist.

The HP algorithm failed to produce a valid partitioning result for the ATR when we targeted the Wildforce-XL. The HP algorithm depends strongly on the node order in the linear ordering array. In general, the node order produced by a topological sort is not unique [42]. This situation arises when one node has no direct or indirect dependence on another and therefore they can be performed in either order. The HP algorithm made 10 attempts to partition the ATR netlist where each time the algorithm started with a different linear ordering solution. No valid result was found for any of these attempts. The HP result was affected greatly by the first few nodes in the netlist where three big size macros are used. The port count for these macros is relatively high when compared to other cells.



a. Run time of the RP algorithm



b. Run time of the RPL algorithm

Figure 4.5 Run time during the ATR partitioning process

The HP algorithm produced a non-feasible solution, shown in Table 4.22, after we relaxed the I/O count between the PEs from 36 to 50 for the Wildforce-XL. This result shows that only the first partition violates the I/O limitation. This partitioning result cannot be implemented on the Wildforce-XL since the I/O limitation is violated. The run time for the HP algorithm depends on the linear ordering array size, the number of the RAM access modules, how many times the algorithm rolls back, and the selected hardware architecture. Figure 4.6 shows the HP run time during the partitioning of the ATR netlist for the relaxed I/O count.

In several examples, the HP and RP algorithms used more PEs to implement one netlist when they are compared to RPL algorithm. For example, the RP algorithm uses 23 FPGAs to implement the ATR algorithm by targeting the Wildforce-XL while the RP algorithm uses only 20 FPGAs. The increased number of the PEs is related to the nature of the RP algorithm. The drawback inherent in the RP partitioning scheme, pointed out in [15], its effect of increasing the connectivity inside the remainder partition R. This effect makes the I/O constraint harder to meet during the final partitioning stages and causes more PEs to be used. In other words, the RP algorithm aims to minimize the cutset for the current partition. In addition, many previous works reported that the performance of partitioning schemes using the FM heuristic degrades as the number of nodes increase [43][44]. The RPL algorithm reduces these weaknesses by performing a preprocessing

Table 4.22 HP Partitioning results for the ATR by targeting the Wildforce-XL

Partition Number	CLB Usage	I/O Count	Nodes Count
1	1118	45	9
2	434	33	11
3	471	9	13
4	367	9	9
5	0	-	0
6	0	-	0
7	408	25	7
8	326	9	8
9	367	9	9
10	0	-	0
11	0	-	0
12	408	25	7
13	326	9	9
14	421	19	9
15	0	-	0
16	0	-	0
17	56	19	1
18	32	30	1
19	80	11	7
20	52	11	1

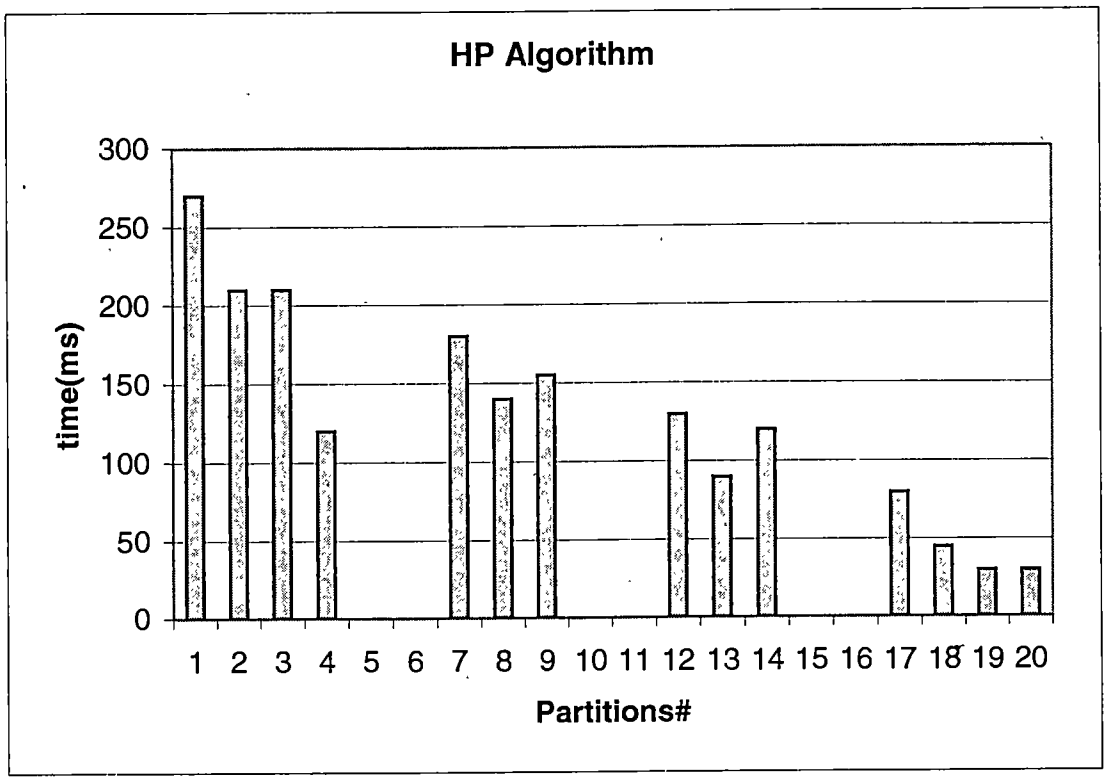


Figure 4.6 Run time of the HP algorithm during the ATR partitioning process

step (levelization) before starting the partitioning process. The preprocessing step reduces the number of nodes used during the partitioning stages significantly. This will increase the performance of the RPL algorithm and decrease the run time. In addition, the RPL algorithm strategy aims to maximize the capacity of the current partition while satisfying the I/O constraint.

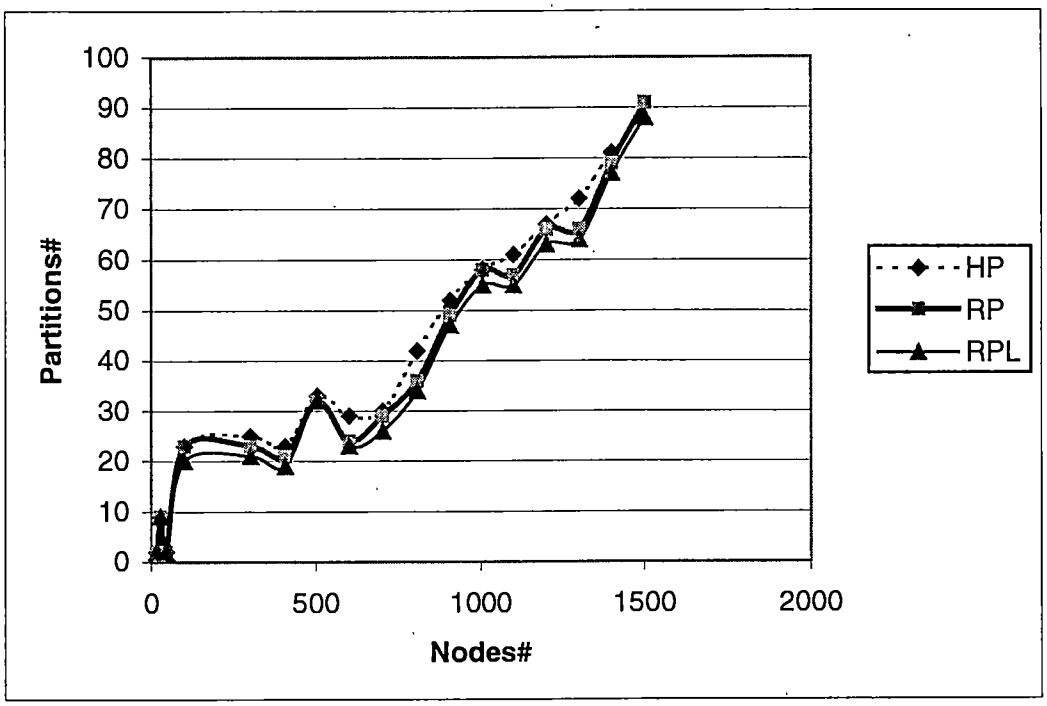
In the HP algorithm, nodes are moved to the current partition according to their order in the linear ordering array. This means, a node is moved to the current partition whether or not it is a good choice. The algorithm stops the movements when a feasible solution is found for the current partition. In other words, no more than one feasible solution for the current partition is recorded during the partitioning process. This is the drawback inherent

in the HP partitioning scheme that degrades the performance of the algorithm and tends to increase the number of the PEs used.

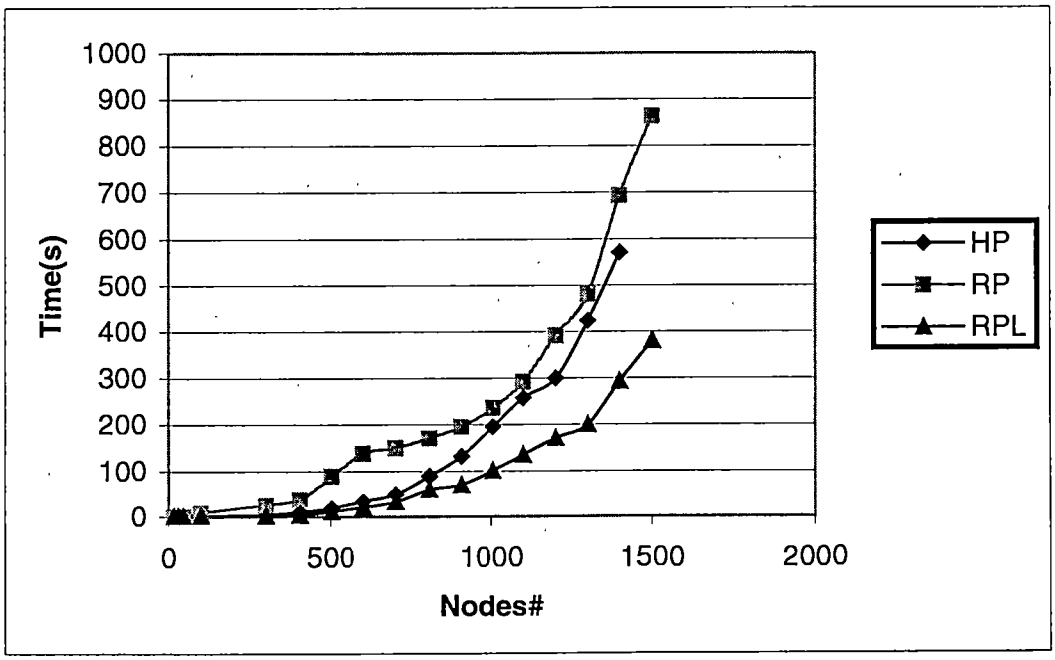
Figures 4.7a-4.7b compare the produced partitions and the run time on the netlists, shown in Table 4.1, for the Wildforce-XL. A similar behavior was obtained for the other hardware architectures. These experimental results demonstrate that the proposed approach, the RPL algorithm, achieves superior performance when compared to the RP and HP algorithms. Tables 4.23-28 show the combined results for the three partitioning algorithms.

4.5 Processing Time for Three Different Applications by Targeting Different Hardware Architectures.

This section discusses the hardware implementation of the automatic target recognition (ATR), the Hipass filter, and the NVL applications by targeting different hardware architectures. First we will give experimental results for the Wildforce-XL implementation. Second we will give an estimate for the processing times by targeting the Wildforce-XL1, the SLACC, and the MSP boards. The ATR application was automatically mapped from the Cantata workspace to the Wildforce-XL platform by using two different partitioning results. Early in this research project, the ATR was implemented manually by the CHAMPION research group to assist in the development of function libraries and hardware for use in the CHAMPION system.



a. Produced partitions by targeting the Wildforce-XL



b. Run time of the HP, RP, RPL algorithms by targeting the Wildforce-XL

Figure 4.7 Partitioning results for the Wildforce-XL

Table 4.23 Partitioning Results for the Wildforce-XL Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	2	<1s	2	<1s	2	<1s
NVL	2	<1s	2	<1s	2	<1s
ATR	Not feasible	2	23	9	20	2
M29	Not feasible	<1s	Not feasible	2	9	<1s
R300	25	4	23	26	21	3
R400	23	10	21	39	19	4
R500	33	19	32	89	32	11
R600	29	34	24	138	23	21
R700	Not feasible	761	Not feasible	61	26	33
R800	42	89	36	171	34	51
R900	52	132	49	195	47	71
R1000	58	184	58	236	55	101
R1100	61	257	57	292	55	135
R1200	67	336	66	392	63	178
R1300	72	424	66	501	64	225
R1400	81	571	79	693	77	293
R1500	Not feasible	1129	91	863	88	371

Table 4.24 Partitioning Results for the Wildforce-XL1 Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	1	<1s	1	<1s	1	<1s
NVL	1	<1s	1	<1s	1	<1s
ATR	7	2	7	6	7	1
M29	4	<1s	4	<1s	4	<1s
R300	6	3	6	17	6	2
R400	4	8	4	31	4	5
R500	6	13	7	62	6	11
R600	5	25	5	101	5	15
R700	6	41	7	125	6	26
R800	7	71	7	143	7	38
R900	10	107	10	177	10	59
R1000	13	161	13	211	13	71
R1100	13	221	13	276	13	98
R1200	13	291	13	314	13	127
R1300	15	383	15	461	15	165
R1400	18	431	17	515	17	199
R1500	21	521	21	594	20	257

Table 4.25 Partitioning Results for the MSP1 Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	1	<1s	1	<1s	1	<1s
NVL	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
ATR	14	1	14	7	12	1
M29	6	<1s	6	<1s	6	<1s
R300	10	3	10	21	10	3
R400	5	9	5	35	5	6
R500	10	15	10	71	10	11
R600	7	29	7	113	7	21
R700	9	47	9	139	9	32
R800	11	81	11	151	11	48
R900	17	107	16	185	16	68
R1000	22	172	21	222	21	76
R1100	21	234	21	281	21	107
R1200	21	313	21	331	21	133
R1300	24	401	24	471	23	181
R1400	27	453	27	523	27	209
R1500	34	503	33	604	32	273

Table 4.26 Partitioning Results for the MSP2 Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	2	<1s	2	<1s	2	<1s
NVL	2	<1s	2	<1s	2	<1s
ATR	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
M29	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R300	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R400	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R500	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R600	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R700	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R800	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R900	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1000	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1100	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1200	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1300	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1400	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
R1500	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s

Table 4.27 Partitioning Results for the SLAAC-1V Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	1	<1s	1	<1s	1	<1s
NVL	1	<1s	1	<1s	1	<1s
ATR	4	1	5	2	4	1
M29	2	<1s	2	<1s	2	<1s
R300	4	2	4	11	4	1
R400	2	6	2	17	2	3
R500	4	9	4	44	4	8
R600	3	19	3	89	3	11
R700	4	37	4	101	4	20
R800	4	62	4	121	4	31
R900	6	89	6	152	6	52
R1000	8	131	8	174	8	63
R1100	8	192	8	205	8	87
R1200	8	216	8	281	8	103
R1300	8	289	8	379	8	127
R1400	10	305	10	411	10	162
R1500	11	351	11	434	11	193

Table 4.28 Partitioning Results for the SLAAC-1P Board

Netlists	HP Algorithm		RP Algorithm		RPL Algorithm	
	Partitions #	Time(s)	Partitions #	Time(s)	Partitions #	Time(s)
Hipass Filter	1	<1s	1	<1s	1	<1s
NVL	Not feasible	<1s	Not feasible	<1s	Not feasible	<1s
ATR	Not feasible	11	5	1	5	1
M29	2	<1s	2	<1s	2	<1s
R300	4	13	4	1	4	1
R400	2	18	2	3	2	3
R500	4	47	4	9	4	9
R600	Not feasible	31	Not feasible	6	Not feasible	6
R700	4	108	4	23	4	23
R800	4	126	4	37	4	37
R900	6	169	6	57	6	57
R1000	9	178	9	67	9	67
R1100	9	211	8	93	8	93
R1200	8	295	8	111	8	111
R1300	9	386	9	136	9	136
R1400	Not feasible	76	Not feasible	73	Not feasible	73
R1500	13	451	11	203	11	203

For the manual implementation, the ATR application was partitioned in such a way so that an image of the size 256x256 can be processed and recorded in the local RAM. In this case, the total memory required to store an image is 65kx8 bits. The local RAM on the Wildforce-XL is limited to 32kx32 bits so that a data amount of 65k bytes cannot be stored without memory management. To overcome this problem, each four words (4x8 bits) were packed together and stored in one address. The ATR was partitioned for multiple board configurations first where the cutset size was limited to 8. Then each board configuration was partitioned further to fit into five FPGAs.

In our partitioning strategy, we attempted to partition for multiple board configurations first. By adding this constraint to the partitioning strategy, the partitioning problem became very complicated. None of the partitioning algorithms could handle this constraint. Therefore, this constraint was relaxed to ease the partitioning problem and to allow a cutset size of 32 between multiple board configurations. In this case, each address in the local RAM is used to store one word of 0 to 32 bits only. By doing this, an image of the size 256x256 can no longer fit in the RAM. To overcome this problem, the image size was reduced to 128x128 so that the memory needed to store an image is 16kx8 bits.

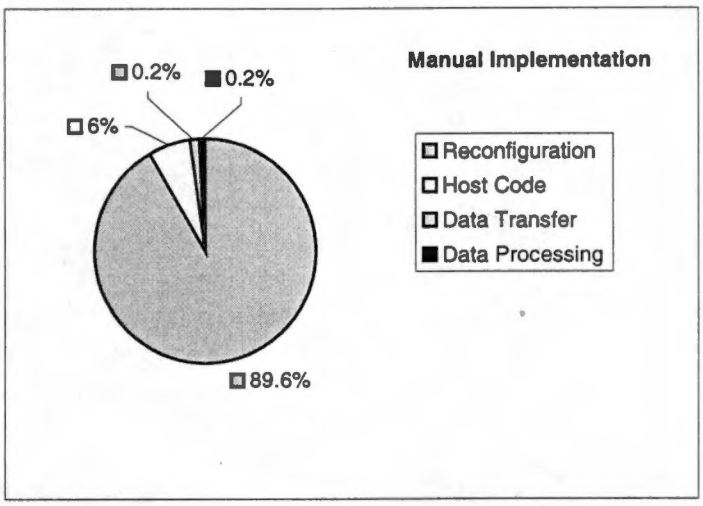
To compare the automated mapping against the manual mapping, we had to rerun the manual implementation for reduced image size (128x128). The average of 10 runs gave the following values, shown in Table 4.29, for the automated and manual mapping:

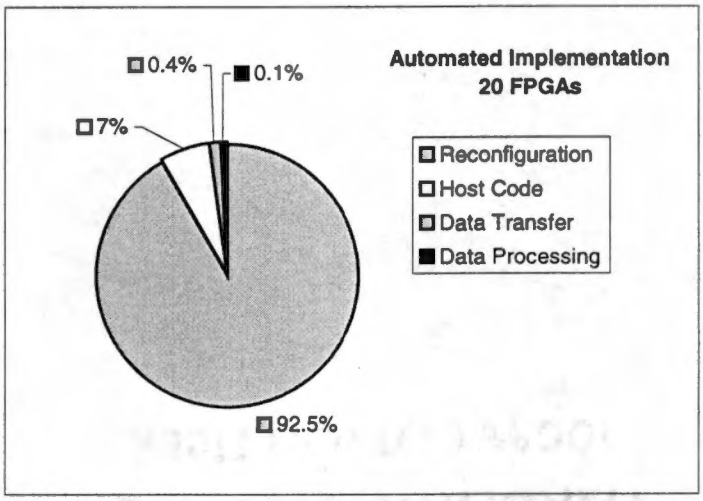
Table 4.29 Processing time for the ATR algorithm

Processing time	Manual Implementation 20 FPGAs	Automated Implementation 20 FPGAs	Automated Implementation 23 FPGAs
Board configuration time	5125 ms	9147 ms	11010 ms
Host code run time + Wildforce setup time	573 ms	695 ms	715 ms
Data transfer time	10 ms	40 ms	41 ms
Hardware execution time	10 ms	10 ms	11 ms
Total time to process one image	**5718 ms**	**9890 ms**	**11777 ms**

These results are also shown graphically in Figure 4.8. For both manual and automated implementation the process time of one image is dominated greatly by the board configuration time. In the manual implementation, some of the FPGAs ended up with the same glyphs. This means that not every FPGA needed to be reprogrammed every time [4]. Only 11 FPGAs needed to be reprogrammed. Because of this, the total processing time for the manual implementation was reduced significantly, nearly to the half, when compared with the automated mapping.

The partitioning process cannot handle the situation where some of the FPGAs might have the same glyphs. This is a very challenging problem that will have to be addressed in future research. In section 4.1 it was mentioned that minimizing the number





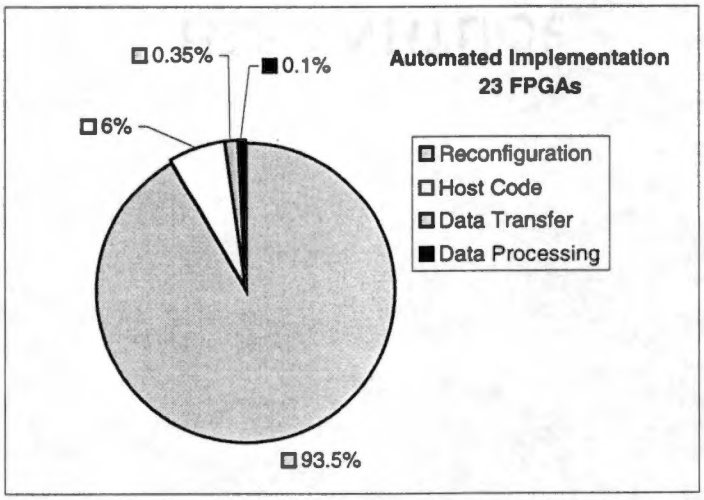


Figure 4.8 Image processing time for the ATR application

of FPGAs used will reduce the total processing time for one application. Referring to the processing time results for the two different automated implementations shown above, it is obvious that the 20 FPGA implementation has a better performance when compared to the 23 FPGA implementation. The 20 FPGA implementation needed four board configurations. At the other side, five board configurations were needed for the 23 FPGAs implementation.

The total processing time of one image can be further improved by targeting larger hardware architecture. A hardware board with more PE capacity and bigger RAMs, which requires only a single board configuration for a particular application, will reduce the processing time significantly be removing the board reconfiguration time and by reducing the host code and the data transfer times. The ATR application has unusual high number of operations requiring RAM access. Some of the delay glyphs in the ATR algorithm required external implementation by accessing the local RAMs. The ATR algorithm needs only three RAM modules to transfer the input and output data. The rest of the RAM modules were needed to transfer data between multiple configurations and to implement delay glyphs. These delay glyphs could not be implemented internally because of the limited PEs resources on the Wildforce-XL.

Targeting the SLAAC-1V board, the ATR algorithm can be implemented by using a single board configuration only. The SLAAC-1V board has enough resources, one million logic gates and 4 local RAMs for each FPGA, to implement the ATR application by using two FPGAs only. In this case, a total number of 8 local RAMs will

be available to implement some of the delay glyphs and the three RAM modules needed to transfer the input and output data. Some of the delay glyphs can be implemented internally since we have enough resources on the SLAAC-1V board and these delay glyphs require small delay values. The remaining 5 RAMs can still be utilized for delay glyphs that require a big delay value. In the ATR application, four delay glyphs must be implemented externally since the required delay value for these glyphs is 65550 cycles. Referring to the manual implementation results above, if the reconfiguration time could be eliminated, the time to process one image would be dominated by the time needed to run the host code plus the time to setup the Wildforce-XL. The average time for running the host code plus the Wildforce setup 661ms. The data transfer time is 10ms. The SLAAC-1V board would need to transfer the processed data three times only, as compared to 11 times for the Wildforce-XL. This would reduce the data transfer time for the SLAAC-1V board to a negligible amount. If we assume that the SLAAC-1V board would need 661ms to setup the board and to run the host code and 10ms for hardware execution, the SLACC-1V board implementation would be 9 times faster than the manual Wildforce-XL implementation, 15 times faster than the 20 FPGA automated implementation, and 18 times faster than the 23 FPGA automated implementation. The same discussion is valid for the SLACC-1P board.

Targeting the MSP board, the ATR algorithm can be implemented by using three board configurations if the local RAM for each FPGA is organized as two banks of 512kX24 bits. In this case, a total number of 6 FPGAs and 12 local RAMs will be available to implement some of the delay glyphs and the seven RAM modules needed to

transfer the input and output data and data between multiple board configurations. The reconfiguration time cannot be eliminated since there is a need of three board configurations. The time to process one image would be dominated by the time needed to reconfigure the MSP board. This means the MSP board will not reduce the processing time to a significant amount when compared to SLAAC-1V board. The MSP board can still improve the performance if it is compared to the Wildforce-XL implementation. Referring to the above Wildforce-XL results, the average total time to process one image for each board configuration of the manual Wildforce implementation is 1430ms. If we assume that the MSP implementation would need this time for each board configuration, the MSP implementation would be1.4 times faster than the manual implementation and 2.7 times faster than the 23 FPGAs implementation on the Wildforce-XL.

The Hipass filter application was automatically implemented on the Wildforce-XL. The application utilizes two RAM glyphs. The average of 10 runs gave a total processing time of 521 ms where the hardware execution time for one image is only 3ms. The total processing time of one image is dominated greatly by the time needed to setup the Wildforce-XL and to run the host code. The total processing time can be reduced by a small amount ΔT by targeting bigger hardware architecture, such as the SLAAC-1V board, where only one PE would be needed to implement the Hipass filter. In this case, ΔT would include the delay time between two PEs, the delay time between the core and the PE interface circuit for two PEs, and the reduced time for the board setup and the host code run time since only one PE would be used.

Based on the above discussion the total processing time for the NVL algorithm can be estimated by targeting different hardware architectures. The NVL algorithm utilizes two RAM access nodes so that two PEs is needed by targeting the Wildforce-XL and only one PE is needed for the SLAAC, the MSP, the Wildforce-XL1 boards. The hardware execution time can be estimated by counting the number of cycles needed to process one image. For one image of the size 640x480 and a running frequency of 25MHz the hardware execution time is 368ms. Targeting the Wildforce-XL and adopting the time needed to setup the board and the host code run time from the Hipass filter, the NVL total processing time for one image is 885ms. The total processing time can be reduced by a small amount ΔT by targeting the other hardware architectures.

The above discussion shows that selecting the proper hardware architecture will improve the performance for a particular application greatly. Targeting a hardware structure with more RAMs for each FPGA is preferred for an application utilizing a high number of operations requiring RAM access. These results show that both the partitioning process and the hardware architecture determine the performance rate for a particular application. Table 4.30 summarizes the total processing time, given in ms, for three different applications by targeting different hardware architectures.

Table 4.30 Processing time for different hardware architectures

Applications	Wildforce -XL	Wildforce -XL1	MSP 2 RAM Blocks	MSP 1 RAM Block	SLAAC-1P	SLAAC-1V
ATR - Manual/20PEs/23PEs	5718/9890 /11777	671	4290	-	671	671
Hipass Filter	521	521-ΔT	521-ΔT	521-ΔT	521-ΔT	521-ΔT
NVL	885	885-ΔT	-	885	-	885-ΔT

5. Conclusion

The goal of this research was to develop and investigate three different partitioning algorithms and determine the one that has a higher performance rate. The research presented in this thesis accomplished these goals. In the first and the second approaches, we discussed the development and implementation of two existing algorithms. The first approach is a hierarchical partitioning method based on topological ordering (HP). The second approach is a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic (RP). We extended these algorithms to handle the CHAMPION partitioning constraints by targeting different hardware architectures. We also introduced a new recursive partitioning method based on topological ordering and levelization (RPL). In addition to handling the partitioning constraints, the new approach efficiently addresses the problem of minimizing the number of PEs used to implement a particular application and overcoming the weaknesses of the HP and RP algorithms. The hardware architectures considered in this research includes the Wildforce-XL, the SLAAC-1V, the SLAAC-1P, and the MSP boards.

The partitioning strategy is based on the following constraints:

1. Capacity per partition
2. Number of I/O pins per partition
3. Each partition can only have one RAM access module

4. Input module and output module must be placed in the first partition and in the last partition.
5. Temporal partitioning constraint. For multiple board configurations, storage of intermediate results between board configurations is needed.
6. Maintaining the acyclic constraint so that all edges point the same way (from left to right).

One of the major goals in this research is to target different hardware architectures. The partitioning algorithms were extended to be dynamic so that the partitioner will read the particular hardware structure data from a partitioning configuration file. This will enable the user to change some specific information in the configuration file instead of working inside the partitioner and gives the user the flexibility to switch from one hardware architecture to another one.

The performances of the three different partitioning approaches were compared against each other. Comparisons between these algorithms were made on a variety of netlists. The comparison was based on the number of PE required to implement the netlist on the targeted hardware board and the running time for the partitioner. Practical netlists and random generated netlists were used to investigate the three partitioning algorithms.

In this research we considered a subset of netlists for CHAMPION applications, which were implemented automatically from the Cantata workspace to the Wildforce-XL platform. This subset of netlists includes the Hipass Filter, the NVL Round0, and the

ATR applications. The automatic target recognition (ATR) was given a special attention in this research because this netlist is a very challenging one. The ATR was first implemented manually by the CHAMPION research group to assist in the development of function libraries and hardware for use in the CHAMPION system. The ATR application has an unusual high number of operations requiring RAM access. This number of RAMs made the ATR netlist a very challenging one.

To the best of our knowledge, there exist no benchmarks that represent our partitioning constraints. For this reason, a random netlist generator to produce benchmarks was developed in this research. By considering large netlists, we were able to investigate the performance of the HP, the RP, and the RPL algorithms. The netlists R300-R1500 were produced randomly. In addition, a very challenging netlist M29, which utilizes 7 RAM modules, was generated manually to challenge the three partitioning algorithms.

We started the evaluation process of the partitioning algorithms with netlists that were known to have a partitioning solution. The RP and HP algorithms had difficulties in producing valid partitioning results for netlists which utilized a high percentage of RAM access modules. For example, the ATR and M29 utilize a high number of RAM modules. The RPL algorithm produced 20 partitions for the ATR netlist within 2 seconds, while the RP algorithm produced 23 partitions within 9 seconds by targeting the Wildforce-XL. The HP algorithm failed to partition the ATR netlist. Both RP and HP algorithms failed

to partition the M29 netlist by targeting the Wildforce-XL board. However, the RPL was able to partition the M29 netlist.

In several examples, the number of partitions produced by RPL algorithm was less than the number produced by the other algorithms. The RPL algorithm uses a level construction step, denoted as a preprocessing step, to reduce the weaknesses of the HP and RP algorithms and to minimize the number of PEs used. Since the RAM access constraint is a very challenging one, the preprocessing step was able to solve any conflicts associated with this constraint before moving to the partitioning step. The RPL algorithm creates partitions by moving levels instead of nodes to the current partition. Because of this, the run time for the RPL was faster when compared to the HP and RP algorithms.

In this research we considered several hardware architectures which includes the Wildforce-XL, the MSP, and the SLAAC boards. The partitioning results for the run time and the number of PEs used varied with the selected hardware architecture. For the three partitioning algorithms, the run time and the number of partitions were reduced significantly by targeting hardware boards with bigger I/O, bigger PE capacity, and bigger RAMs. For example, the RPL algorithm produced 63 partitions for the R1200 netlist within 178 seconds by targeting the Wildforce-XL board. However, the algorithm produced 8 partitions within 103 seconds for the same netlist when we targeted the SLAAC-1V board. In general, this big change reduces the computation time for the commercial tools and the data processing time for a particular application.

Minimizing the number of the PEs used was one of the primary goals in this research. To show the importance of this point, the ATR application was implemented automatically from the CANATA workspace to the Wildforce-XL board for the two different partitioning results produced by the RPL and the RP algorithms. The first partitioning result used 20 FPGAs. The second partitioning result used 23 FPGAs. The resulted total time to process one image for the 20 FPGAs and the 23 FPGAs implementations is 9890ms and 11777ms respectively. The total time to process one image for the manual implementation was 5718ms. The manual implementation used 20 FPGAs where only 11 FPGAs needed to be reconfigured. Because of this, the manual implementation was faster than the automated implementation. In this research, we also discussed the implementation of the ATR application by targeting the SLAAC and the MSP boards. The estimated total time to process one image on the SLAAC board would be 37 times faster than the manual implementation and 77 times faster than the 23 FPGAs implementation on the Wildforce-XL. At the other side, the MSP board would be 1.4 times faster than the manual implementation and 2.7 times faster than the 23 FPGAs implementation on the Wildforce-XL. These results showed that both the partitioning process and the hardware architecture determine the performance rate for a particular application.

In the CHAMPION design flow each sub-netlist resulting from the partitioning step must be converted to a structural VHDL file representing the hardware resources desired for each FPGA. A PERL script file was written to generate the structural VHDL

file and to assign the communication signals between the PEs. The script file identifies the glyphs used in the sub-netlist, the connections between glyphs, and the connections between glyphs and the other FPGAs.

The host code, which is used to communicate with the board, was automated to a certain level. The host code uses a set of function calls provided by the manufacture of the ACS. In addition, the automated host code uses a configuration file produced by a PERL script, which accesses the resulting sub-netlists and extracts the configuration data. This configuration file is used by the host code to determine the number of configurations, the name and location for each programming bit file, if a specific PE needs to access the SRAM, and where to write the result after each configuration.

There are several improvements that can be investigated in the future to enhance the solution quality produced by the partitioning process. The effect of using look-ahead schemes in the partitioning process can be explored. The look-ahead method was used in many previous works where enhancements were reported. In this case, the partitioning process can be improved by defining a gain vector for each node. Using the gain vector allows to swap nodes that reduces the mean cuts in the resulting partitions.

The partitioning process investigated in this research could not handle the situation where some of the FPGAs might have the same glyphs. This is a very challenging problem that will have to be addressed in a future research. This will improve

the data processing time by reducing the number of the reconfigure PEs. It is not clear to the author what form the solution to this problem may take.

The RAM access constraint was the most challenging one for the partitioning process and the performance of applications on the hardware boards. This constraint can be relaxed to a certain level by implementing some of the used RAM modules internally. This will require hardware architectures with big resources. Some of the RAM modules, which are used to transfer the input and output data and between multiple board configurations, need to be implemented externally. In several examples, the I/O constraint between multiple board configurations could not be met during the partitioning process. One way to overcome this problem is to use hardware architectures with bigger RAM resources. Another way is to multiplex data between a PE and the local RAM whenever the I/O constraints cannot be met. This will require extra implementation of multiplexing glyphs.

References

[1] C. Fiduccia and R. Mattheyses, "*A Linear Time Heuristic for Improving Network Partitions*", Proc. Of the 19th Design Automation Conference, pp. 175-181, 1982.

[2] B. Kernighan and S. Lin, "*An Efficient Heuristic Procedure for Partitioning Graphs*", Bell System Technical Journal, v.49, n.2, pp. 291-307, February 1970.

[3] S. M. Sait and H. Youssef, VLSI Physical Design Automation, McGraw-Hill, pp. 43-53, 1995.

[4] B. Levine, *A System for the Implementation of Image Processing Algorithms on Configurable Computing Hardware*, University of Tennessee, Master Thesis, 1999.

[5] S. Natarajan, *Development and Verification of Library Cells for Reconfigurable Logic*, University of Tennessee, Master Thesis, 1999.

[6] N. A. Sherwani, Algorithms for VLSI Physical Design Automation, Kluwer Academic Publishers, pp. 168-190, 1999.

[7] D. Schweikert and B. Kernighan, "*A Proper Model for the Partitioning of Electrical Circuits*", Proc. Of 9th Design Automation Conference, pp. 57-62, 1972.

[8] B. Stanley, *Hierarchical Multiway Partitioning Strategy with Hardware Emulator Architecture Intelligence*, Georgia Institute of Technology. Ph.D. Dissertation, 1997.

[9] S. Brown , J. Rose, and Z. G. Vranesic, "*A Detailed Router for Field-Programmable Gate Arrays*", IEEE tansaction on computer-Aided Design, Vol. 11, No.5, pp. 629-628, May 1992.

[10] J. Spillane and H. Owen, "*Temporal Partitioning for Partially Reconfigurable Field-Programmable Gate Arrays*", Reconfigurable Architectures Workshop, 1998 in IPPS/SPDP'98.

[11] B. Krisnamurthy, "*An Improved Min-Cut Algorithm for Partitioning VLSI Networks*", IEEE Transactions on Computer, v. C-33, n. 5, pp.438-446, May 1984.

[12] S. Kirkpatrick, C. Gellat, Jr., M. Veechi, "*Optimization by Simulated Annealing*", Science, v. 220, pp. 671-680, May 1983.

[13]. R.Kuznar, F. Berglez, and K. Kozminski, "*Partitioning Digital Circuits for Implementation in Multiple FPGA ICs*", Technical Report TR93-03, MCNC, Research Triangle Park, NC, March 1993.

[14] S. Brown and Z. Vranesic, Fundamentals of Digital Logic With VHDL Design, McGraw-Hill, pp. 81-100, 2000.

[15] R.Kuznar , "*PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Netlists*", International Conference on Computer Aided Design, San Jose, CA, November 5-9, 1995.

[16] L. Sanchis, "*Multiple-Way Network Partitioning*" IEEE Transaction on Computers, Vol. 38, No. 1, pp.62-81, 1989.

[17] Zycad, "*K-Way Algorithm*", Concept Silicon Reference Manual – Paradigm RP, pp. 116-119, November 1994.

[18] S. Hauck and G. Borriello, "*Logic Partition Ordering for Multi-FPGA Systems*", International Symposium on Field Programmable Gate Arrays, pp. 1-7, 1995.

[19] R. Burra and D. Bhatia, "*Timing Driven Multi-FPGA Board Partitioning*", Proceedings of IEEE International Conference on VLSI Design, Chennai, India, January 1998.

[20] M. Karthikeya and G. Purna and D. Bhatia, "*Partitioning in Time: A Paradigm for Reconfigurable Computing*", Proceedings of International Conference on Computer Design (ICCD 98), pp. 340-347, Austin, October 1998.

[21] K.M. Gajjalapurna and D. Bhatia, "*Temporal Partitioning and Scheduling for Reconfigurable Computers*", IEEE International Conference on FPGAs in Custom Computing Machines (FCCM 98), pp. 329-330, Napa Valley, April 1998.

[22] G. Tumbush and D. Bhatia, "*K-way Partitioning under Timing, Pins, and Area Constraints*", Proceedings of International Conference on Computer Design, October 1997.

[23] C. Lee and T. Yang and Y. F. Wang, "*Partitioning and Scheduling for Parallel Image Processing Operations*", Proc. IEEE/ACM International Conference on Computer-Aided Design, pp. 497-504, 1998.

[24] C. Lee and T. Yang and Y. F. Wang, "*Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines*", Journal of Parallel and Distributed Computing, 45(1): 29-45, 1997.

[26] C. Shetters, *Scheduling Task Chains on an Array of Reconfigurable FPGAs*, University of Tennessee, Master Thesis, 1999.

[27] S. Hauck, *Multi-FPGA Systems*, University of Washington, Ph.D. Dissertation, 1995.

[28] M. J. S. Smith, Application-Specific Integrated Circuits, Addison Wesley, 1997.

[29] N. Woo and J. Kim, "*An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation*", Proc. Of 30th Design Automation Conference, pp. 202-207, 1993.

[30] C. H. Gebotys, "*An Optimal methodology of Synthesis of DSP Multichip Architectures*", Journal of VLSI Signal Processing, v11, p9-19 1995.

[31] A. Athanas and A. Abbot, "*Real-Time Image Processing on a Custom Computing Platform,*" IEEE Computer, vol. 28, pp.16-24, Feb. 1995.

[32] H. Liu and D. F. Wong, "*Network Flow Based Circuit Partitioning for Time-Multiplexed FPGAs,*" Proc. IEEE/ACM International Conference on Computer-Aided Design, pp. 497-504, 1998.

[33] D. Gajski, N. Dutt, A. Wu, and S. Lin, High-Level Synthesis: Introduction to Chip and System Design: Kluwer Academic Publishers, 1992.

[34] C. T. Hwang, J. H. Lee, and H. Yu-Chin, "*A Formal Approach to the Scheduling Problem in High Level Synthesis,*" IEEE Transactions on Computer Aided Design, vol. 10, pp. 464-475, 1991.

[35] D. Chang and M. Marek-Sadowska, "*Buffer Minimization and Time-Multiplexed I/O on Dynamically Reconfigurable FPGAs,*" Proc. ACM International Symposium on FPGAs, pp. 142-148, 1997.

[36] F. Vahid and D.D. Gajski, "*Clustering for Improved System-Level Functional Partitioning,*" International Symposium on System Synthesis, pp. 28-33, September 1995.

[37] F. Vahid and T.D.M. Le, "*Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning,*" Kluwer Journal on Design Automation of Embedded Systems, Vol. 2, No. 2, pp. 237-261, March 1997.

[38] F. Vahid, "*Techniques for Minimizing and Balancing I/O during Functional Partitioning,*" IEEE Transactions on CAD, Vol. 18, No. 1, pp. 69-75 January 1999.

[39] D. Gajski, "*Specification Partitioning for System Design*," Design Automation Conference, pp. 219-224, June 1992.

[40] S. Narayan, F. Vahid and D.D. Gajski, "*System Level Specification and Synthesis*," International Conference on VLSI Design, January 1992.

[41] T.D.M. Le and Y.C. Hsu, "*A Comparison of Functional and Structural Partitioning*," International Symposium on System Synthesis, pp. 121-126, November 1996.

[42] R. Sedgewick, Algorithms in C++, Addison Wesley Publishing Company, pp. 479-481, 1992.

[43] L. Hagen, *Circuit Partitioning*, PhD thesis, UCLA, 1994.

[44] L. Hagen and A. B. Kahng, " *Combining Problem Reduction and adaptive Multi-Start: A New Technique for Superior Iterative Partitioning*," IEEE Trans. Computer-Aided Design, 1996.

VITA

Nabil Kerkiz was born in Jabalia, Gaza Strip on May 24th, 1965. He lived in Gaza Strip until 1985, when he moved to Germany. He finished his high school education at El-Faluja High School in Gaza Strip and entered the University des Saarlandes for his undergraduate studies in Electrical Engineering. Upon completion of his Diploma of EE in 1994, he returned to Gaza Strip. While there, he worked for Hejazy Company as a Hardware Design Engineer. In 1996 , he moved to the USA to attend the Southern Illinois University at Carbondale. He received his Master of Science in electrical engineering in December of 1997. In January of 1998, he moved to Tennessee to attend the University of Tennessee in Knoxville. He entered the graduate program in electrical engineering at UT in January of 1998 and worked as a teaching assistant for two semesters before beginning work as a research assistant for Dr. Don Bouldin. He has completed all of the requirements for the Ph.D in electrical engineering and that degree will be awarded by UT in December of 2000.

Nabil Kerkiz and his family will move to San Jose in California, in January of 2001, where he will work as a Hardware Design Engineer for Intel Inc.