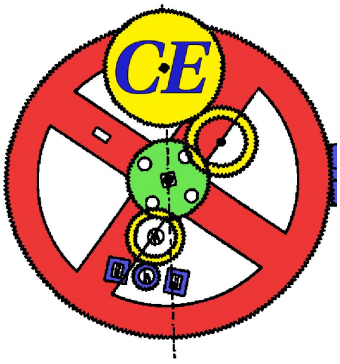


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2008

MSc THESIS

Design Partitioning for Custom Hardware Emulation

Nikolaos Mitas

CE-MS-2008-07

Abstract

Hardware verification is a very important step of system design. Various techniques are used for this purpose one of which is hardware emulation. Hardware emulation is a very efficient and flexible technique with high speed performance in comparison to other approaches.
Emulation using programmmable hardware can provide a very fast and feature rich debugging environment for system verification. The size and the complexity of todays Integrated Circuit designs though may exceed the size of the programmable devices used by the emulator in order to map the design under test. Therefore, in order to create a prototype of emulator and the design under test, we need to find a way to partition the whole design on the several programmable devices of the emulator.
This thesis addresses the problem of design partitioning for a custom emulator using the flatten netlist of the design and implementing a variation of the graph partitioning algorithm of Fiduccia–Mattheyses. The tool that we have developed extends the Fiduccia–Mattheyses algorithm while retaining the linear runtimes that the algorithm has in order to fit the various constraints of a custom emulator.
We extensively test the various parameters of the algorithm and the impact they have on the performance of the tool and report the behavior and the improvement on the number of cutted nets from an arbitrary and a manually clustered partition. In both cases the improvement is more than 50% upon the initial cut.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Design Partitioning for Custom Hardware Emulation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Nikolaos Mitas
born in Kavala, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design Partitioning for Custom Hardware Emulation

by Nikolaos Mitas

Abstract

Hardware verification is a very important step of system design. Various techniques are used for this purpose one of which is hardware emulation. Hardware emulation is a very efficient and flexible technique with high speed performance in comparison to other approaches.

Emulation using programmmable hardware can provide a very fast and feature rich debugging environment for system verification. The size and the complexity of todays Integrated Circuit designs though may exceed the size of the programmable devices used by the emulator in order to map the design under test. Therefore, in order to create a prototype of emulator and the design under test, we need to find a way to partition the whole design on the several programmable devices of the emulator.

This thesis addresses the problem of design partitioning for a custom emulator using the flatten netlist of the design and implementing a variation of the graph partitioning algorithm of Fiduccia–Mattheyses. The tool that we have developed extends the Fiduccia–Mattheyses algorithm while retaining the linear runtimes that the algorithm has in order to fit the various constraints of a custom emulator.

We extensively test the various parameters of the algorithm and the impact they have on the performance of the tool and report the behavior and the improvement on the number of cutted nets from an arbitrary and a manually clustered partition. In both cases the improvement is more than 50% upon the initial cut.

Laboratory : Computer Engineering
Codenumber : CE-MS-2008-07

Committee Members :

Advisor:	Mladen Berekovic, IDA, TU Braunschweig
Advisor:	Danne Klaus, CTG, Intel Corporation
Member:	Stephan Wong, CE, TU Delft
Member:	Sorin Cotofana, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

To the ones who matter the most.

Contents

List of Figures

List of Tables

Acknowledgements

Working on this thesis project has been more than interesting for me. I has been a great pleasure to work alongside my mentors at Intel Braunschweig, Gregor Stellpflug and Danne Klaus. Their constant and invaluable help and guidance made this thesis project possible. I would also like to thank my manager, Franz Olbrich, whose suggestions during the development phase have always pleasantly surprised me.

I would also like to thank my supervisor from TU Braunschweig, assistant professor Mladen Berekovic, for the feedback that he has provided me during the writting of the thesis document and assistant professor Wong for his contribution on the organization of this project.

Last but not least I want to thank all the staff at the Intel IBW site for their help in all the areas that they have assisted me and made my stay in Germany and in Intel in specific much more easy during the last 9 months.

Nikolaos Mitas
Delft, The Netherlands
August 20, 2008

1 Introduction

Design Emulation using programmable hardware has become an increasingly important step of the development of large Integrated Circuits (IC). The size of these large IC designs though, usually exceeds the size of the programmable devices due to the high complexity of the IC. Therefore, in order to create a prototype of emulator and the design under test, we need to find a way to partition the whole design on several programmable devices.

1.1 Development of Many Core Architectures

One of the main tasks that integrated circuit designers have to deal with, is that of system validation and extraction of performance models of the designs. This task is a critical step from the designing stage to the manufacturing stage.

In order to validate the correctness and the behavior of a design various approaches exist depending, not only from the size of the design but also, from the stage of development phase. For example, in early stages of development or for small designs, simulators are commonly used. These programs can replicate the behavior of a given design and the designer can verify if the system is working as expected.

On later stages of development, or for systems that are of considerable size and complexity, such as today's multi–core architectures, simulators become slower and less efficient. An alternative approach to do system validation in such designs is to use hardware emulation. Alongside the design under test, special hardware (*the emulator*) is attached and the whole design is then mapped on reprogrammable devices. The emulator provides a rich debug environment for the system and higher speeds.

The emulator and the design under test need to be deployed in reconfigurable devices, so as to provide the necessary flexibility that is required. We can already see at this point, that the problem that arises is the capacity of these reconfigurable devices.

1.2 Motivation for Emulation

Emulation is a powerfull tool especially for the later stages of the development phase. It can provide the necessary environment for the software development before the chip construction. It also provides verification capabilities that allow us to build a testing environment before we reach the stage of chip prototyping. Emulators have the necessary flexibility needed for validation purposes, which chip prototypes lack, by using reprogrammable devices as underlying hardware.

They manage to do this with considerably higher speed performance in comparison to simulators, that allows us to run long tests for debugging thus making them a good

verification platform. For this thesis, we will be dealing with a custom emulator design the details of which will be described later on.

1.3 Motivation for Netlist Partitioning

The essence of netlist partitioning problems is to divide a system design into two or more clusters in such way that the intercluster connections are minimized and cluster size fit the size criteria that is defined by the capacity of the underlying hardware. The motivation for partitioning is the drastic increase of the size and complexity of current designs that no longer fit on one device and the limited amount of I/Os that programmable devices have.

Even though FPGA manufacturers have done big steps in the direction of gate array technology in the recent years, the gap of FPGA capacity is substantial for modern non-trivial designs to be mapped on a single FPGA. Instead, multiple FPGA boards are used for a single design prototype. While this increases scalability, and in this sense it can decrease the cost as the underlying hardware can be of smaller capacity, it also introduces the problem of design partitioning. The decision of where to map various components of the design at hand is a non–trivial problem. The quality of partitioning is a critical factor on the cost, performance and complexity of the prototype. Poor partitioning will result in very poor performance.

The goal of this thesis is to partition the design under test onto the multiple FPGAs of the emulator.

1.4 Thesis Outline

This thesis document is organized as follows. In Chapter 2 we provide the essential information on the specific technology and the infrastructure that we will be working on. At that point a description of emulator technology and competitive validation techniques will be given. We then describe the mathematical background of graphs and hypergraphs that will be used to formulate the partitioning algorithms. Afterwards, we formulate the problem of netlist partitioning, present the existing algorithms and evaluate the suitability of these solutions. In the end of this chapter we justify our algorithm selection by presenting, from an abstract level, the advantages and disadvantages that these solutions have and how these are of importance to our needs.

In Chapter 3 we will describe the constraints that need to be fullfilled for obtaining a good quality partitioning solution. We then detail the algorithm and formulate the constraints that we have explained on the core algorithm. We will elaborate on the details and propose solutions on the implications that intercluster connections create. We explain the problems of multi-FPGA partitioning, such as heterogeneous (different FPGAs) and asymmetric (none all to all connections) FPGA topology, intercluster combinatorial paths, component/I/O prelocking etc. We then present the solutions that the selected algorithm provides on these problems. There we are going to explain the expected behavior of the algorithm in various situations and extend it so that it will incorporate solutions for the problems that arise and are not directed by the core algorithm.

In Chapter 4 we will present the implementation details of the partitioning tool that we have developed. We are also going to explain the decisions we made during the development phase. Further on, we shall explain the changes that need to be done on the existing tool–flow in order to deploy our partitioning tool.

In Chapter 5 we will present the experiments that we have performed on our emulator design. We will examine the parameters we have implemented and how these influence the partitioning solution. Based on the experiments and the results we will identify what parameters are of high importance and therefore should be fine–tuned more.

Finally in Chapter 6 we will summarize what we have seen in this thesis project. We will give the state of the algorithm extension as well as the implementation of the tool that we have developed. In the end of the thesis we propose where future research should be directed for custom design partitioning as we have observed throughout the project.

2 Background

2.1 Verification Techniques

In this section we provide the background on the existing verification/validation techniques and performance modeling methodologies. There is a number of different approaches for performing these techniques, depending on the level of abstraction and the development stage that we want to target. The most significant of those techniques are instruction set simulation and cycle accurate simulation, general purpose or custom emulation and silicon prototyping.

2.1.1 Instruction Set Simulation

In the early stages of processor design, computer architects are faced with exploring a very large design space. Simulators are effective and widely used tools for evaluating different design points. Simulation in general, is mainly performed by pure software, although there are cases where software/hardware co-design (hardware accelerated simulators) are used.

An instruction set simulator (ISS) is a program that mimics the behavior of a design by reading instructions and using variables to keep track of the states of the registers of the targeted design.

The fact that an ISS is running at a more abstract level – functional level (instruction set level) rather then the gate level – provides enough detail to run executable programs intended for the design under test. Unlike lower level simulation, the overhead for ISSs is concentrated on instruction decoding, functional operation and instruction scheduling and consequently is not dependent to a specific implementation of the architecture of the targeted design.

This on the other hand, has the disadvantage that in general, this approach does not provide timing accurate information for the architecture. There have been some efforts to make accurate timing information available for ISSs [29, 16] but this is usually difficult to implement and the speed of cycle–accurate ISS is very slow because of complex pipeline mechanisms, that are found in modern designs. therefore employing a cycle–accurate ISS for design space exploration becomes extremely difficult [25]. Nevertheless this approach provides low cost verification and performance modeling, fast turn around and low setup times.

2.1.2 Cycle Accurate Simulation

Accurate modeling of processors is also a critical task in the development of both hardware and software. As a result cycle-accurate timing is a requirement for system vali-

dation. Cycle–Accurate simulators (CAS) is a variant of a simulator which involves full modeling of the micro–architecture.

In modern designs this means explicit modeling of all stages of the execution pipelines (main and sub-pipelines). Usually it also requires modeling any instruction grouping or reordering, branch prediction logic as well as the resolution of any dependencies that might occur between instructions. This provides potentially the most accurate model but at the same time has a negative effect on simulation speed.

It becomes obvious that simulation time for CAS is a function of the input benchmark size and the level of micro–architectural detail simulated. Decreasing either of these parameters leads to faster simulation time. There have been efforts on the use of statistically reduced datasets and time sampling of the execution trace to reduce simulator input size, and the use of analytical modeling as a faster alternative to detailed simulation. However, these techniques often suffer from high errors in corner cases because they require critical information that was not retained in the simulation [30]. As with the ISS though, this is an approach that also has fast turn around and low setup times with relatively low cost.

2.1.3 Emulation

Hardware emulation is a technique with which we imitate the behavior of a system design or parts of it with another piece of hardware. The main objective of emulation technology is to provide a fast, efficient and feature rich debugging environment for the system. It can be thought of as a competitive technique to simulation, although there are fundamental differences on the approach that each technique takes and the level of abstraction and the stage of the development phase that they target.

Emulation is done by facilitating special reprogrammable hardware for the emulation. Emulating a system is done by building the actual system, or parts of it, in order to use it as a replacement of the system. The emulator is the result of mapping the whole system, or certain components of it, on reconfigurable devices (e.g. FPGAs). The procedure is similar to prototyping. The use of the same tools and tool flows is utilized with small variations, but whereas prototyping provides very little debugging capabilities emulators provide a debug rich environment which include traces and probes of any net in the design under test. Traces are signals that are collected during runtime into a local memory of the emulator and are flushed into another medium when this memory has been filled. Probing an arbitrary net at any given moment and getting the state back with advanced readback functions extending this way the debugging capabilities compared to prototyping where a probed net has to be explicitly defined during synthesis.

We can deduce that the main difference between simulation and emulation, is that while simulation mimics the outward appearance of a system, emulation mimics the cause processes used by the real system. This is more evident by the fact that hardware emulation requires that the description of the parts under test are synthesizable whereas this is not the case with simulation. The simulator needs only a behavioral model of the system. The main advantage of emulation is speed.

2.1.4 Silicon Prototyping

Silicon prototyping is mainly used on the later stages of development where the design is fully developed. Prototypes are used for post–silicon validation. While the speed of a prototype is considerably higher than emulation it only provides limited debugging capabilities. As a result they are usually used for system analysis and software development on the finalized system. [31]

2.1.5 Advantages and Disadvantages of Emulators

1. An emulator's biggest advantage is the fact that it performs many times faster than a simulator. In terms of order of magnitude, the speed of an emulator compared to that of a simulator can be up to 10.000–100.000 times faster [44] although a more realistic speedup may be in the range of 1.000 times. This practically allows the designers to run kernels and applications of significant size with extensive input data in large designs and still get results and debug information in reasonable time. This difference in performance is of high value for validation purposes. Current simulators take several hours to days to simulate small kernels on a full design making system and full chip validation practically infeasible. Simulation is much more efficient when used for testing components or functional unit blocks rather than performing simulation of full systems. This is due to the fact that it takes much more time to set up a design for emulation than for simulation.

2. The emulator requires that the functional blocks can be synthesized as they are going to be mapped on real hardware by the synthesis tool flow. While this can be a disadvantage, as a synthesizable design is not available in the early stages, it effectively means that the emulator is closer than CAS and ISS to the gate level. This makes emulation more restrictive as it requires synthesizable components. Simulation is more versatile as it will work also with behavioral models.

3. In simulation, speed is largely affected by the size of the design. The larger the design the more data have to be calculated by the software, which are inherently performed by serial calculations. In an emulator the size of the design also affects the speed but whereas in simulators the decrease of performance proportional to the size, in an emulator the performance decreases when the size of the design reaches or exceeds the threshold of the size of the target FPGA and needs to be distributed to multiple FPGAs. Otherwise the emulator does not suffer from performance degradation as the underlying hardware works concurrently. In the cases where the design has to be mapped on more than one FPGAs, good partitioning limits the problem and this is the subject that this thesis addresses.

4. An emulator can also provide cycle accurate information by doing clock scaling of the actual design. This way it is possible to find race conditions and other timing related issues that may arise in complex designs such as multicore and manycore designs, where two or more cores try to access the memory at the same time. Cycle accuracy is also possible in CAS, but is more difficult for ISS.

5. An emulator is also quite helpful if we want to get performance models out of a design. The fact that we are able to run applications with a large amount of input data in reasonable time, makes it possible to benchmark a design and do performance profiling. Because emulators provide large debugging capabilities we are also able to locate performance bottlenecks of the hardware and the software that is running. The speed that the emulator runs allows us to do profiling on a variety of different applications and input data. On contrary, simulating a large design with a lot of stimuli takes prohibitively long times to do any sort of measurement for profiling purposes. On the other hand, whereas prototype chips run at even higher speeds than the emulator they provide little information on the performance of parts of the design and therefore can only benchmark the overall performance of the system.

6. It becomes clear that emulators are good platforms for the late stages of hardware development and early stages of software development such as driver development or validation test programs.

7. Another advantage of emulators is the ability to get traces from an design. This can be done by specifying which registers will be monitored, which is usually done during synthesis by annotating various registers that the designer wants to monitor. The state of the registers will be recorded to the emulator's memory.

8. The disadvantage is that they suffer from performance degradation when we have a large number of probes inserted in the design. The trace memory of the emulator will fill up faster and the need to flush tracing data from the on–board memory to external media will occur more often, but the emulation itself will run at constant speed which is bound only by the clock speed of the emulator. Increasing the trace memory of the emulator can reduce this effect. Simulators also tend to slow down when simulation is performed with lots of probes.

9. A disadvantage of the emulator compared to simulation is that, whereas we can backtrace infinitely on a simulator, this is not feasible for the emulator as it would require to keep track of all net signals at all given moments. Instead, emulators can rerun from the start, the whole test case until the desired point in reasonable time thus diminishing this limitation.

10. Another disadvantage of an emulator is that it can only stop on cycle boundaries in contrast to a simulator which can have more accuracy, and finer grain stepping.

11. Emulators require more effort to setup and perform a test run than simulators as they require some changes to the RTL description. Also in order to make a change in the HDL description and perform another test run requires that we go through the whole toolflow which sometimes can take many hours which make turn–around times considerably higher than simulators.

12. Also the cost of an emulator is several times higher than the cost of simulators. Nevertheless validation technology has become so important that the advantages of emulators outweigh the disadvantages on the cases that performance is critical.

An overview, in terms of speed, setup/turn-around times and cost, of the aforementioned validation methods can be seen in Figure 2.1. While the cost, the required setup time and the turn around times are bigger in emulation the performance difference still provides an advantage compared to simulation. On the other end, test chip production is done in very late stages of development, costs several times more and is not flexible, so turn around times are considerably higher.

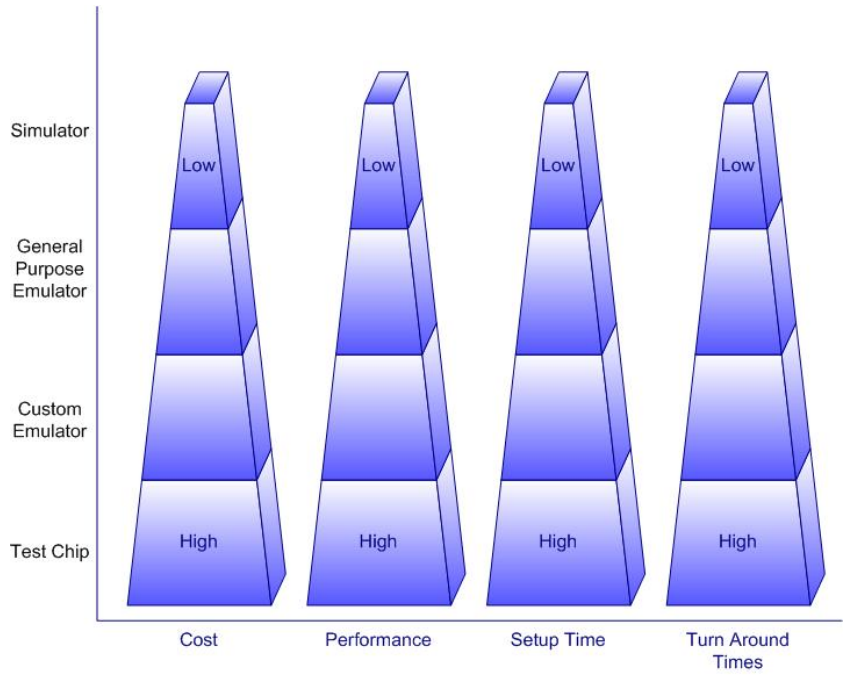


Figure 2.1: Comparison of different validation techniques

We have seen how emulation fits into the various verification techniques. In the next sections we will present in details some of the features of FPGAs which are the basis of most emulators.

2.2 Field Programmable Gate Arrays

Having explained the motivation for using emulators, we will now provide some details on the underlying FPGA technology that is used on our emulator. We will not provide detailed information on FPGA's as this is out of the scope of this thesis, but we will explain which features of the FPGAs are of interest to us for the task design partitioning. therefore at this point we describe some of the primitive components that current FPGAs have.

2.2.1 Architecture

The main building block of FPGAs is the Configurable Logic Block (CLB). Every CLB consists of a LUT of 4 or 6 inputs, some selection circuit like multiplexers (MUX), and

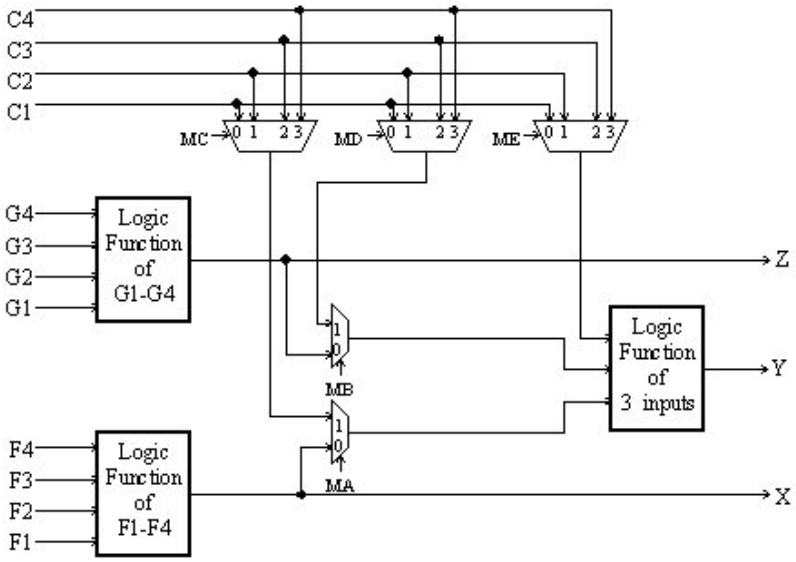


Figure 2.2: Configurable Logic Block [32]

flip-flops (FF). The LUT is reconfigurable so that it can handle any type of combinatorial logic (AND, OR, XOR etc), shift registers, RAM or any 4–input logic function. A common CLB is depicted in Figure 2.2. In our partitioning approach, CLBs which are the most common block on an FPGA, will be used as the main resource for partitioning decisions.

Nowadays FPGAs have dedicated resources for commonly used constructs. This is done to reduce unnecessary use of CLBs to describe these elements, which would potentially waste more CLB resources than necessary. A frequent component in system design are flip-flops. FPGAs provide primitives for these elements. Flip-flops are sequential logic elements. We will see later on that FF are of high importance in order to find good timing solutions for partitioning.

Block RAMs are another type of dedicated resource of modern FPGAs. They are on–chip memories of the FPGA which can be used for RAMs in the test design and are frequently used in designs. FPGAs also incorporate DSP elements to implement more efficiently DSP functions.

2.2.2 Interconnections

For the propagation of the signals between the functional blocks (CLBs, BRAMs etc...) flexible routing is used. In order to transfer the signals from one block to another, internal interconnections have to provide fast on chip signal propagation.

To facilitate external I/O, modern FPGAs provide multigigabit interconnection solutions. The FPGAs that we will be targeting (Xilinx Virtex 4 and 5) have high-speed, serial, chip-to-chip connectivity. Nevertheless signals traveling across FPGAs have longer propagation delays than signal propagating internally. The whole partitioning approach aims to minimize signals traversing FPGA boundaries.

2.2.3 Multiple FPGA board

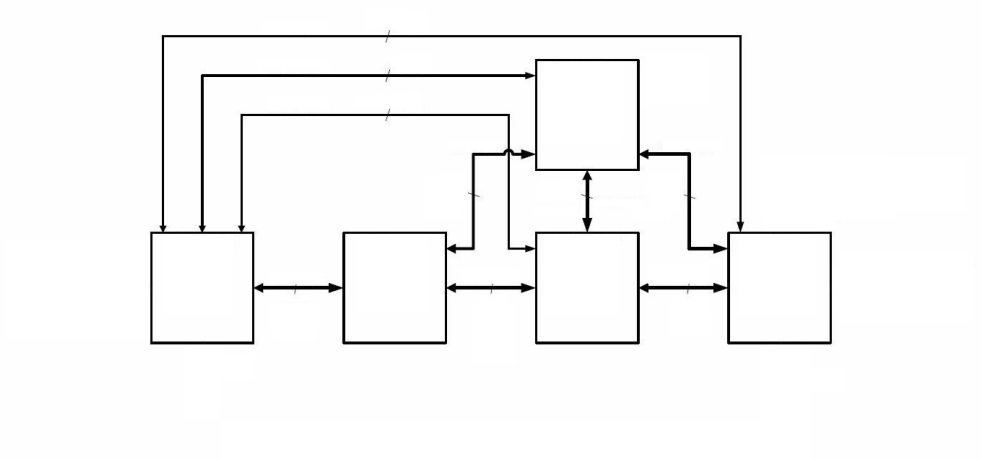


Figure 2.3: Board topology

The board that we will be targeting is composed by multiple FPGAs of different size and family with asymmetric topology. In essence, we different capacities of resources (total and dedicated) that do not have all-to-all connections. An overview of the board topology that we will be targeting can be seen in Figure 2.3.

2.2.4 Time Division Multiplexing

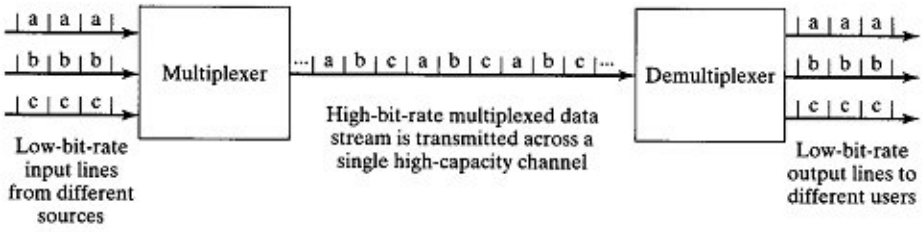


Figure 2.4: Time Division Multiplexing

Although modern FPGAs provide a substantial amount of external I/Os, the amount of signals that need to traverse across FPGAs will often exceed the number of I/Os for a complete partitioned design. As the existing I/Os of an FPGA can not cover all the signals, we incorporate Time Division Multiplexing (TDMX) [42], which multiplexes signals on the source FPGA and sends it through the RocketIO interface to the receiver FPGA where the receiver demultiplexes the signals. The obvious disadvantage is that given the constant transfer rate between FPGAs, the more signals have to traverse across

FPGAs the higher the clock cycles that will be needed to transfer them, which in turn requires lower overall clock frequency for the emulator in order to cope with the external I/O bandwidth. At this point, we introduce the TDMX factor which is the value that denotes the number of signals that have to be multiplexed. In Figure 2.4 we can see an example of time division multiplexing with a TDMX factor of 3.

2.3 Netlist vs HDL partitioning

Until now we have not discussed on the partitioning approach that we will follow. Although we will extensively explain the details of current partitioning approaches at the end of this chapter, it is useful to explain here the reasons that we have implicitly picked netlist over HDL partitioning.

Working on a netlist level is more advantageous than working on HDL level. The main advantages of working on an HDL level is that we are able to do RTL simulation without manual effort, whereas working on the netlist allows us to perform only perform a netlist simulation. Also, partitioning a circuit at the netlist level has much more objects than at the HDL level. This makes netlist partitioning more time consuming than HDL. This problem has been limited with linear netlist partitioning algorithms that limit the runtime. Apart from that though the advantages of working on a netlist level outnumber those of working on HDL.

Some of the advantages of netlist over HDL partitioning are the following:

1. We do not need any HDL modifications to perform partitioning. Partitioning can be performed directly on the netlist while in HDL we would require high manual effort.

2. Gated clocks are resolved easier as we can find the instances on the netlist.

3. We have a very accurate area estimation. Every object of the netlist is an entity that has been created by the synthesis tools and is going to be mapped in the hardware. This is not the case for an HDL object which can be larger functional unit (e.g. a shift register instead of the flip–flops that compose it). HDL area estimation becomes hard and difficult to automate.

4. Interconnection estimation for HDL becomes difficult as we must distinguish buses in the HDL description. In the netlist representation a net is a physical interconnection that exists between two objects. Buses are explicit instantiated and do not have to be treated separately.

5. We can uniquify components as we have explicitly instantiated all the cells that are used. On the HDL level multiple components can be instantiated almost implicitly. Multiple architectures of the same component also make difficult to calculate the real area of the components.

For all the above reasons we have decided to follow netlist partitioning.

2.4 Graphs

The first decision that we had to make was on the representation of netlists. We chose the graph representation that we later extended to incorporate the more complex data structure of hypergraphs. We will discuss these two structures and point out their suitability on the problem of netlist partitioning.

In this section we will also provide the mathematical background, terminology and notation that will be used later, on the approach that we followed to solve the partitioning problem.

2.4.1 Graph Definition

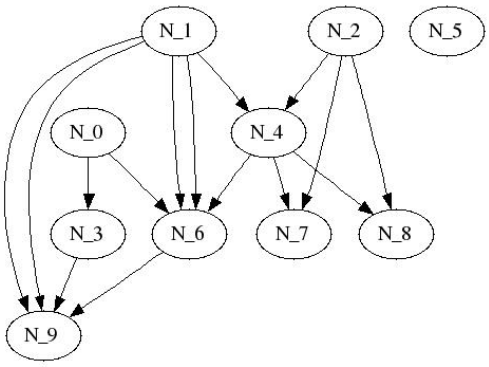


Figure 2.5: A labeled directed, non-strict graph with 10 nodes $V = N_0, \ldots, N_9$ and 15 edges $E = \{\{N_0, N_3\}, \{N_0, N_6\}, \{N_1, N_9\}, \{N_1, N_9\}, \{N_1, N_9\}, \ldots\}$

A graph is a set of nodes (or vertexes, or points) and a set of edges (or lines, or arcs) that connect *pairs* (cardinality of 2) of these nodes. More formally we can write that a graph is a pair $G = (V, E)$ of sets, where V is the node set and E the edge set, satisfying $E \subseteq [V]^2$. This translates that the elements of E are 2–element subsets of V [13]. An edge $e \in E$ is defined by the pair of nodes that is connects $\{n_1, n_2\}$ and is represented as $e = \{x, y\}$. A way to picture a graph is shown in Figure 2.5. A subgraph (or partition, or cluster) G' is a graph whose nodes and edges form subsets of the graph nodes and edges of a given graph G.

At this point it is useful to provide some conventions that will be used on this document:

- A graph with node set V is said to be a graph on V.
- The node set of a graph G is denoted by $V\{G\}$ and the edge set by $E\{G\}$. We shall not, though distinguish strictly between graph and its "node and edge set", thus meaning that we will refer to a node as $n_0 \in G$ rather than $n_0 \in V(G)$ in order to preserve intuitiveness and comprehensibility. This will apply also with edges ($e_0 \in G$ instead of $e_0 \in E\{G\}$)and so on, unless explicitly stated otherwise.

- We also denote node and edge *sets* with *uppercase* letters (e.g. N and E respectively), whereas *single* nodes and edges with *lowercase* letters (e.g n and e).

2.4.2 Properties

Following are some of the attributes of graphs that we will use[1].

- The *order of a graph* G is the number of nodes the graph contains and is denoted by $|G|$.
- We associate to every node a value $w(n) \in \mathbb{N}$ that specifies the weight of every node.
- A graph structure can be extended by assigning a weight to each edge of the graph. Weighted edges are used to represent structures in which pairwise connections have numerical values.
- The weight of a subgraph is the sum of the weights of all the nodes that belong to the subgraph $\sum_{n \in C} w(n)$

The number of edges of G is denoted by $||G||$. An edge e and a node n of that edge $n \in e$ are called *incident*. Two nodes n_1, n_2 are *adjacent* if there is an edge e in graph G that connects the two nodes. Two edges e, f where $e \neq f$ are called *adjacent* if they share at least one common node[2].

We can distinguish the following categories of graphs depending on the properties.

2.4.2.1 Undirected and Directed Graphs

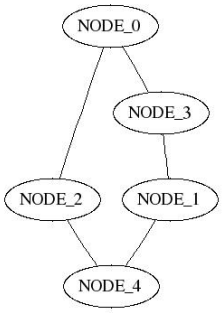


Figure 2.6: Undirected Graph

An *undirected* graph (commonly referred as *graph*) is a graph that does not provide information for the orientation of the edges. An undirected graph is depicted on Figure 2.6. The notation for the edges in the case of the undirected graph is not dependent on the ordering of nodes, so $e = \{x, y\}$ is equivalent to $e = \{y, x\}$.

[1]We will not present all the graph attributes, as they are not of interest for us nor is the intent to provide all the background on the field of graph theory.

[2]We will see later, in the case of non-strict graphs, where we can have more than one common nodes.

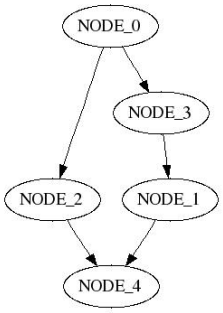


Figure 2.7: Directed Graph

A *directed* graph (also referred as digraph) is an orientation of an undirected graph. This type of graphs provides information on the direction of the edges by specifying a tail (or source) node and a head (or destination) node. The directed edge is represented also by $e = \{x, y\}$, but here the ordering of the nodes is important as it designates the source and destination (tail and head) of the edge. The head of the edge is node x and tail is node y. As we will be using directed graphs for the rest of the document, we adopt the notation of $e = \{x \rightarrow y\}$ or simply $\{x, y\}$, which implies the direction of the edge. We will also refer to nodes of directed graph as source and destination.

2.4.2.2 Strict and Not Strict Graphs

A *strict graph* G is a graph that does not allow self arcs and multi-edges in contrast with non–strict graphs. Multi-edges are defined if $e_1, e_2 \in G$ with $e_1 = \{x, y\}$ and $e_2 = \{x, y\}$, where $e_1 \neq e_2$. A directed multi-edge graph is shown in Figure 2.8.

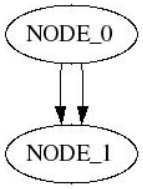


Figure 2.8: Multi–edge Graph

An edge e is a self-arc if $e \in G$ with $e = \{x, x\}$. A directed self-arc graph is shown in Figure 2.9.

Figure 2.9: Self–arc Graph

2.4.3 Hypergraphs

An important generalization of graphs are hypergraphs. A hypergraph is a pair (V, E) of disjoint sets V and E. V is the set of elements called nodes, and E is a set of non–empty sets of *any cardinality* of the nodes V. In essence graphs are a special form of hypergraph with cardinality of 2.

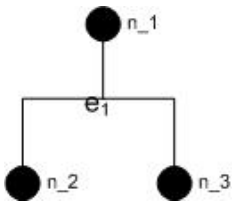


Figure 2.10: Hyperedge e_1

Hyperedges are denoted as $e = \{P\}$ with P being the set of nodes that the hyperedge connects.

Connections of hyperedges and nodes are better represented with incidence matrices. The incidence matrix of a n–node, m–edge hypergraph can be derived by 2.2.

$$A = (a_{ij}) \tag{2.1}$$

$$a_{ij} = \begin{cases} 1, n_i \in e_j \\ 0, otherwise \end{cases} \tag{2.2}$$

At this point it is useful to provide slightly more complex example of hypergraphs in order to demonstrate incidence matrices. Let G be the hypergraph of Figure 2.11(a) , E the set of edges and V the set of nodes. A hyperedge is a generalization of an "edge" that connects an arbitrary number of nodes as shown also in Figure 2.11(a). In Figure 2.11(a) edges e_1, e_2, e_3 are of cardinality of 3,2,4 respectively as they connect 3,2,4 nodes respectively.

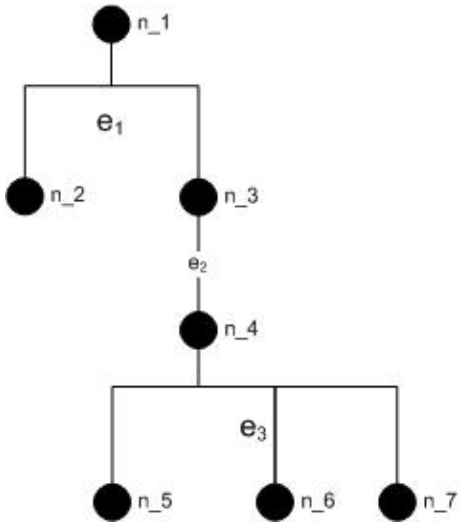
The incidence matrix of a node $n \in G$ denotes the connection of the node with all existing edges $\begin{pmatrix} e_1 & e_2 & e_3 & \dots \end{pmatrix}$. Accordingly is the incidence matrix for a hyperedge in graph G, which displays the connections with all the nodes $\begin{pmatrix} n_1 & n_2 & n_3 & \dots \end{pmatrix}$. In general we can represent for the whole graph G, with $n = |V|$ and $m = |E|$, the incidence matrix of size $n \times m$. An example is given in Figure 2.11.

2.4.3.1 Directed Hypergraphs

A directed hyperedge is an ordered pair, $e = (X \rightarrow Y)$, of disjoint subsets of vertexes; x is the tail of e while y is its head. In the following, the tail and the head sets of hyperedge e will be denoted by T(e) and H(e), respectively[17].

A directed hypergraph is a hypergraph with directed hyperedges. In the following, directed hypergraphs will simply be called hypergraphs. An example of directed hypergraph is illustrated in Figure 2.12.

The directed hypergraph's incidence matrix is defined by 2.3.

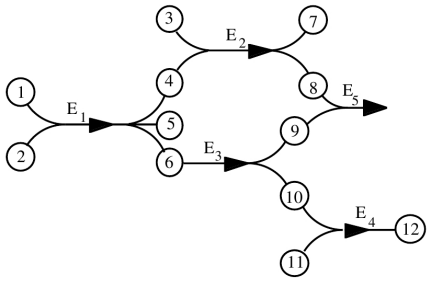


(a) Example Hypergraph

	m_1	m_2	m_3
n_1	1	0	0
n_2	1	0	0
n_3	1	1	0
n_4	0	1	1
n_5	0	0	1
n_6	0	0	1
n_7	0	0	1

(b) Incidence matrix

Figure 2.11: Hypergraph



(a) Example Directed Hypergraph

	e_1	e_2	e_3	e_4	e_5
1	-1	0	0	0	0
2	-1	0	0	0	0
3	0	-1	0	0	0
4	1	-1	0	0	0
5	1	0	0	0	0
6	1	0	-1	0	0
7	0	1	0	0	0
8	0	1	0	0	-1
9	0	0	1	0	-1
10	0	0	1	-1	0
11	0	0	0	-1	0
12	0	0	0	1	0

(b) Incidence matrix

Figure 2.12: Directed Hypergraph

$$a_{ij} = \begin{cases} -1, n_i \in T(e_j) \\ 1, n_i \in H(e_j) \\ 0, otherwise \end{cases} \tag{2.3}$$

The representation of netlists with simple graphs poses a problem, as a single net of a netlist can connect more than two nodes. Hypergraphs provide a data structure that can handle the information provided by the netlist.

2.5 Graph Partitioning

Before we introduce the existing algorithms later on this section, we present the formulation for the bi-partitioning problem which is the basis of the partitioning algorithms.

2.5.1 Problem Definition

We define P^k as the set of sets of clusters of each partition $P^k = \{C_1, C_2, \dots C_k\}$. We also define $F(P^k)$ to be the *cost function* that the algorithms will try to minimize. For the simplest case where we have a bi-partitioning problem $k = 2$ with the cost function to be *the number of hyperedges incident to nodes in both* C_1 *and* C_2 (commonly referred as *cut*). A min-cut bi-partitioning algorithm seeks to divide V into two clusters $P^2 = \{C_1, C_2\}$ where $C_1 \in V$,$C_2 \in V$ with $C_1 \subset V$, $C_2 \subset V$ and $C_1 \bigcup C_2 = V$ while minimizing the cut.

- *Min–Cut Bi-partitioning*
  Minimize $F(P^2) = |E(C_1) \bigcup E(C_2)|$ such that $C_1, C_2 \neq \emptyset$. Clusters may be unbalanced, so $w(C_1)$ can be different than $w(C_2)$.

- *Min-Cut Bisection*
  Minimize $F(P^2) = |E(C_1) \bigcup E(C_2)|$ such that $C_1, C_2 \neq \emptyset$. Clusters must be of such weight that $w(C_1) - w(C_2) \leqslant e$ where e is the weight of a node. If all nodes are of equal weight then e takes that value. If we have different weights of nodes then e has to be defined. Usually we select e to be the maximum weight of the "heaviest" node so as to allow more freedom.

- *Size Constraint Min-Cut Bi-partitioning*
  Let *upper size* U_1 and *lower size* L_1 be the maximum and minimum weight for cluster C_1 and U_1, L_2 the maximum and minimum for C_2. Minimize $F(P^2) = |E(C_1) \bigcup E(C_2)|$ such that $C_1, C_2 \neq \emptyset$. In this case cluster weights have to conform to $L_k \leqslant w(C_k) \leqslant U_k$ where $k \in [1, 2]$.

- *Minimum Ratio Cut Bi-partitioning*
  Minimize $F(P^2) = \frac{|E(C_1) \bigcup E(C_2)|}{w(C_1) \times w(C_2)}$. The numerator favours low cutsize whereas the denominator favors balanced clustering.

Lets see how all the above translate, if we apply these formulations to a simple partitioning example. For the graph of Figure 2.13 if we apply:

1. the min-cut bi-partitioning we have $C_1 = \{v_1\}, C_2 = \{v_2, v_3, v_4, v_5, v_6\}$ and a cutsize of 18 but the resulting clusters are very unbalanced.

2. the min-cut bisection we have $C_1 = \{v_1, v_2, v_3\}, C_2 = \{v_4, v_5, v_6\}$ and a cutsize of 300.

3. the size constraint min–cut bi-partitioning, for unitary weights of all nodes and $U_1 = 4$, $U_2 = 4$, $L_1 = 2$, $L_2 = 2$ we get the clusters weights:

$$2 \leqslant w(C_1) \leqslant 4$$
$$2 \leqslant w(C_2) \leqslant 4$$

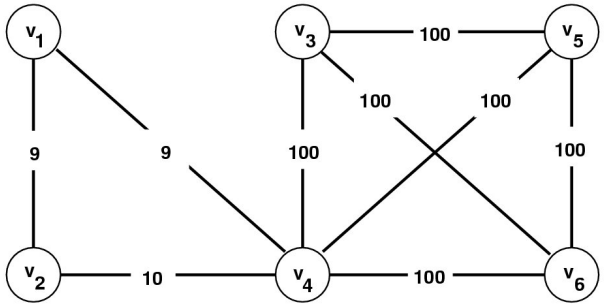


Figure 2.13: Example weighted graph

With these parameters we have a cut ratio of cutsize of 19 and clusters $C_1 = v_1, v_2$ and $C_2 = v_3, v_4, v_5, v_6$.

4. the minimum ratio cut bi-partitioning with a ratio of $\frac{|E(C_1) \bigcup E(C_2)|}{w(C_1) \times w(C_2)} = \frac{19}{8}$, with $w(C_1) = 2, w(C_2) = 4$. For these parameters of cutsizes and weights we get the same result as with the previous settings of size constraint min–cut bi-partitioning $C_1 = v_1, v_2$ and $C_2 = v_3, v_4, v_5, v_6$.

2.5.2 Introduction to Partitioning Algorithms

At this point, having introduced the basic notation and problem description, we give an overview on the most significant graph partitioning algorithms and their advantages and disadvantages. The algorithms we will examine can be categorized as random/exhaustive search, move based heuristics and meta–heuristics. At the end of this chapter, we compare the algorithms and discuss why the Fiduccia–Mattheyses heuristic is the most promising algorithm for our problem.

Graph partitioning problems are intractable, as the search space and in consequence the runtime increases much faster than polynomial rates, with the increase of the graph size. The problem of finding the optimal solution for partitioning problems has been shown to be NP-complete [33].

2.5.3 Exhaustive/Random Search

Because of the size of the search space, exhaustive search can be used only for very small graph partitioning problems and provide results in reasonable time. The simplest algorithm for this type of problems is to perform random search for possible solutions and compare the candidate solutions of this population. Unfortunately, this solution can only provide a good solution if the "good" solutions are of considerable size compared to the total solution space. When the search space is of significant size there is no practical benefit of using random search. It would require a prohibitively large population of solutions in order to get good quality results, leading us essentially to an exhaustive search of the search space.

2.5.4 Move Based Heuristics

Due to the complexity of graph partitioning, heuristic algorithms are employed. Heuristics are algorithms that *"attempt"* to provide solutions of arbitrary good or optimal quality, with provably good runtime for intractable problems such as graph partitioning. The quality of the solution is not guaranteed though. In practice, heuristics can be thought of as "rules of thumb" that apply well on certain problems.

Move–based algorithms try to construct iteratively better solutions from a given initial state by moving nodes across the boundaries of the partitions. They rely on the neighborhood of the current state on every iteration and the previous history of the optimization. As neighborhood we define the currently reachable solutions from a given point in the search space. The key factor of moved-based approaches is the amount of the perturbation that a move causes. Every move that the algorithm makes alters the neighborhood of the current state and after such a move a *new* neighborhood is reachable that previously was not.

The amount of perturbation allowed at a given state defines the size of the neighborhood of the that state. It is important that the allowed perturbation is fairly limited so that the transition from one state to another is more fluent. This allows the algorithm to make good decisions on a smaller scale, taking in consideration the locality of the candidate solutions and making the decision–making process more fine grained. This on the other hand has the disadvantage that a relatively small amount of perturbation can result in a very small neighborhood which lead to entrapment to local optimal solutions in the search space.

2.5.4.1 Kernighan–Lin Algorithm

The Kernighan-Lin algorithm (K–L) [24] is one of the earliest move based approaches proposed for graph partitioning that performs "relatively" good. K–L is a bisection (2.5.1) algorithm which requires an initial state (initial partitioning) in order to operate (Figure 2.14). The algorithm tries to optimize the cut size by making pair swaps of nodes between the two partitions. The neighborhood as state is defined from the allowed pair swap combinations. As a move–based approach, K–L applies the same methodology iteratively (*pass*) on the previous state in order to achieve a better solution.

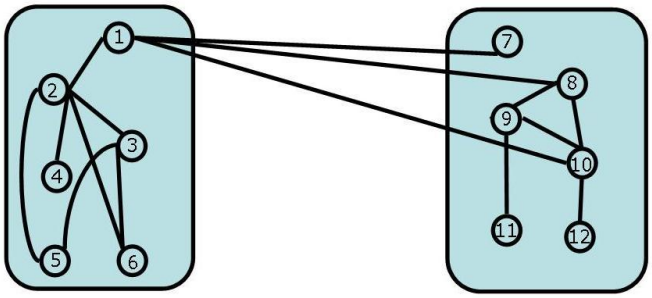


Figure 2.14: K–L Initial state (random - driven/previous pass)

The K–L algorithm performs the following steps:

1. At the beginning of a pass the algorithm calculates the total cut for the given initial partitioning (Figure 2.15).

2. It then calculates the *gain* on the cut for all permissible pair-swaps combinations (Figure 2.16).

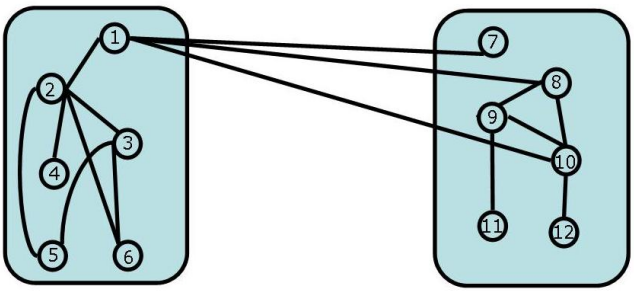


Figure 2.15: Calculate the current cut size. Cut size here is 3

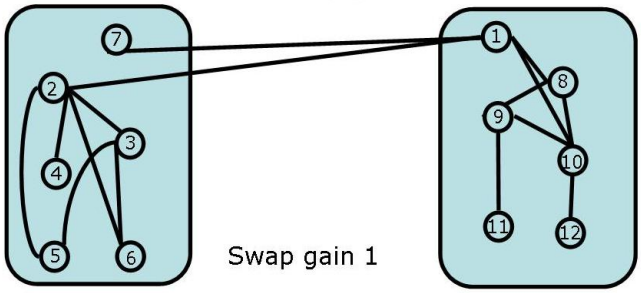


Figure 2.16: Calculate gain for swaps of node pairs {1,7}

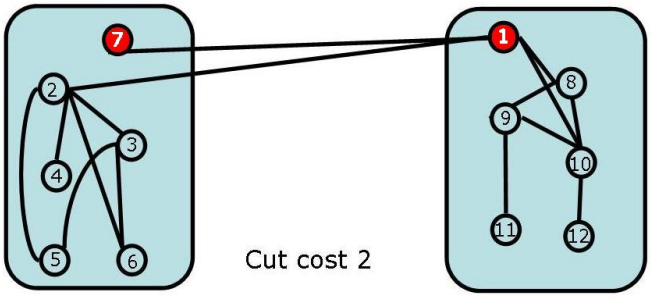


Figure 2.17: Lock the moved modules

3. It selects the swap–pair candidate with the highest gain and performs the swap of nodes.This can also be a negative gain move if at a given moment all moves are of negative gain. The total cut of the partitioning is updated and stored. A comparison with the previous state is performed and the state with the best cut is kept.

4. The pair of nodes that have been swapped are then locked in order to avoid moving them again at a later stage (Figure 2.17).

5. The algorithm continues from *step 2*, until all nodes are locked at which point a pass of *K–L* has ended.

6. The algorithm restores the state where the optimal cut stored in step 3 and unlocks all nodes.

The algorithm can be applied multiple times (passes) where the best cut found on one pass is fed again to the algorithm, until it reaches the point where there is no further improvement from the previous pass. This usually happens after a few passes of the algorithm.

The algorithm behaves like a greedy algorithm, as it always selects the highest gain move. Nevertheless the algorithm has some limited hill–climbing capabilities, as it will select a negative gain move if all other moves have higher negative gains. This might give a better solution in a later stage of the pass. In practice, the algorithm is known to perform good in the late stages, a few moves before the optimal point of the pass, where all the available moves are 0 or negative gain moves. Another advantage of the algorithm, is that it has a easy, straight forward formulation and simple data structures.

A disadvantage is the algorithm highly depends on the initial partitioning provided. A bad initial partition for the *K–L* algorithm will result to a local optimum which might be arbitrarily bad solution. Several ways to get around this limitation have been explored. The most promising of which incorporate some sort of clustering (compaction or contraction) or similar tactics to avoid getting trapped in a local optimum [7, 8, 38]. An alternative approach to the clustering variations, that also provide good solutions, is multiple starting point *K–L* algorithms [10]. This is an efficient and straight forward solution to avoid local optimums, but for the case of the *K–L* algorithm it has a negative effect on runtime.

This is mainly because the major disadvantage of *K–L* is that it requires $O(n^3)$ time per pass, where n is the number of nodes. This is caused by *step 2*, where the search for the optimal move is performed and requires $O(n^2)$ comparisons of swaps, though there are some improvement on the runtime of the algorithm that perform in $O(n^2 \log(n))$ or better [14]. Nevertheless, the speed limitation seems to have a big effect on the algorithm itself as well as the extensions that exist, especially in combination with approaches like multi–start, that was previously mentioned.

2.5.4.2 Fiduccia–Mattheyses Algorithm

The Fiduccia–Mattheyses (*F–M*) [15] algorithm is an improvement of the *K–L* algorithm. It is an move–based, iterative, min-cut, bisection heuristic whose worst case computation time, per pass, grows linearly with the size of the hypergraph $O(pn)$, where n is the number of nodes and p the highest connectivity of the node. Hence it can be applied to larger graphs than the *K–L* algorithm which has a complexity of $O(n^3)$.

To achieve this speedup in runtime *F–M* algorithm uses efficient data structures to avoid re-computation of the gain for all nodes like *K–L*. *F–M* tries to optimize the cut size by making moves of single nodes from one partition to another. This causes

the intermediate solutions to violate the bisection criteria because we have single nodes moving from one partition to another in contrast to the pair swaps in the K–L algorithm. A naive implementation of F–M allows deviation from the bisection criteria equal to the size of the largest node. There are extensions to the algorithm though, to make it a bi-partitioning algorithm and allow relaxed size constraints. While we will see the full details of the algorithm along with the extensions in Chapter 3.2 where we analyze our approach, it is advantageous to provide here a brief overview.

The key feature of the algorithm is the elimination of the gain recalculation of all nodes by providing a mechanism that keeps track, what nodes have been affected after every move. This is achieved by initially computing the gain for all nodes (as in K–L) and storing the data to a *bucket sorted array* as illustrated in Figure 2.18 [43, 35].

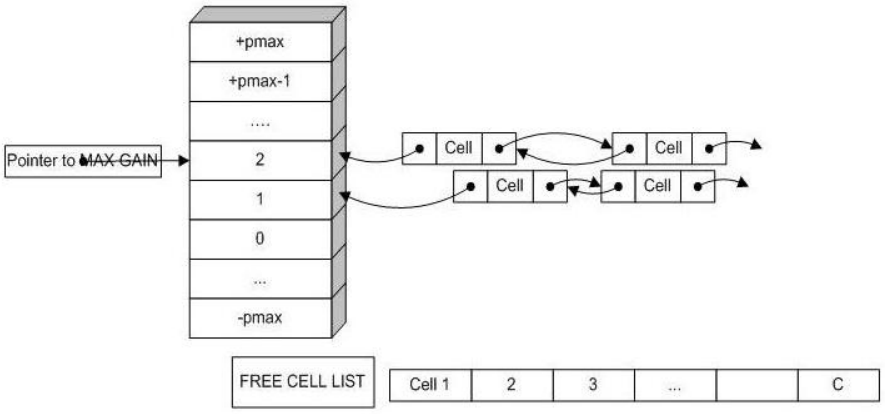


Figure 2.18: Bucket sorted array

The bucket sorted array is an array of containers that store nodes of equal gain. The array is sorted according to this gain. This array along with an efficient incidence matrix for all the nodes insure that, every time the algorithm decides to move a node from one partition to another, it will update (according to the incidence matrix) only the gains of the necessary (affected by the move) nodes in the bucket sorted array.

The algorithm first decides the source and destination partition according to the deviation criteria (tolerance of size). It will then select the highest gain from one partition according to the bucket sorted array (top of the bucket) and will then perform the move. The node is then extracted from the bucket sorted array and placed to the locked node list, so that it will not participate to any further moves during the same pass of the algorithm. The algorithm checks the adjacent hyperedges of the moved node in the hypergraph and updates only the nodes that are connected to that hyperedge.

With this mechanism the algorithm, for a given graph G with net set $E \in G$, has a runtime of $O(|E|)$ (number of nets of G). For the case of hypergraphs we will see that the runtime is $O(p)$, where p is the number of pins of the hypergraph.

The algorithm has similar behavior with the K–L algorithm. It has greedy–like

behavior with limited hill–climbing capabilities, as it will also select negative moves if all other moves have higher negative gains. Because it is an iterative algorithm, we can early exit at any point if the solution is of sufficient quality.

As with *K–L* the initial partitioning has to be provided and it is of critical importance to the quality of the results. Many extensions of the algorithm have been devised to avoid getting trapped in local optimal states. Clustering along with hierarchical (multilevel) partitioning, multi–start approaches and look–ahead schemes [27] have been used extensively and are considered as standard improvements of the core algorithm.

F–M dominates the literature and the industry as it provides good quality results with very good runtime. It is an intuitive and simple algorithm to describe and implement and incorporates more features on top of the underlying algorithm compared to *K–L* and the meta–heuristics that we will see below (such as multiway partitioning [39], different or multiple cost functions [41], multilevel approach, lookahead schemes, etc).

2.5.5 Meta–Heuristic Solutions

Meta–heuristics are general heuristic methods that apply to various optimization problems as long as the problems can be formulated to fit the algorithms. The two heuristic algorithms we just examined both demonstrate the behavior of greedy algorithms. Greedy algorithms are meta–heuristics that get trapped relatively easy in locally optimal solutions, as they can only make downhill movements. Other meta–heuristic solutions, like simulated annealing (*S–A*), genetic algorithm (*G–A*) and taboo search (*T–S*) demonstrate different behavior. As we saw with *K–L* and *F–M* algorithms, the main problem is that they highly depend on the initial conditions. Although many workarounds have been proposed, the problem is inherently part of the algorithm's design.

2.5.5.1 Simulated Annealing

Simulated annealing (*S–A*) [26] , is a probabilistic optimization method inspired by the controlled annealing in metallurgy. *S–A* requires an initial solution and a given neighborhood, which, for graph partitioning, translates as an initial state of the partition and a given set of neighboring states. Whereas in greedy algorithms only maximum gain downward moves are selected, *S–A* can select a worse solution in order to arrive at a later stage at a potentially better neighborhood. The probability that a bad solution will be selected is defined by $e^{\frac{-\Delta}{T}}$ (Boltzmann acceptance rule[40]) where Δ is the difference of the cut cost of the new state minus the current cost and T is the current temperature, which is decreasing while the algorithm progresses. As we can see, the *S–A* has a higher probability to select random bad moves in early stages of the algorithm. As the algorithm progresses, the temperature T, which is a monotonically decreasing function of the stage of algorithm, decreases (system cools down).

A study by Johnson et al[22] compared *S–A* mainly to *K–L* and the similar *K–L* derived improvement algorithms of coalesced *K–L* (C-*K–L*). The authors tested the approaches with several random graphs. They observed, that *S–A* is a competitive approach that performs good on some types of graphs in comparison to *K–L*. However it gets substantially outclassed on other type of graphs in terms and quality and runtime even when compared to the simple *K–L* algorithm, which has very long runtime. They

conclude that multi–start *K–L* might be a better overall approach, as the time that *S–A* will take to give good results would be sufficient for multi–start *K–L* or C-*K–L* to provide similar or better quality results. Although the fitness criteria of *S–A* on graph partitioning problems has not been fully identified they do make some interesting observations for *S–A*:

1. To get the best results relatively long annealing runs must be allowed.

2. Additional time in only high temperature (early stages) or low temperature (later stages) states does not seem to be as effective as adding time uniformly throughout the schedule.

3. Very high temperature states, where almost all moves are permitted by the Boltzmann acceptance rule, do not have a determined effect on the quality of the result so prolonged stay in such states could be avoided in favor of runtime.

4. Expanding the search space by loosening or even allowing some violation of the basic constraints of the problem definition, so long as penalty for the violation is included in the cost function might result to a smother solution space. therefore, escaping from local optima would be easier and for small penalties it might be still result to solutions legal or close to those.

It seems that VLSI netlists, which hold some natural hierarchical structure, do not take advantage of the *S–A* algorithm. *S–A* seems a more promising solution in the case where we have random graphs with no information of the hierarchy, and therefore the integrated mechanism of hill climbing in *S–A* is more influential on the final solution. Still, the literature seems to favor more multi–start iterative solutions.

2.5.5.2 Taboo Search

Taboo search (*T–S*)[18] is also a general combinatorial optimization technique. As the initially proposed algorithm by Glover [19] suggests, taboo search is a similar to *S–A*. In practice, for a simple implementation of *T–S*, the main difference from *S–A* is the fact that *T–S* keeps also a fixed size "taboo" list of it's recent moves (pairs of nodes that have been swapped recently). The taboo list prevents *T–S* to cycle around solutions in the search space that have already been visited in the recently thus providing a locking mechanism similar to *K–L* and *F–M*.

Although the size of the list is limited and therefore the possibility to revisit previously explored solutions is not eradicated, it does make the algorithm more efficient in searching the solution space than *S–A*.

More advanced implementations of the algorithm use attributes of states instead of single states, in the taboo list, thus excluding a more wide range of solutions previously explored. This though, can result to exclusion of potentially good solutions that have the same attributes with states that have been explored.

T–S has been applied in graph and hypergraph partitioning. A promising solution for *T–S* has been implemented by Arebi and Vannelli where they used a genetic algorithm to provide starting points for the *T–S* [5, 4, 6].

2.5.5.3 Genetic Algorithm

Genetic algorithms (*G–A*) [21] are a part of evolutionary algorithms family, used in several search problems. *G–A* algorithms require an initial population of solutions. As they are inspired from evolution theory, they operate with analogous behavior to evolution. *G–A* provide a crossover function which selects two solutions from the initial population, a mutation function that selects the traits to be passed to the offspring and a selection function that selects which solutions from the initial population will be replaced by the offspring generated from the mutation function. *G–A* have in general a variety of crossover, mutation and selection functions but implementations of these functions are more application specific. There are also implementations where two or more of the previously described steps of evolution are merged into one function.

Applying *G–A* to graph and hypergraph partitioning has been studied by Bui and Moon which provide the most prominent results[9]. They note that *G–A* algorithms are slower than other algorithms for doing the same tasks as they need a large number of generations to converge to a good set of solutions, if they ever find any. They also denote that *G–A* are ill-equipped to search local optima in a region of the solution space.

In [9] Bui and Moon present hybrid *G–A*s with local optimization techniques, mainly a variation of *F–M*. They suggest that *G–A* should be used to provide good initial populations where, later, local optimization techniques can be used.

2.5.6 Discussion on Graph Partitioning Algorithms

In this section we have seen an overview of the most common and widely used algorithms for graph and hypergraph partitioning. We have seen that *K–L* algorithm provides an efficient basic methodology for partitioning but requires very long runtime to complete. Also, as a greedy-like algorithm it demonstrates some tendency to get trapped in local optima. While there are ways to improve on the runtime, *K–L* will still lack the runtime provided by its more advanced variant, the *F–M* algorithm. Another issue with *K–L* is that it is a true bisection algorithm that has also been adapted to incorporate bi-partitioning (uneven partitions) but still lacks the flexibility of *F–M* that has, either inherently or by small adjustments in the core algorithm, the ability to provide results tailored for most of the cases.

Simulated annealing provides a mechanism that is supposed to override the problem of entrapment in local optimal solutions that the greedy alternatives have but achieves that in a high cost of runtime. Nevertheless, this would still be acceptable if the algorithm would provide consistently better results, but existing literature suggests that *S–A* does not perform well on generic cases of graphs.

Taboo search gives a solution to the long convergence times of *S–A* without affecting the quality of the results in general, although this highly depends on the implementation of the taboo list. A disadvantage is that quality is dependent on the initial solution and the run times are still high compared to *F–M*. While there are solutions for limiting the dependability of the algorithm to the starting partitioning like imposing smaller perturbation for the neighborhood and thus tightening the locality, this also limits the overall hill-climbing capabilities of *T–S*.

Genetic algorithms seem to deal with the problem of finding a good initial cut that

can be used by other methods quite effectively in terms of quality for random graphs. They provide generally good solutions for cases that there are no prior information on the nature of the graph (hierarchy, clusters) but they suffer from high run-times and local search limitations. Hybrid *G–A* seem to get past that problem by incorporating local heuristics to do the fine tuning of the starting solution provided by the *G–A*. This, depending on the usage of *G–A* in the hybrid solution, sacrifices some of the quality for better runtimes. Hybrid *G–A* have the disadvantage that they are quite inflexible and complex algorithms to deal with combination of limitations in the search space as the limitations have to be addressed by the genetic algorithm as well as the local heuristic.

The *F–M* algorithm seems the most promising candidate for hypergraph partitioning. Although it is a bisection algorithm, it is very easily extended to bi-partitioning which we are interested in. It has the best runtimes compared to its rivals, while it provides good results even if they might not be the globally optimal solutions. Runtimes for the algorithm are of critical importance for our case as we are dealing with very large graphs with $100,000--1,000,000$ nodes. Also the fact that the problem of local optima entrapment has been extensively addressed by literature provided a variety of solutions. A widely used tactic is by providing information on the nature of the hypergraph and several industry tools are known to implement this. All this along with the intuitiveness, flexibility and low implementation complexity of the algorithm make *F–M* the right choice for our problem.

Partitioning for Custom Emulation

3

In the beginning of this chapter we give an overview of custom emulation and the constraints that it has. We then provide the details of the ***F–M*** algorithm. We will then propose for each of the constraints, candidate solutions as extensions to the F–M analyzing in detail the problem and the solution that we propose.

3.1 Custom Emulation

Custom emulators are a type of emulators designed for specific needs and their architecture are often tailored according to their application[28]. Although our custom emulator is targeting a specific application area, many of the challenges that custom emulators introduce are common across areas. We present in brief the challenges for our platform.

Multi FPGA: The amount of FPGAs used in the emulator is one aspect of the architecture of an emulator. Multiple FPGAs improve scalability of the emulator but on the other hand one must take in consideration to partition the design under test among many FPGAs. The FPGAs can also be of different types, thus one should also be able to distinguish the dedicated resources of an arbitrary FPGAs. This means that partitioning decisions should be customized for the specific FPGA resources. The area is one parameter but usually more fine grain parameters like BRAMs, DSP slices etc should also be considered if we want a more fine grain partitioning solution.

Asymmetric topology: The topology of the emulator is also a hard constraint. FPGAs used in the emulator are not necessarily connected in a fully connected or symmetric topologies (star, mesh, hypercube, etc). Partitioning solutions should honor the topology imposed by the architecture if we want to avoid using FPGAs for routing between cells that are not directly connected.

Locked components: Some components of the emulator design have to be assigned in a specific FPGA as they might serve for external I/O or other functionality that might only be performed on a specific resource that is not contained on all FPGAs.

Time–Clean solution: Timing of the proposed partitioning solution has to allow us to fullfill the timing constraints of the application. An important parameter for timing clean solutions is the creation of combinatorial paths. Combinatorial paths occur from the creation of fast paths of combinatorial logic that cross FPGA boundaries more than once. We will see in detail later on this Chapter how this can affect the performance of the emulator's speed.

3.2 Fiduccia–Mattheyses Core Algorithm in Detail

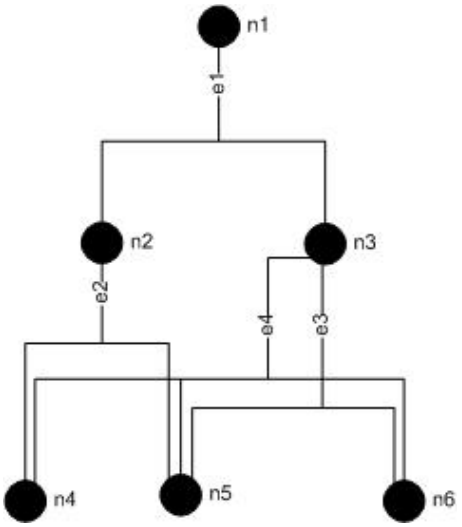


Figure 3.1: Graph example

The K-L algorithm suffers from constant recalculation of all cells every time a cell has been moved (Chapter 2.5.4.1). The F-M algorithm is based on K-L but gets passed this problem by incorporating special data structures. We describe here these data structures by providing an example netlist. As we have seen in Chapter 2, netlists are represented efficiently by hypergraphs.

3.2.1 Data Structures

Consider the example of Figure 3.1. From the input data that describe the netlist we construct the data structure of Figure 3.2. In this data structure we store the connections of every CELL to NET and vice versa. This data structure allows us to quickly find the adjacent CELLS of one selected CELL, by looking to which NETS the CELL is connected and then checking to which *other* CELLS these NETS are incident to. The union of the found CELLS are adjacent to the selected CELL.

For example, let us find the adjacent cells of cell $n3$. Looking at the NETS data structure of Figure 3.2, we see that $n3$ is connected to nets $e1, e3, e4$. We then look at the CELLS data structure for each of these nets (hyperedges). Excluding the cell $n3$, that we have selected from the list of cells, $e1$ is connected to $\{n1, n2\}$, $e3$ is connected to $\{n5, n6\}$ and $e4$ is connected to $\{n4, n5, n6\}$. The union of all these sets $\{n1, n2\} \bigcup \{n5, n6\} \bigcup \{n4, n5, n6\} = \{n1, n2, n4, n5, n6\}$ provides the cells adjacent to cells $n3$. Similarly the cells adjacent to cell n2 are $\{n1, n4, n5\}$.

As with K-L, F-M also needs to find the gain of moving a node from one partition to another. F-M stores this information so that it will not recalculate this value every time a move has been made. The data structure to hold the gain of moving one node from one partition to another is the bucket sorted array shown in Figure 3.3. We require one bucket sorted array for every partition.

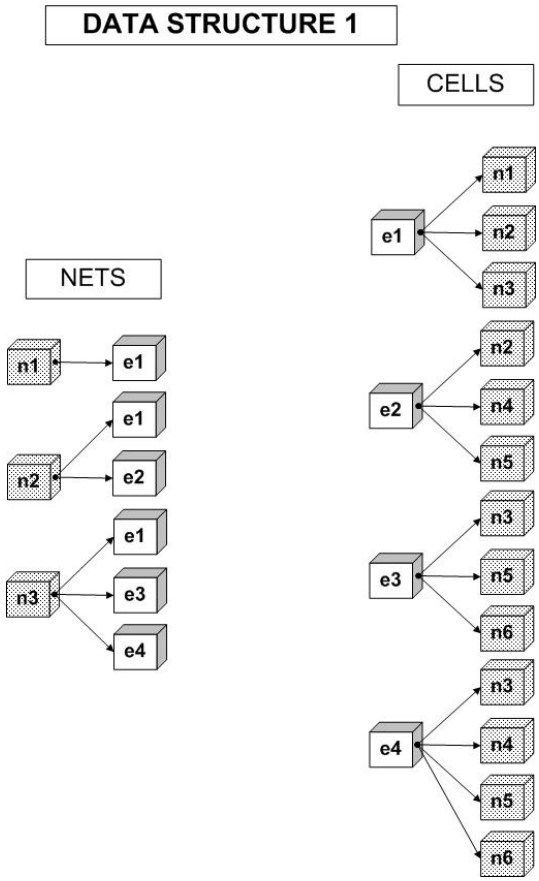


Figure 3.2: Data structure 1 (DS–1)

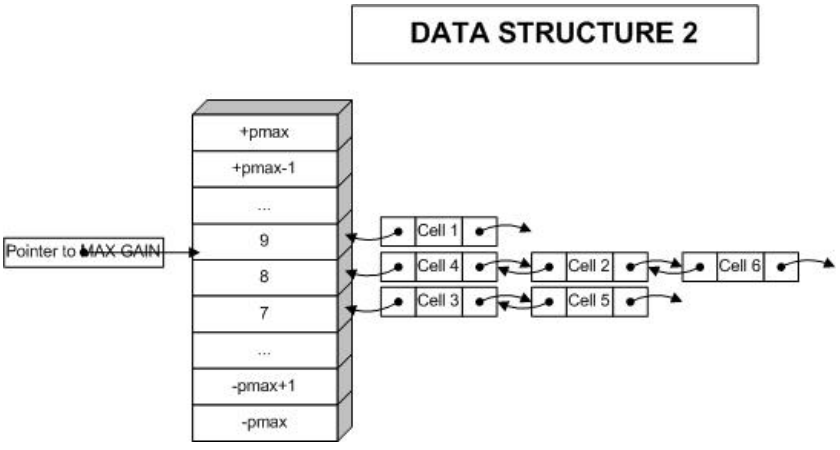


Figure 3.3: Data structure 2 (DS–2)

It becomes obvious at this point, that the value of gain of moving one cell from the source partition to the destination partition, is bounded by the number of pins of the cell. Maximum gain case occurs when all the adjacent nodes are on the *destination* partition, so by moving the cell there we gain p on the cut cost, where p is the number of pins of the cell. In contrast minimum gain case occurs when all the adjacent nodes are on the *source* partition, so by moving the cell to the destination partition we lose p on the cut cost. Thus, the gain value of all the cells is bounded by the maximum value of pins of the cells $pmax = max\{p(i)\}$, $pmin = -max\{p(i)\}$.

The bucket sorted array stores the gains of the free cells of the netlist by grouping several cells with the equal gain in the same bucket. All the cells in the same bucket are linked in a doubly linked list that is connected to the index bucket of array. We also maintain a *MAX–GAIN* pointer to the bucket of the highest gain.

To prevent the cell-moving process from "thrashing" nodes, or going to an infinite loop, each of the cells of the graph is locked in its new partition immediately after the move. Also the cells that we wish to exclude from the moving process (e.g. preassigned cells) are also locked and are not inserted in the bucket sorted array. For this we provide a LOCKED CELL LIST. Having in mind these data structures we proceed to see the steps of the algorithm.

3.2.2 Algorithm Explanation

We create DS–1 from the description of the netlist. The algorithm then performs the following.

1. *Given is the initial partition (Random–Directed/Previous Pass)* shown in Figure 3.4 and the desired ratio $r_{desired}$. The ratio at any given moment is $r = \frac{|A|}{|A|+|B|}$ where $|A|, |B|$ are the sizes of the partitions. r is the selection criterion.

2. Calculate the gain $g(n)$ of all cells. In Figures 3.5 and 3.6 we calculate the gains for moving nodes *n1* and *n3* respectively.

3. *Insert the nodes* according to their gain into DS–2 (one DS–2 per partition) and keep track of the highest gain bucket by updating the MAX-GAIN pointer. We have constructed the two DS–2 for the simple example of Figure 3.4, in Figures 3.7 and 3.8.

4. Select from which partition we should move the cell according to the ratio r. If $r < r_{desired}$ then the source partition is partition A and destination partition is partition B and vice versa for the case where $r > r_{desired}$.

5. Retrieve from the selected DS–2 of the source partition the head of the linked list which is the top bucket with the highest gain and is referenced by the pointer MAX–GAIN (Figure 3.9). Move the selected node to the other partition and move it from the GAIN BUCKET to the LOCKED CELL LIST (lock bucket). Update the size of the new partition by adding and subtracting the weight of the moved cell to the destination and source partition respectively.

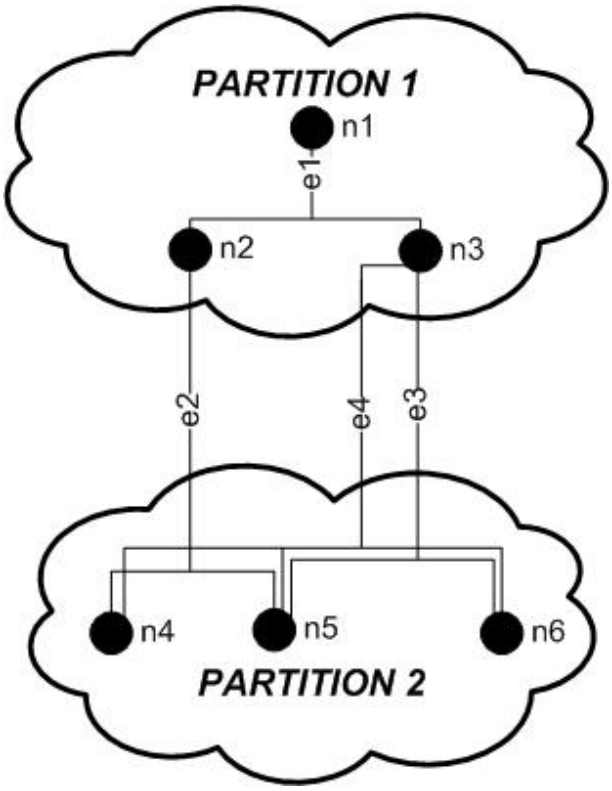


Figure 3.4: Initial Partitioning

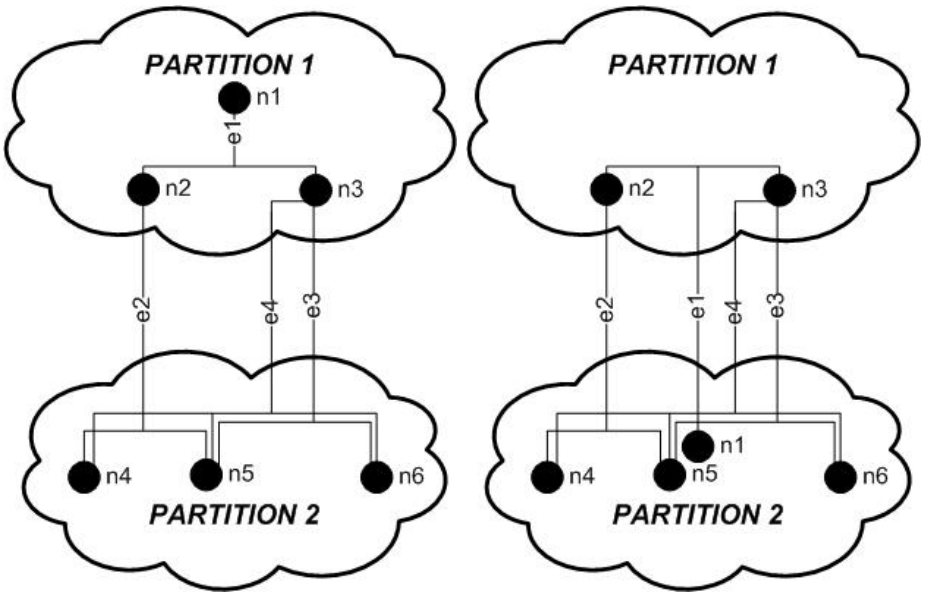


Figure 3.5: Gain of moving node *n1* to *PARTITION 2 is -1*

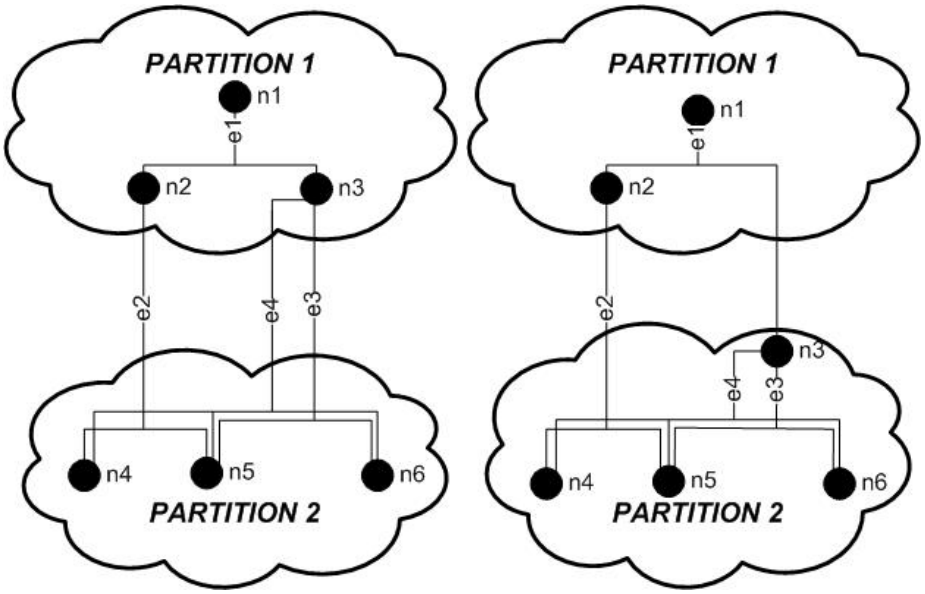


Figure 3.6: Gain of moving node *n3* to *PARTITION 2 is +1*

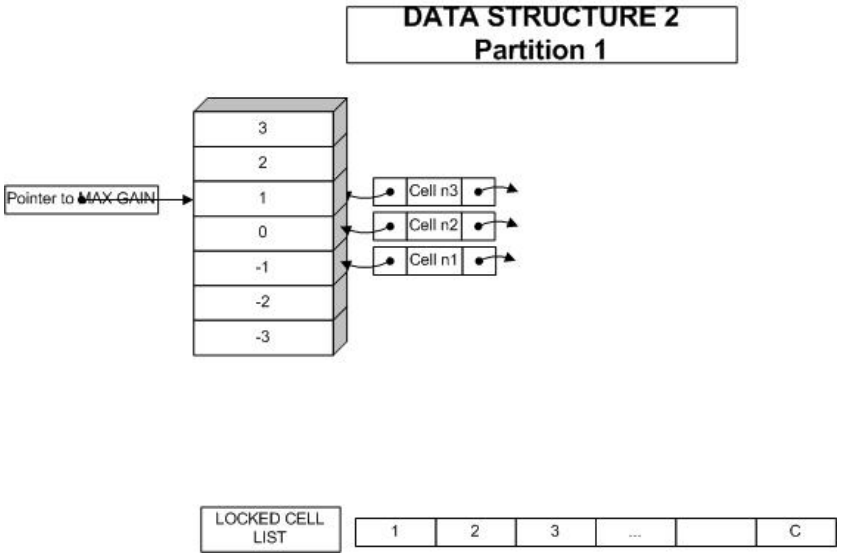


Figure 3.7: DS-2 of partition 1

6. *Update* the gains of all cells that are adjacent to the moved cell. Adjacent cells can be found by looking up in the DS–1. Adjacent cells might be in both partitions. The update of DS–2 is performed by popping the adjacent cell from the linked list of the gain bucket that is found. Then a recalculation of the gain is performed and the node is reinserted to the new gain bucket. We also make sure to update the MAX-GAIN pointer if the moved node or the recalculated nodes have affected the max gain value.

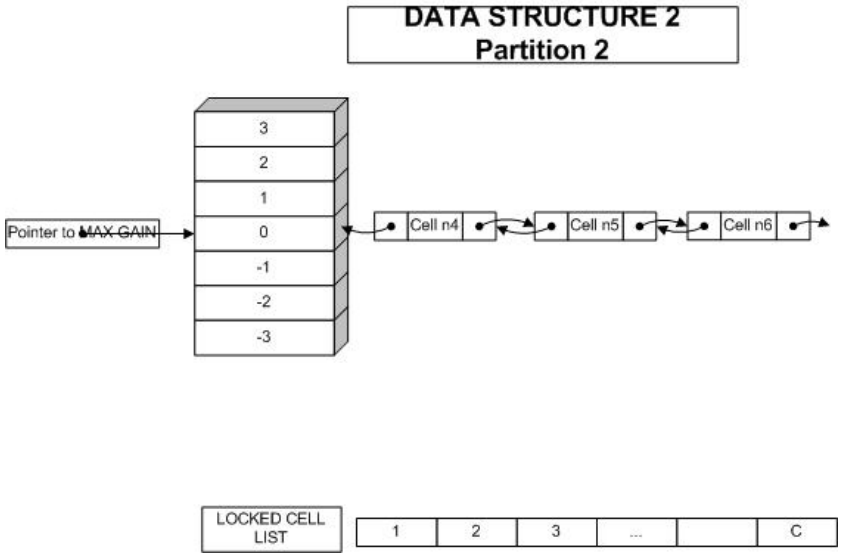


Figure 3.8: DS-2 of partition 2

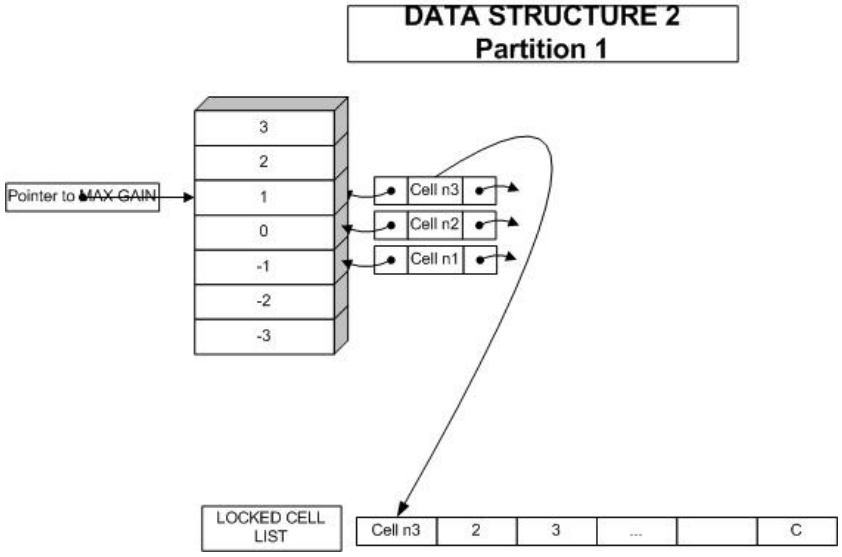


Figure 3.9: Selection of head node from top bucket of DS-2 of partition 1

7. Check if this move improves on the solution and store the state of the best cut of the pass.

8. Continue at step 4, until all cells are locked or the ratio prevents further moves. Otherwise free all cells. This ends the pass of the algorithm.

9. If there was no improvement from the previous pass then exit, otherwise restore best cut solution kept in step 8. Set initial partitioning = best solution and perform another pass.

3.2.3 Analysis of F–M Algorithm

At this point we make some observations on some implicit decisions, parameters and behavior of algorithm. The first observation for the algorithm is that it has a linear runtime. This stems from the fact that the algorithm only recomputes the gains of the neighborhood of the moved gain. The gain recalculation of the neighboring cells can be done by simple gain increment/decrement which can be done in constant time. Thus if the moved node is connected to nets of maximum connectivity of p the complexity would be $O(np)$ in the worst case, providing a linear increase of runtime.

We have already seen that the size of a partition is defined as the sum of the weights(sizes) of the constituent cells. In step 1 we defined the desired ratio $r_{desired}$ for bi-partitioning as the cell distribution between the two partitions. This for a known total size of the netlist defines which cells go in the two partitions. The ratio is a hard constraint for F–M. The algorithm tries to minimize interconnections between partitions but can not violate this constraint. This is seen in step 4 where the algorithm's selection of the source and destination partitions is performed according to $r_{desired}$ and the current ratio. This though implies that the ratio has a tolerance equal to the size of the largest cell of the netlist. In practice this means that we have a constant toggling of source and destination partitions. We extend the idea of ratio tolerance to incorporate user defined values. For a specified tolerance t we have $r_{desired} + t$ to be the range where the algorithm can select as source and destination any of the partitions. This allows the algorithm to select the globally best move by selecting the highest MAX-GAIN bucket of the two partitions. Of course if we also reach the boundaries of the tolerance we may also have the same behavior as before but nevertheless the application of relaxed size constraints provides more flexibility. There has been some work on relaxed size constraints on the F–M algorithm [12] as well as for hierarchical approaches [12] and for multiway partitioning [37].

The algorithm displays some limited hill-climbing capabilities. Although the algorithm suggests that it will only do positive gain moves there are cases where the algorithm makes zero gain or even negative moves to obtain a better solution. This happens when the algorithm has exhausted the positive gain buckets from the two partitions so it is obliged to select either from a zero gain or negative bucket. A arbitrary "zero gain move" may have an impact to other zero gain moves thus increasing the gain of other cells currently in zero gain buckets. Although there is not a mechanism to foresee this behavior we will see, in the experiments that we have conducted, that it happens relatively often especially in the early stages of the algorithm. Negative gain moves that improve the total cut cost happen in similar situations. If all moves are of negative gains then selecting, one might result to increasing *more than one* negative gain cells up to the "zero gain" bucket. It is possible that we then fall in the previously described state. There are more cases that this behavior can occur, as for example when we have reached the $ratio + tolerance$ boundaries and therefore the algorithm *has to* make a move from a partition with only negative moves instead of the positive ones from the other partition.

Another observation stems from the fact that the algorithm works iteratively and has greedy–like behavior. At the end of each pass the algorithm has found an optimal solution. The algorithm though does not specify how to store the best state of the

partitioning. One naive approach would be to store the partition of all cells when an improvement on the cut cost has occurred. This though means that the algorithm would have to constantly take snapshots of the state of the partitions especially in the early stage of the pass where the algorithm goes downhill. This for a considerable size of netlist would increase dramatically the runtime as well as the memory consumption of the extra snapshot. An alternative solution is to specify a threshold of allowed consecutive negative moves and when this is crossed re-roll back to the best solution. Since the algorithm is greedy, the expected behavior is that it will do the best moves in the beginning and will leave bad moves for later when it has reached the optimal solution of the pass. The size of the threshold of negative moves allows us to correctly identify where the optimal solution. We need though to make sure that the threshold is big enough so that it will not mistake a locally optimal solution as the globally optimal as we have seen that the algorithm can escape from local optimal solutions. On the other hand the size of the threshold has to be *relatively* small as we need to reserve a queue[36] of moves that we can roll back to. The size is usually defined empirically as a percentage of the total number of moves, which in turn is linear to the size of the netlist. An alternative solution would be to store the best cut size during the pass and rerun it until that point. Although this would guarantee the optimal solution of the pass, it would also mean that the algorithm would have to finish a whole pass before it can perform the rerun. It is also obvious that because it is greedy algorithm, the ratio of moves $\frac{Move\ Counter_{global\ optimal}}{Move\ Counter_{full\ pass}}$ is relatively small and thus the algorithm would waste time on running until the end of the pass. This effect is more obvious as the whole F–M algorithm gets the best cut from the final pass when the algorithm has been settled in an optimal(global or local) solution which happens after some passes. Nevertheless the fact that F–M has a linear runtime allows us to perform this solution without creating too many problems and bottlenecks on the runtime even if we run a full pass.

An obscure point on the implementation of the algorithm is the selection–insertion policy for the moving of cells in the DS–2. The original F–M[15] algorithm suggests that the head of the MAX-GAIN bucket should be chosen for selecting the highest gain cell. Also the insertion of recalculated nodes should be done in the head of the linked list of the bucket sorted array. The reasoning of this LIFO (last in first out) policy is not explained but one can speculate that it enforces "locality" in the choice of cells to move, but this is rather a speculation in the literature. One can easily expand this policy to FIFO without much loss on runtime. This would practically mean that the algorithm would either insert the recalculated cells in the end of the linked list or that it would select the last cells in the end of the linked list. This would increase the runtime a bit as we would have to traverse the linked list every time we needed to select a node from a bucket DS–2 and it would increase it a bit more if we insert the recalculated nodes in the end as this would have to be done possibly more than one times. A random policy would also increase the runtime as we would have to calculate the number of the selected cell in the linked list and it would also require to keep track of the length of the list so that we do not reiterate on the same bucket. The effect on the quality that this selection has, is not obvious and has not been tested thoroughly. In fact while some favor random selection [27, 39] others prefer LIFO [15]. Although slight modification on the organization of the buckets can dramatically effect the algorithm, we expect that

the impact on the quality and runtime, of the *selection between these three policies*, to be relatively small as it only affects the level of randomness of cell selection in the same bucket.

3.3 Custom Emulation Architecture - Algorithm Extensions

The constraints that we have in our architecture require some modifications on the algorithm in order to be fulfilled.

3.3.1 Multi FPGA board – Asymmetric Topology

Our target board consists of more than two FPGAs so we need to extend the F-M algorithm from 2-way partitioning to k-way partitioning.

An efficient way to extend F–M to k-way has been proposed by Sanchis[39]. Sanchis makes use of binding numbers and gain vectors specified by Krishnamurthy[27] which improve the F–M algorithm by providing a lookahead mechanism for the F–M.

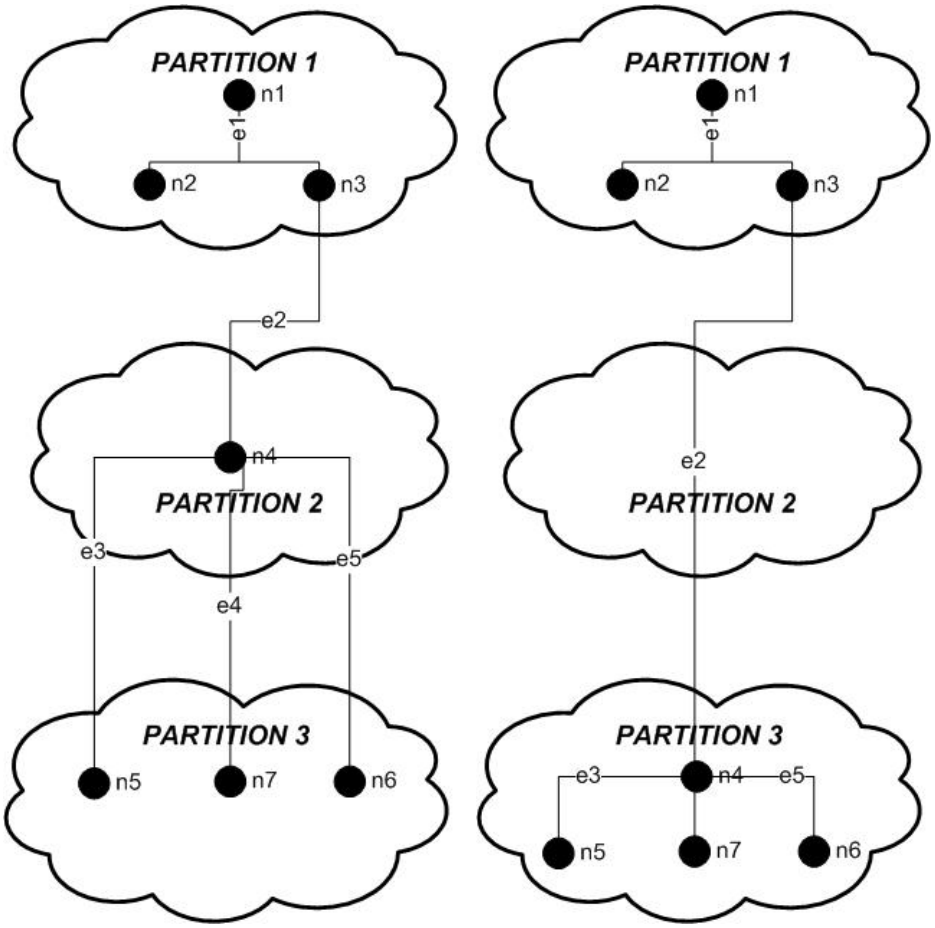
The F–M algorithm is known to make seemingly good moves that might turn out bad after few steps ahead, thus locking itself to a suboptimal area of the search space. This is mainly because it lacks a "tie-braking" mechanism that would allow the algorithm to have some insight on the next potential moves. It has been observed[20] that for the `Primary1 MCNC` benchmark (a 833 cell netlist) the top gain bucket has 15-30 cells and the algorithm makes some bad moves when it comes to choose the moving cell. Krishnamurthy tries to minimize this effect by replacing the gain values with gain vectors.

In short the first value (1st depth) of the gain vector of a cell n shows the number of cells of a net $e \in E_n$, that have been locked on the partitions (binding number). This way we can see how many cells have to be moved from one partition in order to transfer the whole net on one side. Provided that we have a k depth vector this mechanism allows the algorithm to see k moves ahead and thus is able to move whole subnets on one partition or make better moves on later stages. This also increases the complexity of the algorithm from $O(np)$ to $O(npk)$. Sanchis proposes this data structure instead of the normal gain values of F–M for k-way partitioning. She points out that the F–M tends to minimize connections between two blocks but maximizes connections inside these blocks and therefore cannot be efficiently applied iteratively.

We do not make use of this extension but instead we propose a similar strategy while keeping the original gain values of the F–M algorithm. In 2–way partitioning we saw that we have one gain value for moving a cell from one partition to another. In k–way, instead of having one gain value you have a vector of gains for every cell, of size equal to the amount of FPGAs-1(connection to all other partitions). This vector covers all arbitrary moves of the cell to the partitions(in our case 4). If a cell does not have a direct connection to a partition then the move is prohibited by setting the gain value to $-\infty$. Because the k-way partitioning together with the immediate locking of a cell after a move tightens the constraints and does not allow flexibility on the redistribution of the cells it would be advantageous to allow k moves for a cell before becoming locked. This would allow the cell to get locked after it has settled to a more fitting partition.

This could result to some thrashing of cells between adjacent partitions and also would increase the runtimes by a factor of k as the locking would now come later.

A problem that arises from k-way partitioning is distant connections of cells (cells that connect through nets that cross more than one FPGA). When we move a cell from one FPGA to another we might find that we create paths through an FPGA that has only routing for the path through the FPGA. This is something that we want to avoid as signals that traverse from one FPGA to another via a third FPGA, are multiplexed and demultiplexed *twice* before they arrive at the destination FPGA thus increasing the delay and lowering the performance of the emulator.



(a) Creating routing paths (b) Creating routing paths

Figure 3.10: Creating routing paths

Consider the example of Figure 3.10. The algorithm will select cell $n4$ to be moved to partition 3, creating a path through partition 2 with only routing of the net $e2$ (Figure 3.10(b). There are some ways to avoid these situations. Initially we can limit the moves to the ones that do not create this situations and also check if the distant cells can be moved to an adjacent partition with total positive gain. Another way would be to provide some tolerance for these situations and let the algorithm allow this situations

temporarily while penalizing the move and giving a higher gain to the move that brings the distant cell closer.

We believe that this approach is more extensible than the connectivity vector as it can deal with the asymmetric topology of our board.

3.3.2 Heterogeneous FPGA

Custom emulator boards may consist of different types of FPGAs, having different amount of IOs and different resources. This means that the capacities of the FPGAs are different and may also have different amount of resources. Thus we need to have an unbalanced partitioning depending on the area of each FPGA. In the algorithm proposed by Fiduccia and Mattheyses, they propose only the use of the area as a criterion to make a move. This can though be changed to other factors (specific resources of one FPGA) or even to more than one criterion. For multiple criteria we need also to provide a hierarchy list which will specify the priority when calculating the cost function (from harder to softer constraints). An example would be a list of the amount of all the resources of the FPGA (block rams,LUTs, etc...) and have multiple ratios for every one of the resources. The selection of the moved cell should then be done depending on the priority that these resources have.

3.3.3 Initial Conditions

The quality of the solution provided from F–M is highly dependent on the initial partitioning provided in step 1. One way to deal with this problem is to provide multiple starting points for the algorithm. This can be done, as suggested before, by hybrid solutions where another algorithm (possibly genetic algorithm) provides candidate starting points for the F–M. An popular alternative is by clustering groups of cells. By providing some design information to the F–M algorithm the algorithm can cluster components of the design, which are inherently more tightly connected, thus trying to optimize only the interconnections outside these clusters. This way it avoids splitting the clusters in the beginning and getting trapped in local optimal solutions.

A more advanced way to perform clustering is by applying multilevel F–M algorithm [2, 23]. In brief a multilevel approach would cluster *recursively* the netlist until a desirable size of hypergraph(netlist) is reached with a coarser grain level. It would then be able to effectively solve this smaller problem by applying a "flat" partitioner (F–M or other) as we reduce the modules (sets of cells) that can be moved.

The smallest hypergraph achieved by the recursive clustering procedure would be partitioned with a very fast initial solution generator (e.g. random) and then iteratively improved by F–M. The resulting partitioning solution is then interpreted as a solution for the next level (less clustered). This is done until we have reached a totally flattened netlist.

This approach has the disadvantage that clustering a design is a non-trivial problem. It therefore introduces another problem on effectively partitioning a arbitrary design. A leading implementation of a multilevel hypergraph partitioning tool *hMetis*[1] has been developed with integrated clustering algorithms and has improved runtime and solution quality. As suggested by [11] though, it has increased complexity due to the complex

heuristics and the non–trivial tunings that it incorporates. We believe that the most significant limitation is that hMetis does not try to produce timing correct solutions nor does it target the other constraints that we have such as prelocking, asymmetric topology and heterogeneous FPGAs.

In our approach we will deal with the initial partitioning problem by providing one level of clustering given by the user. We will not attempt to solve this problem for a flatten netlist with no hierarchical information. We assume that the designer knows the structure of the designs and can specify some information on the hierarchy.

3.3.4 Prelocking a Custom Board

Another constraint that our topology introduces is that parts of our design have to be explicitly mapped on a specific FPGA. This is mainly logic for the connection for the inter–board communication or parts that require routing through one FPGA. We deal with this problem by assigning cells or whole cluster of cells, to a partition and *directly* placing in the LOCKED CELL LIST. This way we prevent the algorithm by moving these components.

3.3.5 Combinatorial Paths – Timing

Until now we have only examined the effect of the number of crossing nets between two FPGAs. The connection of Time Division Multiplexing (TDMX) factor and the number of nets that cross intercluster boundaries is obvious. The more nets that cross FPGA boundaries, the more signals have to travel in a certain amount of time from one FPGA to another. In order to achieve this we have to lower our clock speed to allow a higher TDMX factor. While this is a critical factor for determining the quality of the partitioning it is not our only consideration.

3.3.5.1 Problem of Combinatorial Paths

One important parameter that has to be taken into account during partitioning is the creation of combinatorial paths. Combinatorial paths occur from the creation of fast paths of combinatorial logic that cross partition boundaries more than once. These paths decrease the overall performance of the design by decreasing TDMX and limiting the amount of signals that can cross FPGAs.

We explain this with an example of creation of combinatorial paths. In Figure 3.11 we see a part of a design that consists of two positive edge FF (sequential elements) and a cloud of gates (combinatorial elements) between the two FF. Initially everything is mapped in one FPGA (a). The total delay of the "cloud of gates" is $25ns$ as noted in the case where everything is mapped on one FPGA (a).

Lets assume that the partitioning algorithm splits the design into two partitions cutting *once* between the two FFs(b). The timing constraint that we have to reach from one FPGA to the other is $190ns$ and is defined by the frequency of application clock[1]. During this time frame we must cross the FPGA boundary and also consider

[1]We will not get into many details in the custom emulator's implementation. Instead we will only consider the constraints that are set from the design.

the $25ns$ that is needed from the combinatorial logic. We consider that multiplexing–demultiplexing (MUX-DMUX) has a time period of $10ns$ to transfer one signal.

Lets calculate the TDMX factor for the partitioning solution of case b where the cloud is cut in two parts of delay of $15ns$ and $10ns$ for partitions 1 and 2 respectively. In this example we require $15ns$ to pass the first cloud of gates and $10ns$ for the second, subtracting these two static delays from the total available $190ns$ we are left with $165ns$ to transfer the signals between the two FPGAs. Given that we only have to cross FPGAs once we can have a TDMX factor of 16, spending the remaining time of the application clock.

Now lets assume that at some point the partitioning algorithm puts the combinatorial logic in such a way that a path has to travel from FPGA 1, through FPGA 2 and return back to FPGA 1 as shown in (c). In order to calculate the TDMX factor we follow the same principle. Initially we subtract the static time needed by the combinatorial logic. This time though we have to cross the FPGA boundaries twice. The remaining time of $160ns$ has to divided in two equal segments for the MUX-DMUX. We therefore can only have a TDMX factor of 8.

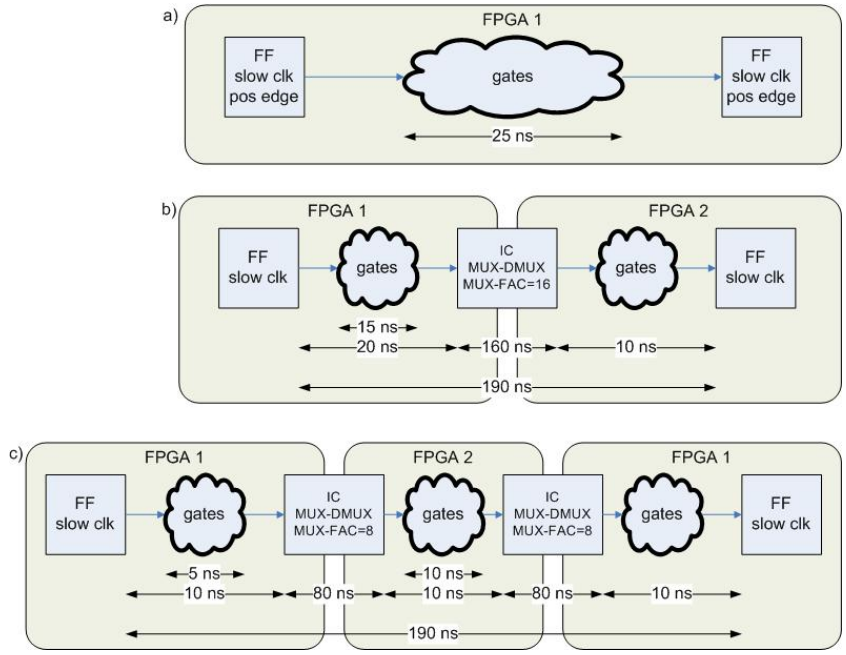


Figure 3.11: Creating combinatorial paths

We can observe that combinatorial paths reduce the amount of signals that can be transferred in 1/2 and as a result reduces the quality of the partitioning.

3.3.5.2 Approaches to Avoid Combinatorial Paths

A simple approach to resolve combinatorial paths would be to assign weights on the nets depending on whether they are part of a combinatorial path or not. The algorithm would also have to dynamically recalculate the weight of all nets, when a cell is moved.

This would result in a much higher complexity as not only all nets would have to be recalculated but also a certain amount of path analysis (which is also an intractable problem) would have to be performed in order to identify the combinatorial paths.

In order to avoid the creation of such situations we propose clustering of the "clouds of gates" and introduction of locks (hard constraints) for the algorithm to take into account while making cell moves.

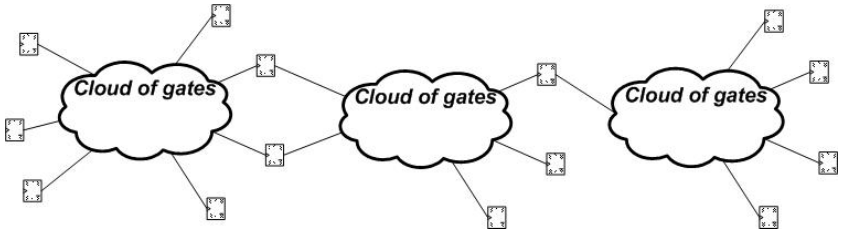


Figure 3.12: Clustering combinatorial logic

The idea is to cluster the clouds of gates around FFs(Figure 3.12). This way we will be able to create clusters of gates, where we can assign flags if they have been cut or not and prevent the creation of combinatorial paths.

After the cell selection from the gain buckets, made by the core F–M algorithm, we check if the selected cell is in a cluster that has already been cut. If a cell belongs to a cluster that has not yet been cut and it does not create a combinatorial path, the algorithm allows the move. Prohibiting the algorithm to move a cell from a cluster that has already been cut will greatly limit the algorithm's possible moves. In order to relax this constraint for the algorithm, we extend the cases where the algorithm is allowed to make certain moves.

If a node is on the border of the cut as is node $n2$ in Figure 3.13, then the algorithm should allow moving the cell across the cut as it will not create a path.

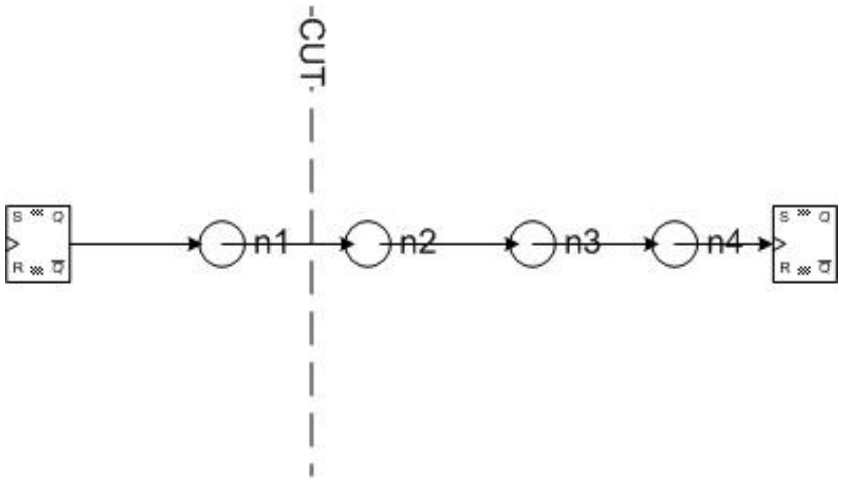


Figure 3.13: Moving border cells across cut

In more complex situations where the border cell has drivers from multiple partitions

the algorithm should perform some limited path analysis before making a move. In the example of Figure 3.14 we see that moving the cell $n2$ across the cut will introduce a combinatorial path. In order to resolve this situation we can prohibit the move. We expect though that the algorithm will run into similar situation quite often and this will influence greatly the flexibility of F–M. Alternatively we can perform some path analysis every time we reach situations like these. By looking up a limited number of stages for the graph we can traceback the path $n2 \rightarrow n6 \rightarrow n5 \rightarrow n3 \rightarrow n2$. If the traceback is deep enough then the algorithm will be able to find the combinatorial path. To resolve it, we can move cell $n2$ and give a higher gain for all the cells in the combinatorial path. We can also calculate the potential gain of pulling all the cells in the path along with cell $n2$ on one side.

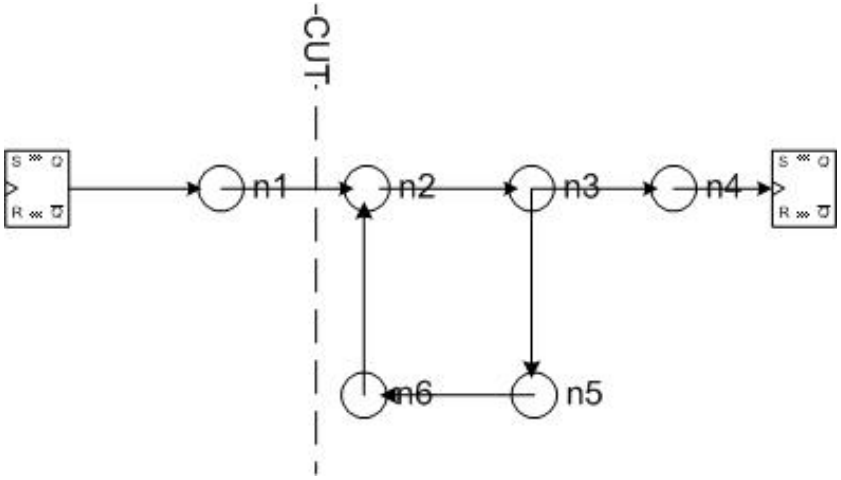


Figure 3.14: Moving border cell creates combinatorial path

This implies that on every non–trivial cell move some path analysis has to be performed. This increases complexity and we expect that this will increase the runtime significantly.

A disadvantage of this method is that in order for it to be applicable, we need to have *multiple, discrete* "clouds of gates". Unfortunately clustering clouds of gates surrounded by FFs is very dependent on the design. Common nets, such as reset nets,clock trees etc. . . , traversing from one cloud to another have to be removed as they make clusters difficult to identify. Also, on a pipelined design, lookahead and feedback signals from one stage of the pipeline to the other, merge the discrete stages making clustering all the more difficult.

We have applied this type of clustering in our design and we have come up with one big cluster of more than 80% of the total design size and several very small clusters. This did not allow us to proceed further with the aforementioned methodology as the main cluster will always be cut.

We believe that this approach, in the cases where it can be applied, can reduce the combinatorial paths significantly especially if we allow some limited path analysis on every move and since the F–M algorithm has linear runtimes we can apply this

methodology without a huge increase of runtime.

The challenge that this methodology introduces in this case is clustering clouds of gates between FF boundaries. Designs with simple pipeline stages are better candidates for direct application of the methodology although the approach can also be applied on more advanced designs. Boundaries can also be formed if we have a very small number of signals travelling from one cloud to another. This though would require that, for the case of complex pipeline designs, the clustering algorithm can be able to distinguish clusters by the amount of crossing signals.

4 Implementation

In this chapter we present the details of the netlist partitioning tool that we have developed. We explain how we have used the underlying graph structures and how we have extended them in order to represent hypergraphs. We also describe the F–M specific data structures that we have used on the implementation.

4.1 Programming Language and Library Selection

We have chosen C programming language for the implementation of the tool. This decision was made mainly because we did not find any good candidate *hypergraph* library, but instead we found that there are stable graph libraries that have already implemented the basic functionality required.

Since hypergraphs are an extension of graphs we saw fitting to use an existing graph library and extend it to include hypergraphs instead of building the tool from scratch. This helped us concentrate on the main objective which was the implementation of the partitioning tool. Another factor for our decision of selecting the *agraph* library (and in consequence C programming language) was performance. Knowing that our workloads would be of substantial size, the library selection had to incorporate fast and efficient data structures and memory management.

> *Agraph is aimed at graph representation; it is not an library of higher-level algorithms such as shortest path or network flow. We envision these as higherlevel libraries written on top of Agraph. Efforts were made in Agraphs design to strive for time and space efficiency [34].*

4.2 Graph Library

We have taken as a starting point the *Agraph* library. The library provides data structures for graphs, subgraphs, nodes and edges. It also provides internally and externally defined attributes for these graph objects. Graph objects may have associated string name–value pairs. The library can internally import *".dot"* files. The *".dot"* file format contains a graph description in the *.dot* language[34]. When a graph file is read, the *Agraph's* parser takes care of the details of the name–value pairs. For more information on the *dot* language see Appendix A.

4.3 Hypergraph extension

Due to the fact that plain graphs lack the "node to multiple nodes" connections of a hyperedge we had to devise an equivalent representation for hyperedges using simple

graphs. To achieve this, we represent hyperedges with DUMMY nodes. For every hyperedge on the netlist we insert a DUMMY node. This node connects to all the adjacent nodes of the hyperedge. In order to make it clear that this is a hyperedge, we name the node as `edge_name@hypernode` and define it as a dummy node in the *cell* attribute of the node. Simple "1 to 1" edges remain as before. The substitution of nets with the "hyperedge equivalent" is depicted in Figure 4.1.

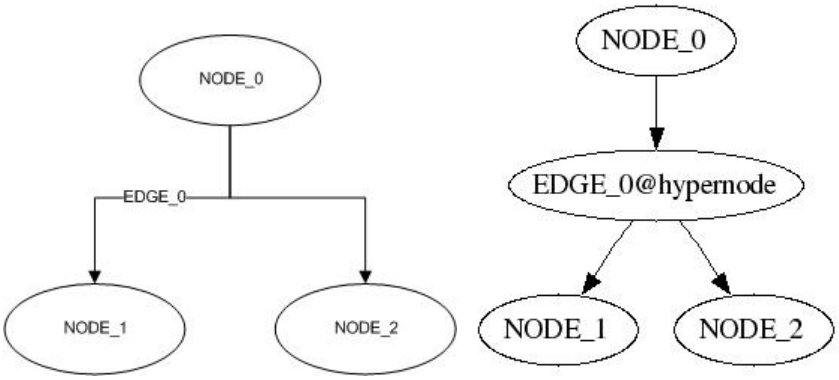


Figure 4.1: Net and Hyperedge equivalent with edges.

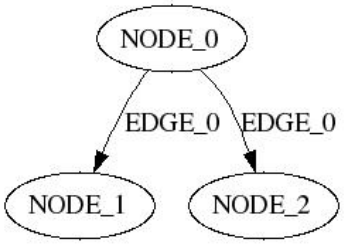


Figure 4.2: Net representation with edges.

The representation of Figure 4.2 where a hyperedge is represented by two different edges of the same name, in this case `EDGE_0`, is not proper for calculating the cut of a netlist. Consider the case where nodes `NODE_1, NODE_2` are on one partition and node `NODE_0` is on the other. The cut cost can not be calculated directly by looking up how many edges are crossing partitions, as multiple edges of the same hyperedge might get counted more than once. This complicates the calculation of the cut cost as well as the gain.

Instead with our scheme with the extra DUMMY node, the calculation of the cut is easier. For simple edges we can determine whether the edge is on the cut by looking up the *head* and *tail* nodes. In the case where we have a hyperedge, the cut can be calculated by looking up if the connections of the DUMMY node are all on the same partition. If not, then the hyperedge belongs to the cut.

4.4 Tool Flow Deployment

We also need to change our design flow in order to incorporate partitioning. The changes on the toolflow can be seen in Figure 4.3. We synthesize the RTL code, producing a flatten EDIF (Electronic Design Interchange Format) file. EDIF is a standardized format for describing a netlist produced by most of the synthesis tools. The EDIF file format is actually very close to being an extended hypergraph description format.

This EDIF netlist is given to the EDIF2dot format converter which exports a *".dot"* file. The exported *".dot"* file doesn't have DUMMY nodes so it can not efficiently describe the netlist. In this intermediate *".dot"* file, a hyperedge is represented with two different edges of the same name, as shown previously in Figure 4.2.

We have developed a preprocessing tool that takes as input this file and modifies the graph to include dummy nodes in the place of the hyperedges. The output of the preprocessing tool is a new *".dot"* file which can be used for the main partitioning tool.

The partitioning tool then imports the *".dot"* file as well as some user defined constraints. The essential inputs for the tool to operate are:

1. The ratio of the desired area distribution among the two partitions $\frac{|A|}{|A+B|}$ and $\frac{|B|}{|A+B|}$ where $|A|, |B|$ are the sizes of the partitions.

2. The percentage of tolerance of the area ratio, defining a range of valid partition area ratios.

3. The replacement–insertion policy for the gain buckets of the algorithm.

The partitioning tool uses the modified algorithm we have described in Chapter 3. It uses the "test run - partitioning run" approach that we have described. It is invoked with the parameters and the user constraints in order to search for the optimal solution of the pass. The tool calculates the optimal cut for that pass of the F–M algorithm. It also gives us information on the whole pass. We then have to invoke the tool with the optimal or a desired (suboptimal) cut point. We do this until the algorithm settles and the tool does not provide a better solution. This usually happens after a few passes (ranging from 1–10) depending on the prepartitioning and the freedom of the cells. In the end, we get as output *".dot"* file along with two lists of cells to be mapped on each partition.

Any prepartitioning and locking of cells is done before running a pass of the tool. We have implemented this with the use of Perl scripts. The user has to specify which are the components that need to be locked and the partition that he wants to preassign them. The scripts modify the *".dot"* file by attaching attributes about the state of the cells (initial partition, free/locked).

We can then split the original EDIF netlist with the information from the partitioned cell lists, into two separate EDIF files. These map these into the respective FPGA by invoking the *Place and Route* tools, creating the bitstreams which we upload on each FPGA.

4.5 Tools

Having described how the tools are deployed in the toolflow we now describe in brief how the tools operate internally.

4.5.1 EDIF2dot and Hypergraph Converter

The EDIF parser (`EDIF2dot`) extracts the cell graph from the flatten netlist of the EDIF Netlist. It creates hyperedges by assigning the same names ,that come from the synthesis tools, to multiple edges of a node. The preprocessing tool has to be run once before the partitioning can operate. It loads the *".dot"* file by calling the library function call so we can then search for multiple edges of same name attached on a node. Having identified a hyperedge we create at that point a special node (DUMMY). As a first step we connect the source (driver) of the edge to the DUMMY node. We then connect the destination cells of the edges with that name (driven cells) to the dummy node and delete the initial direct edges from the driver to the driven cells. The algorithm also checks and reports multiple drivers on the same nets which is a state that we want to avoid.

4.5.2 Prepartitioning scripts

The prepartitioning scripts we have implemented operate by pattern matching on the *".dot"* file. For this reason they have been written in the Perl programming language. The user has to provide two files `prepartition1` and `prepartition2` , where he must specify the name patterns of the cells to be mapped in each partitioned. The script then searches in the input *".dot"* file for the patterns specified in the two prepartition files. When it matches a pattern on a *cell* it assigns the node to the respective partition. A similar script performs the locking mechanism.

4.5.3 Partitioning Tool

After we have run the preprocessor, we can invoke the partitioner. The partitioner requires as parameters the ratios for the two partitions the tolerance and the insertion policy.

For the implementation of the partitioner we initially implemented the F–M data structures and the extended attributes of the nodes. We attached to every node the following attributes :

- gain: The gain value
- weight: The area weight of the cell
- type: The type of cell in the netlist (Flip flop, MUX, LUT, DUMMY, ...)
- partition: In which partition the cell has been preassigned
- lock: If the cell is locked on a partition (has to be used in conjunction with partition attribute)

- head and tail: The previous and next element on the doubly linked list behind the bucket sorted array

The main functionality of the tool is performed by the calls shown in Listing 4.1.

Listing 4.1: partitioning framework

```
1
2 /* Assign nodes to partitions */
3   create_initial_partitioning();
4
5 /* Calculate initial gains - Initialize the data structures */
6   calculate_init_gains();
7
8 /* Invoke partitioning algorithm
9   move_algorithm();
```

create_initial_partitioning

Having imported the graph and initialized the internal data structures, the program checks which nodes have been preassigned a partition and which are locked. The nodes that are locked or just preassigned on a partition get placed directly. We assign these nodes by keeping track of the fill of the partition. Although the weight of preassigned nodes on one partition *can* violate the desired ratio this will have an affect on the behavior of the algorithm as it will force it to make moves from one partition only until it reaches the limit of $ratio + tolerance$.

As we have seen, the F–M algorithm requires an initial partitioning to operate so the free cells have to get an initial partition as well. This is done after having placed the locked/preassigned nodes according to the specified desired ratio. The free cells are assigned to partition 1 until this has reached the desired ratio (specified as parameter on invocation). *Dummy nodes have a weight of 0 and therefore do not influence the ratio.* As soon as we have reached this point we start filling partition 2. In essence, we depend on the synthesis tools for the initial partitioning as we assign partitions to the nodes in the order that the synthesis tool has generated them. We expect that the synthesis tools will keep some hierarchy information of the design by writing cells of the same component as neighbors in the created EDIF netlist files. The program can be extended to incorporate also advanced clustering algorithms in order to place the tightly integrated components together and therefore help the algorithm optimize on the cut. We did not though implement this functionality as this would require the use of clustering algorithms which is not a trivial problem.

calculate_init_gains

Having assigned all the cells in the partitions, we can then calculate the number of cutted nets of this initial distribution of cells. The initial cut cost is calculated by counting the number of simple edges with non–dummy nodes as head and tail plus the number of dummy nodes that connect to nodes that are on more than one partitions (the analogy

with a hyperedge that crosses partition boundaries). The partition of dummy node is not important.

We then calculate *for the initial state* the gain on the cut for moving each node from one partition to the other. Hyperedges increase complexity when we have to calculate the gain because for a potential move of a node that connects to a hyperedge we have to check whether all the connections of the dummy node are on the same partition. In our implementation we improve the runtime by early exiting when we find that the dummy node connects to one node on each partition (and therefore the net is on the cut). This improves the runtime as we do not need to identify where the rest of the cells incident to that net have been placed.

Afterwards, we insert the nodes in the bucket sorted array (Data Structure of Figure 3.3) of the respective partition by pushing every node on the *head* of the doubly linked list. *Dummy nodes are not inserted in the data structure and therefore do not influence the algorithm nor do they increase the runtime.* They are only there to help us calculate the initial gain. While filling the Data Structure of Figure 3.3. We also keep track of the MAX-GAIN-POINTER which points to the bucket of the highest gain, for both partitions.

move_algorithm

Having filled the data structures we call the move_algorithm function. The first step that the function performs is to select source and destination partitions. The node selection priority is:

1. ratio/tolerance
2. gain
3. weight

We select according to the ratio and the tolerance parameters which are treated as hard constraints. If the tolerance allows moves from either partition, we select the move with the highest gain. We also take in account the weight of the node. We select the lightest of the two nodes as that move will not drive the ratio to extremes as fast as moving the heaviest node.

Having selected the partition and in consequence the bucket, the algorithm has to decide on which node from that bucket to select. We can select the head, the tail or a random node of that bucket depending on the policy. The head node can be selected faster than and tail, as we have to traverse the whole linked list before we find the last node. Random selection is implemented by doing an arbitrary number of traversals through the linked list in order to select a random node. Because we do not keep track of the bucket size (amount of nodes in the bucket), we require at least number of traversals equal to the total amount of nodes in order to be able to select from any node in the linked list. This means that we might exhaust the size of the linked list and therefore have to rerun the whole linked list from the beginning. We solve this by increasing a counter every time we traverse from one cell to the next. When we reach the end of the list we set the maximum number of traversals as high as the counter and rerun. We have

implemented the linked list behind the bucket as doubly linked list so that we are able to pop any cell from the list in linear time.

Knowing the selection policy we make the move of the node we update the total cut. We have implemented the locked cell list as an attribute of the node. We set this variable accordingly depending to whether the node is locked from the user(prelocked), locked because of it has moved or free.

We then have to update the gains of the nodes that are connected to the moved node. We did not implement the Data Structure 3.2 by an adjacency matrix as the original F–M algorithm proposes, mostly due to the fact that this is a very large sparse matrix which for very big designs would consume a lot of memory. Instead we search the heads and tails of the edges (or hyperedges) of the edges connecting to the moved node every time we want to find an adjacent node.

We update the gain of the adjacent nodes by calculating it the same way did during the initial gain calculation and storing it to the gain attribute of the node. As soon as we have calculated the new gain of the adjacent node we locate the node in the linked list by providing a direct pointer (not through traversing the linked list). We pop it from the list and place it in the bucket with the new gain index. We always insert the node in the head of the new bucket independently of the selection policy.

We move cells until one of the two partitions becomes empty. Then the algorithm stop and the tool exits. The tool provides a report of the cells that are assigned on the two partitions, the resulting graph file and the optimal cut cost. We use the full test pass approach in order to find the optimal point of the pass of the algorithm. This has the benefit that it gives us the optimal solution of the pass, whereas an early exit after a limited amount of negative moves might not allow the pass to climb out of the local minimum. This is mainly because the runtimes for the size of our netlist are relatively small. In the case that the netlist becomes larger that the full pass runtime becomes tool long we can early exit the tool interactively and perform another pass by passing a user specified parameter as a stopping point.

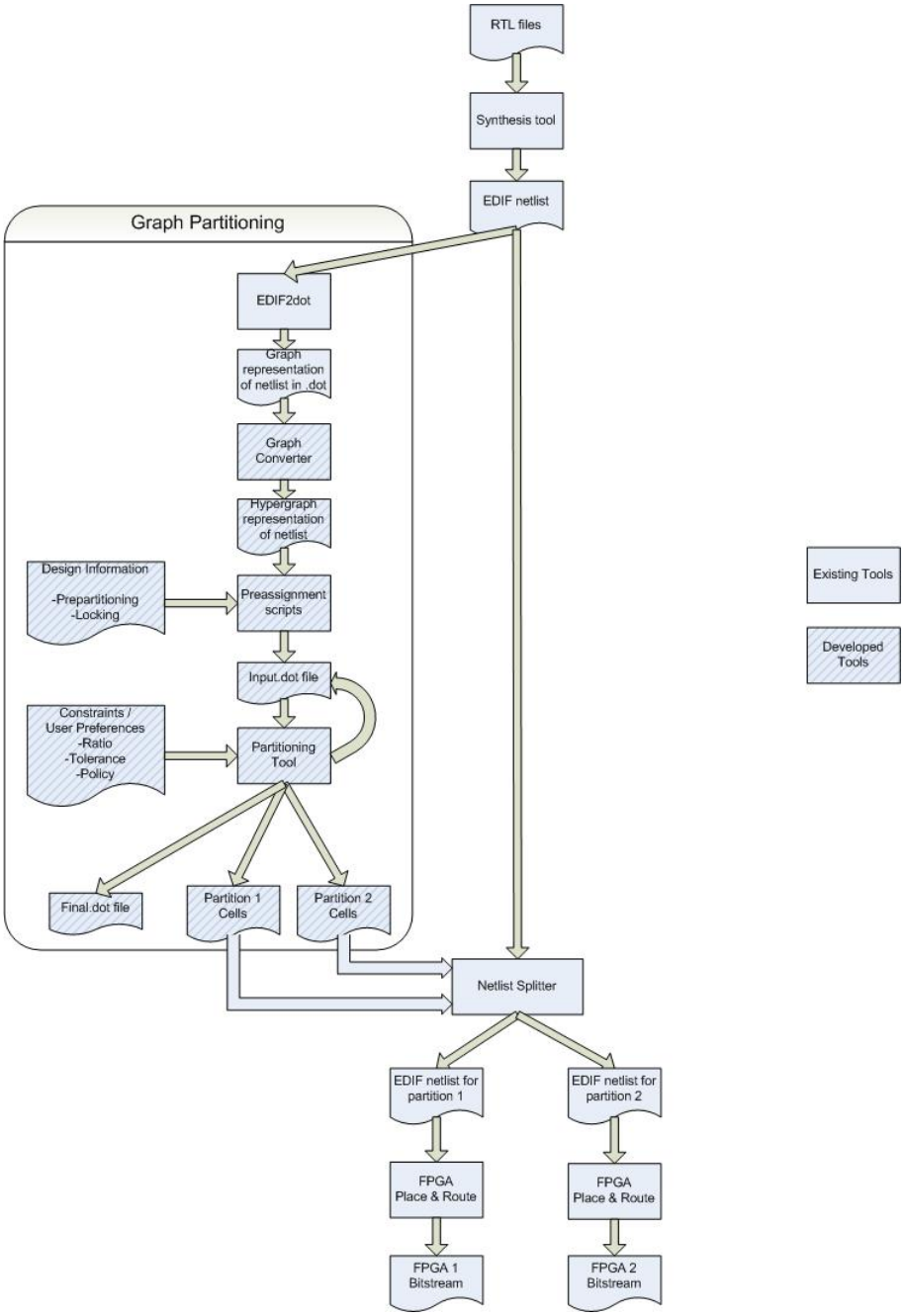


Figure 4.3: Partitioning Toolflow

5 Experimental Results

We have conducted a series of experiments for the tool we have developed. In this chapter we present the results of these experiments. We examine the overall behavior of the partitioner as well as the behavior during every pass of the algorithm. We try different settings and observe the variations on the quality of the resulting partitioning solution. By examining both the runtime behavior of the tool as well as the quality of the solution we explain the behavior we are seeing on the tests we have conducted and provide some insight on the optimizations that could further increase the quality or the speed of the tool. Our tool compared favorably with existing commercial tool for two way partitioning and outperformed it when the user provided to the tool information about the design.

5.1 Custom Emulator

The design that we are going to experiment on is the custom emulator along with the design under test. The total size of the design after synthesis is 205325 cells. Although the tool can have different weights for every cell, for our experiments we consider these cells of equal weight thus the weight of the graph is $|design| = 205235$. The number of hyperedges of the design and thus the number of dummy nodes inserted by the pre-processor are 72399, whereas the total amount of nets are 210144. This means that approximately 34.45% of the design nets are hyperedges and therefore an approach to make a partitioner with simple graph representation would result to a high error in calculations. A similar amount of error is introduced in the cut calculations during an F–M pass of the algorithm, making a simple graph implementation of the F–M algorithm a bad candidate for performing the partitioning.

5.2 Solution Quality

5.2.1 Overall Behavior

We made a number of experiments in order to understand the behavior of F–M from which we will present the major observations. Through the experiments that we have conducted we were able to see the following :

- We can easily see the greedy behavior of F–M algorithm in Figure 5.1 for an initial 50%-50% distribution of cells with 10% tolerance and LIFO policy. The optimal is reached in the first 25% of the total number of moves of one pass.
- In pass 2 (Figure 5.2) we can see that the algorithm reaches the optimal point a lot faster then the first pass (Figure 5.2). This is expected as the algorithm will

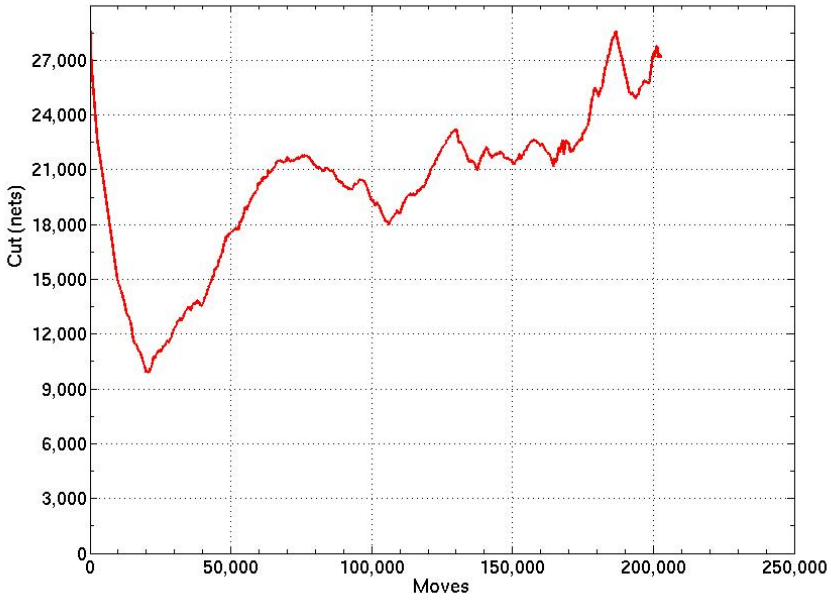


Figure 5.1: LIFO policy, 50-50% distribution, 10% tolerance, Full pass 1

"tighten" partitioning the solution on every pass. We might see some variations of this behavior but the occurrences of these exceptions are not expected to be many.

- For the same random initial partitioning the algorithm has a big improvement on the first pass. In Figure 5.4 we see a big improvement on the first pass from a cut of more than 25000 nets down to 10000 and smaller improvements for the following passes until the algorithm "freezes". For the behavior of the first pass we deduce that the starting point of the algorithm has a lot of slack. This means that the starting point can have a high amount of positive moves, but also high gain moves, for cells that have a high connectivity (fan–in, fan–out), thus greatly improving the solution in the early stages.

- For the passes 2 and higher (Figure 5.3) the algorithm reaches the optimal even earlier. Then it makes a series of bad moves until in the end of the pass reaches solution similar to the starting point. This is because the algorithm has tightened the neighborhood of good moves and only the first few lead to a better solution. Having reached an optimal the algorithm then performs only negative gain moves for the rest of the pass.

- The fact that the algorithm will return in a solution similar to the starting point comes is due to that, we have essentially swapped all the cells from partition 1 to partition 2. At this point it is beneficial to point out that the algorithm might not find the optimal in the beginning of the pass but instead near the end. This can happen when the algorithm has swapped almost all the cells and has reached the

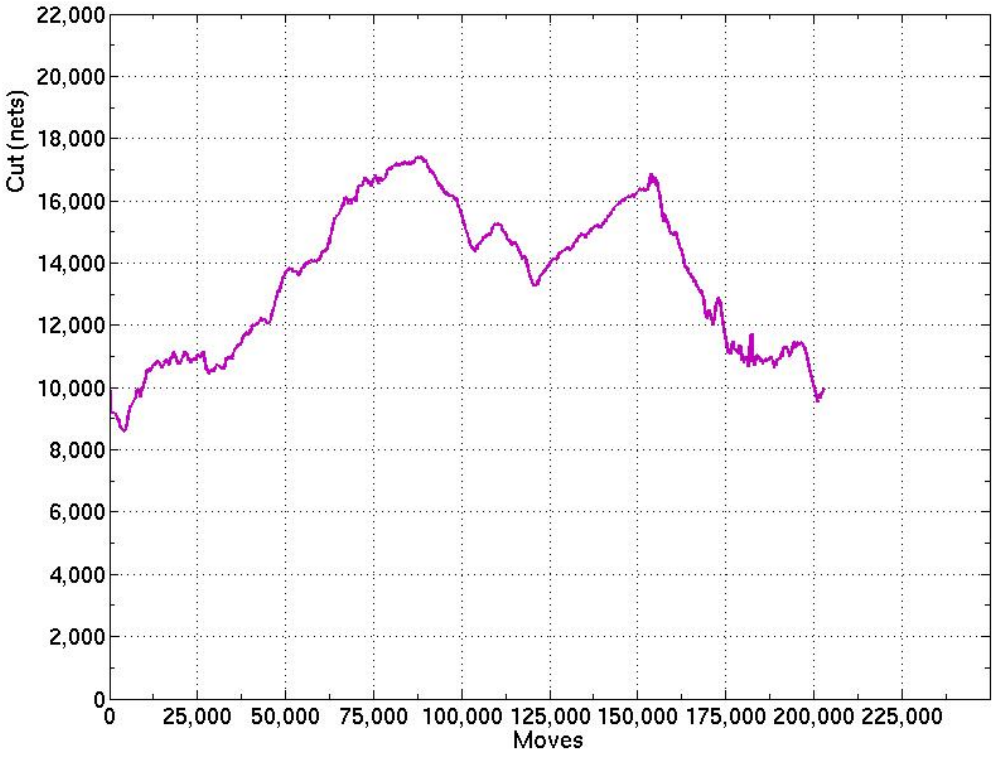


Figure 5.2: LIFO policy, 50-50% distribution, 10% tolerance, Full pass 2

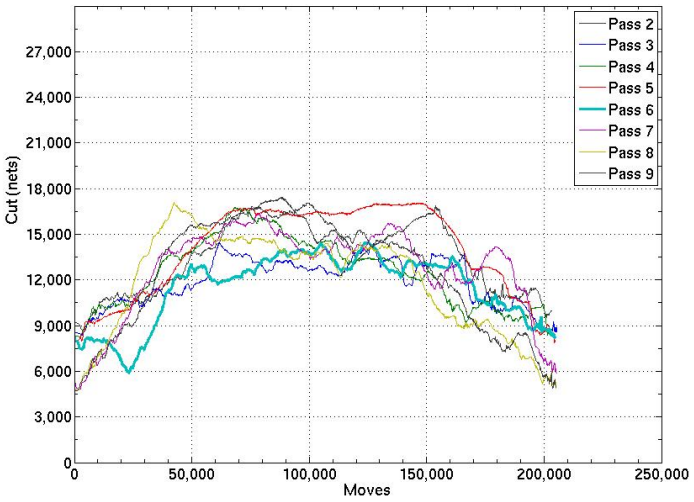


Figure 5.3: LIFO policy, 50-50% distribution, 10% tolerance, Full passes 2-9

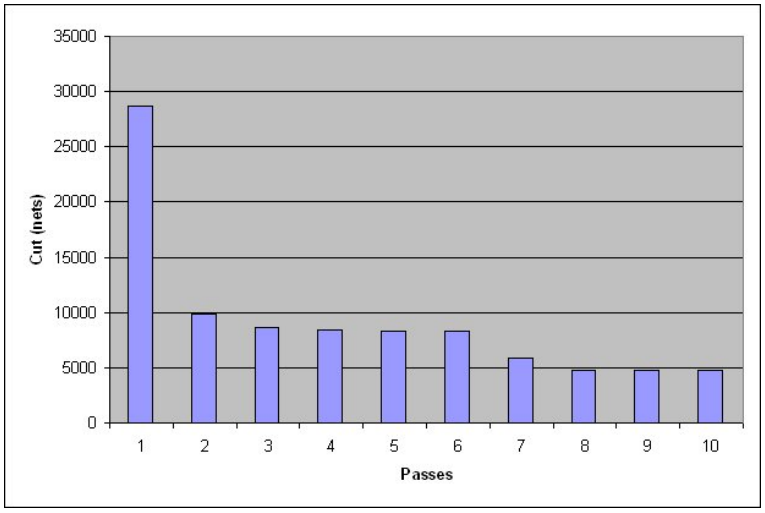


Figure 5.4: Cut after every pass of the algorithm (LIFO policy, 50-50% initial distribution, 10% tolerance)

same neighborhood of the starting point. Although this happens less frequently it can still be seen on Figure 5.5. There, the optimal of the pass is reached at the end after about 200.000 moves.

- The progress on the cut improvement after every pass can be seen in Figure 5.4. We can also observe that the algorithm makes a series of passes (4, 5, 6) of small improvement ($< 100nets$) when it reaches a state that can improve by more than $2000nets$.

- As can be seen from Table 5.1 the algorithm does not necessarily move towards the borders of the ratio (passes 1-5). We see that it moves towards the border and stays there after a few passes (passes 5-10) when it decides to move larger components. The algorithm can perform likewise only if the slack for a pass is loose enough or the initial partitioning has a cut inside a large, tightly integrated component so the algorithm will try to move the cut out of this component.

- In Figure 5.3 we observe something interesting for pass 6. The algorithm was able to make a series of bad moves in the range of 3000–6000 and climb out of the local optimal solution and finding a better solution in the range of 21000 moves. This is more obvious in Figure 5.6.

5.2.2 Policy

In this section we provide an evaluation of the three different replacement–insertion policies (*LIFO, FIFO, RANDOM*) that the user has to select from.

In Figures 5.7, 5.8 and 5.9 we compare the policies during passes 1,2,3 respectively. Although we see that the organization of the sorted buckets has some effect in the whole

	Initial	**Final**				
Pass	Cut(nets)	Cut (nets)	Fill 1 (nodes)	Fill 2 (nodes)	Ratio 1	Ratio 2
1	28642	9898	103033	102292	0.5	0.5
2	9898	8601	102176	103149	0.5	0.5
3	8601	8385	101779	103546	0.5	0.5
4	8385	8284	102049	103276	0.5	0.5
5	8284	8280	103276	102049	0.5	0.5
6	8280	5880	123110	82215	0.6	0.4
7	5880	4806	123008	82317	0.6	0.4
8	4806	4779	123098	82227	0.6	0.4
9	4779	4742	123193	82132	0.6	0.4
10	4742	4741	123192	82133	0.6	0.4

Table 5.1: Partitioning results for: Ratio: 50%-50%, Tolerance: 10%, Policy: LIFO

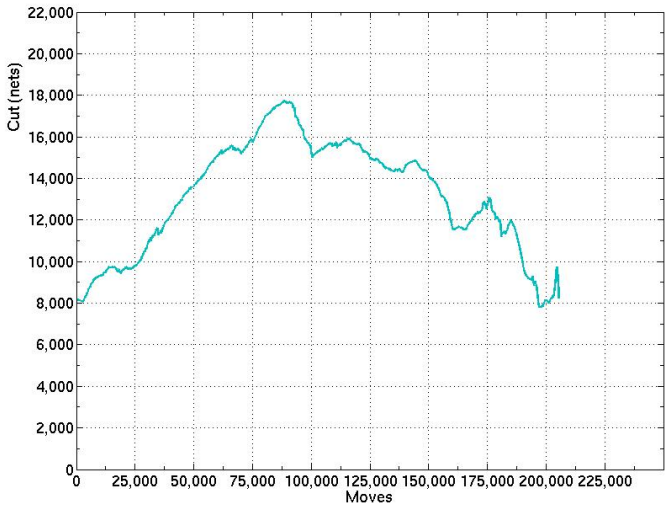


Figure 5.5: LIFO policy, 50-50% distribution, 10% tolerance, Full passes 3

pass of the algorithm, the effect on reaching the optimal point which happens early is rather small. This can be seen in Figures 5.7(b).

Although the literature suggests that LIFO is favorable [3], because of the locality that it imposes, as naturally clustered cells tend to move sequentially, this is not the case in our experiments. This comparison makes clear that, while the bucket sorted array is a core element of the algorithm, the effect that different policies and organizations have in the solution quality are slight in comparison to the main idea of the algorithm, which is the greedy behavior.

We can speculate that the difference between these policies is more on the level of randomness that these policies impose. While LIFO and FIFO policies have some pattern on the insertion of cells (thus limiting the randomness of the selection from the same

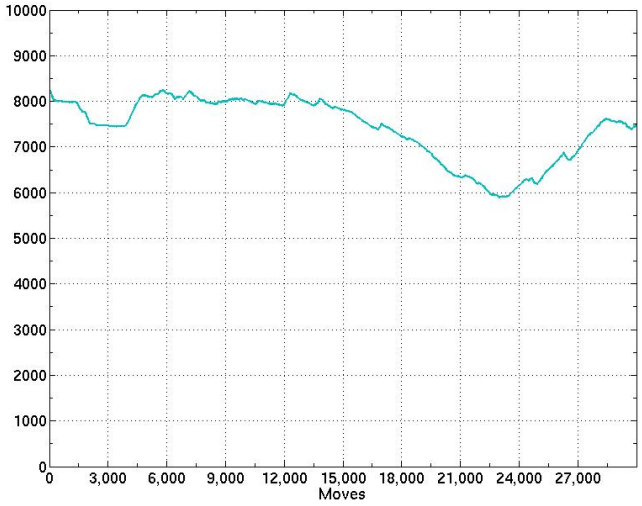


Figure 5.6: Climbing out of a local optimal. LIFO policy, 50-50% distribution, 10% tolerance, partial pass 6

bucket) RANDOM imposes a more free selection.

5.2.3 Tolerance

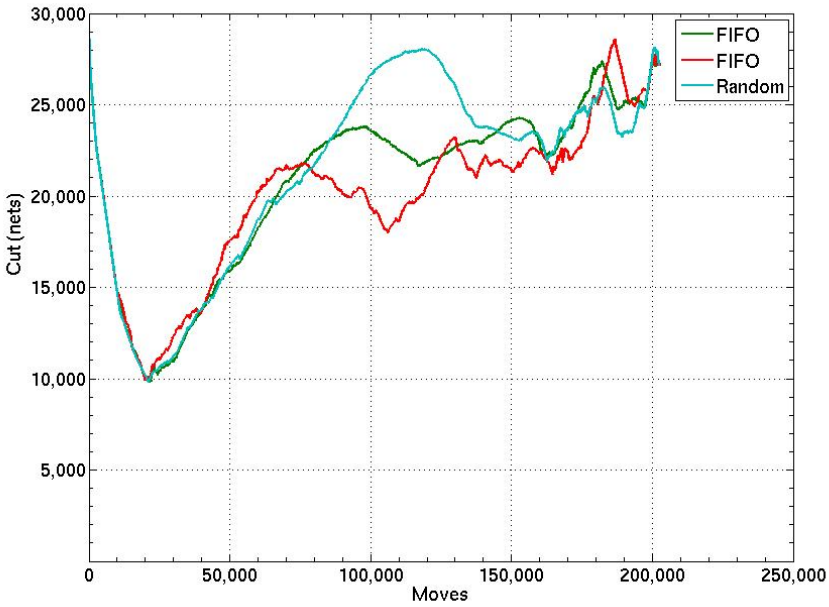
In this section we explore the effect that the tolerance and ratio parameters have on the partitioning solution.

The algorithm selects to move cells from one cluster to the other preserving the area and tolerance constraints. A small value will not allow the algorithm to move consecutively enough cells from one partition to the other thus moving a whole component of a design. We have conducted experiments with same initial cut and target ratios but different tolerances. The results for every of these experiments can be seen in Tables 5.2, 5.3 and 5.4.

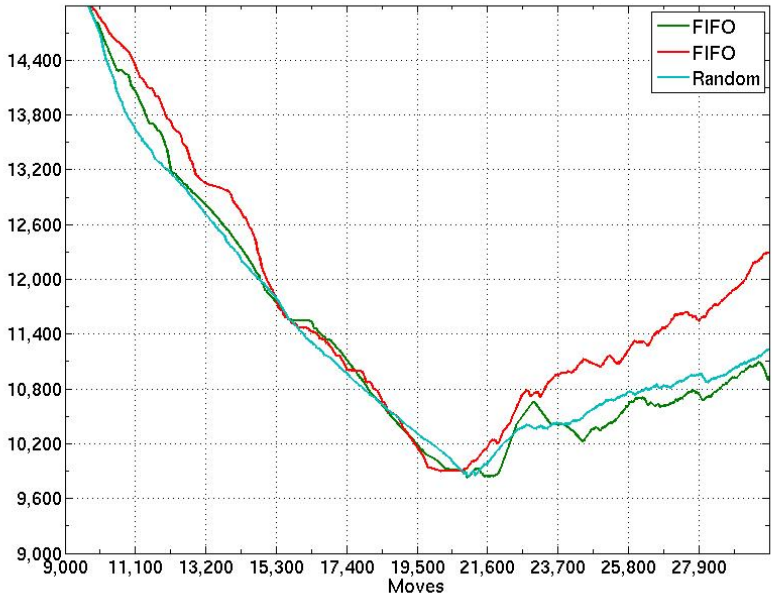
For every case of tolerance we have:

- 0% Tolerace: The algorithm settles in a local optimal position relatively fast (7 passes).
- 10% Tolerace: The algorithm requires more passes (10) to reach an optimal point much lower than the 0% solution.
- 20% Tolerace: The algorithm reaches an optimal after 8 passes and reaches a better solution than the cut provided with 10% tolerance.

We can see that the difference on the final cut between 0% and 10% tolerance is $\frac{8408-4742}{8408} = 42\%$. The improvement between 20% and 10% is $\frac{4436-4741}{4436} = 6,4\%$ while the difference of the area distribution between the 10% and 20% tolerance is only 1%.

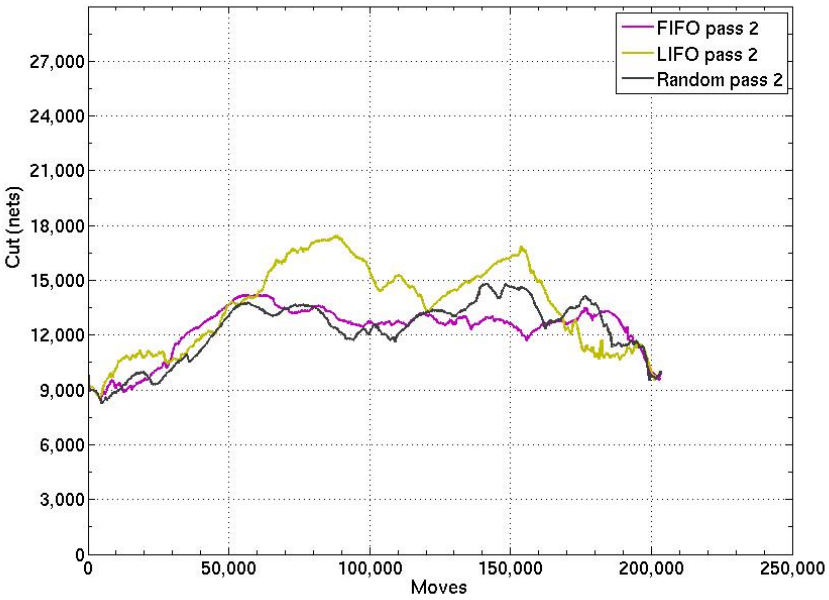


(a) Full 1st pass

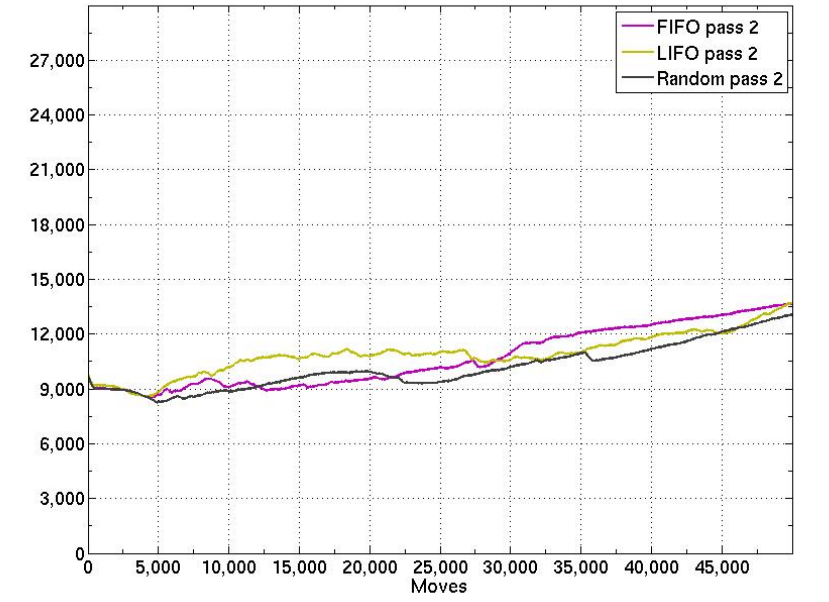


(b) Optimal point

Figure 5.7: Comparison of FIFO, LIFO, random insertion policy. 50-50% distribution, 10% tolerance, optimal point, first pass

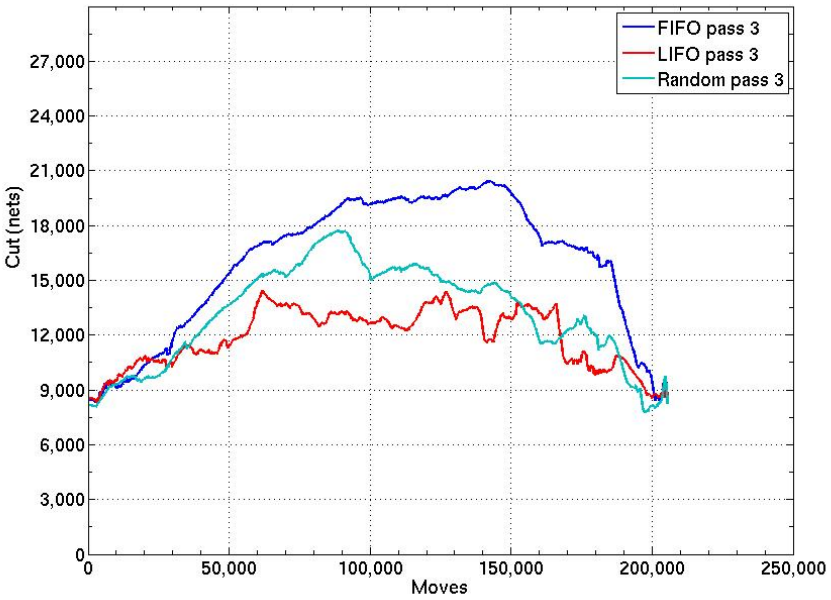


(a) Full 2nd pass

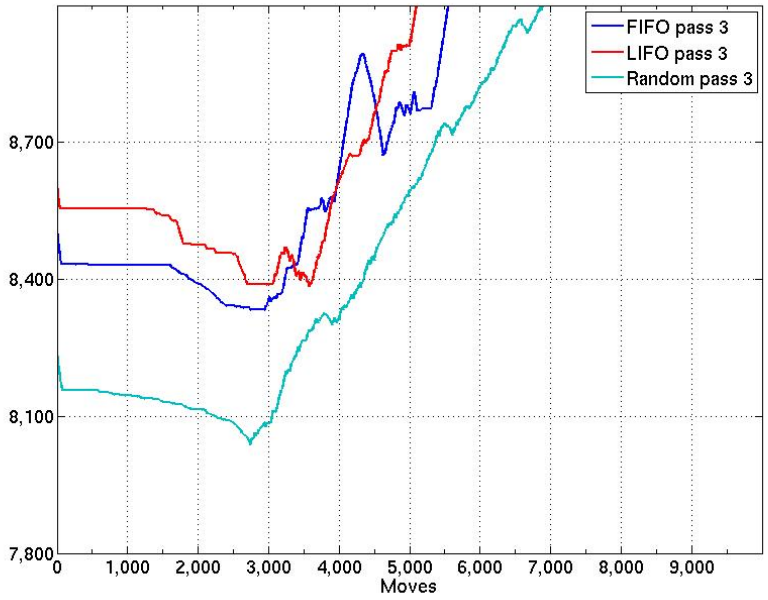


(b) Optimal point

Figure 5.8: Comparison of FIFO, LIFO, random insertion policy *2nd pass*. 50-50% distribution, 10% tolerance.



(a) Full third pass



(b) Optimal point

Figure 5.9: Comparison of FIFO, LIFO, random insertion policy *3rd pass*. 50-50% distribution, 10% tolerance.

	Initial	**Final**		
Pass	Cut (start)	Cut (end)	Ratio 1	Ratio 2
1	28642	10876	0.5	0.5
2	10876	9047	0.5	0.5
3	9047	8745	0.5	0.5
4	8745	8555	0.5	0.5
5	8555	8455	0.5	0.5
6	8455	8408	0.5	0.5
7	8408	8338	0.5	0.5

Table 5.2: Algorithm Progress (Ratio: 50%-50%, Tolerance: 0%, Policy: LIFO)

	Initial	**Final**		
Pass	Cut (start)	Cut (end)	Ratio 1	Ratio 2
1	28642	9898	0.5	0.5
2	9898	8601	0.5	0.5
3	8601	8385	0.5	0.5
4	8385	8284	0.5	0.5
5	8284	8280	0.5	0.5
6	8280	5880	0.6	0.4
7	5880	4806	0.6	0.4
8	4806	4779	0.6	0.4
9	4779	4742	0.6	0.4
10	4742	4741	0.6	0.4

Table 5.3: Algorithm Progress (Ratio: 50%-50%, Tolerance: 10%, Policy: LIFO)

	Initial	**Final**		
Pass	Cut (start)	Cut (end)	Ratio 1	Ratio 2
1	28642	9898	0.5	0.5
2	9898	8601	0.5	0.5
3	8601	8535	0.5	0.5
4	8535	8279	0.5	0.5
5	8279	6663	0.51	0.49
6	6663	5032	0.6	0.4
7	5032	4436	0.61	0.39
8	4436	4424	0.61	0.39

Table 5.4: Algorithm Progress (Ratio: 50%-50%, Tolerance: 20%, Policy: LIFO)

An interesting point is that pass 5 of the 20% tolerance provides a solution of 6663 cuts with distribution of 51-49% which is significantly lower than the result that 0% reaches.

We can therefore deduce that it is *advisable* that we set a relatively large tolerance value even if we want to partition a design with strict area constraints in order to allow

the algorithm to move more freely around the search space. The end result can still be within the limits that we want as the tolerance value does not suggest that the algorithm will move to the boundaries of the allowed area. We can safely assume that this happens because the 0% tolerance does not allow moving of large components to one partition and then balancing the ratio.

	Initial	**Final**		
Pass	Cut (start)	Cut (end)	Ratio 1	Ratio 2
1	14768	5044	0.41	0.59
2	5044	4019	0.41	0.59
3	4019	3721	0.41	0.59
4	3721	3713	0.41	0.59
5	3713	3709	0.41	0.59

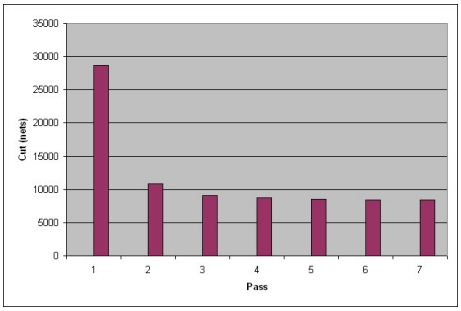
Table 5.5: Algorithm Progress (Ratio: 40%-60%, Tolerance: 10%, Policy: LIFO)

	Initial	**Final**		
Pass	Cut (start)	Cut (end)	Ratio 1	Ratio 2
1	34669	11317	0.61	0.39
2	11317	10157	0.64	0.36
3	10157	9359	0.63	0.37
4	9359	8692	0.64	0.36
5	8692	8587	0.64	0.36
6	8587	8455	0.65	0.35
7	8455	8403	0.64	0.36
8	8403	8375	0.64	0.36

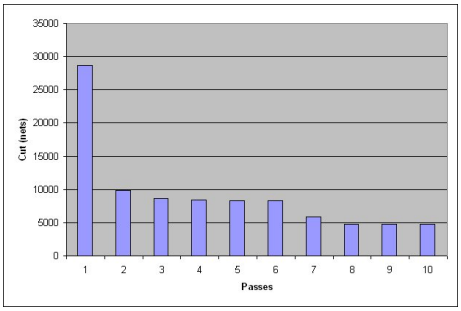
Table 5.6: Algorithm Progress (Ratio: 60%-40%, Tolerance: 10%, Policy: LIFO)

We have also done some experiments with a $40-60\%$ and $60-40\%$ distribution with 10% tolerance the results of which can be seen on Tables 5.5, 5.6. We note that the starting points are different for these two policies as we have explained during the implementation the initial cell assignment is done by the parser and the order it has has placed the cells in the netlist. If no prior cell preassignment has been done then it fills the partitions until they reach the target ratios. Therefore the initial partitioning is 40-60% and 60-40% respectively for the two experiments. Nevertheless the result that we take for 41-59% distribution is 3717. This result is even better then the one achieved by the 50-50% target distribution and 20% tolerance by $4436 - 3717/4436 = 16\%$.We provide the progress of all the experiments in the Figure 5.10

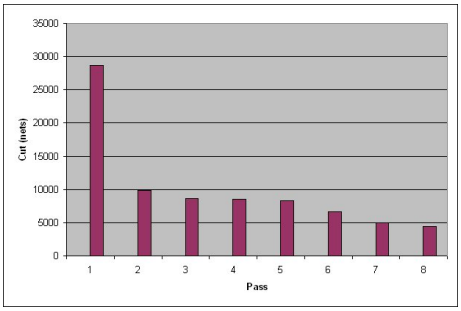
We can conclude that tolerance is an important parameter of our partitioning algorithm and experiments have showed us that large values of tolerance improve on the partitioning quality.



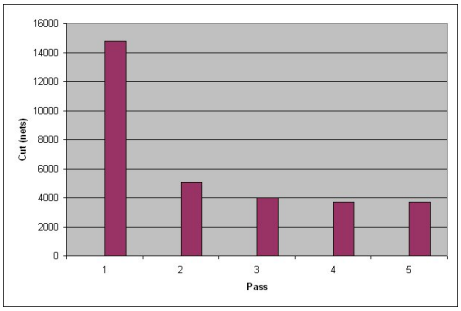
(a) 50%-50%, 0% tolerance



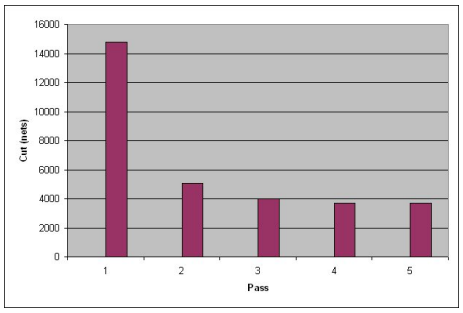
(b) 50%-50%, 10% tolerance



(c) 50%-50%, 20% tolerance



(d) 40%-60%, 10% tolerance



(e) 60%-40%, 10% tolerance

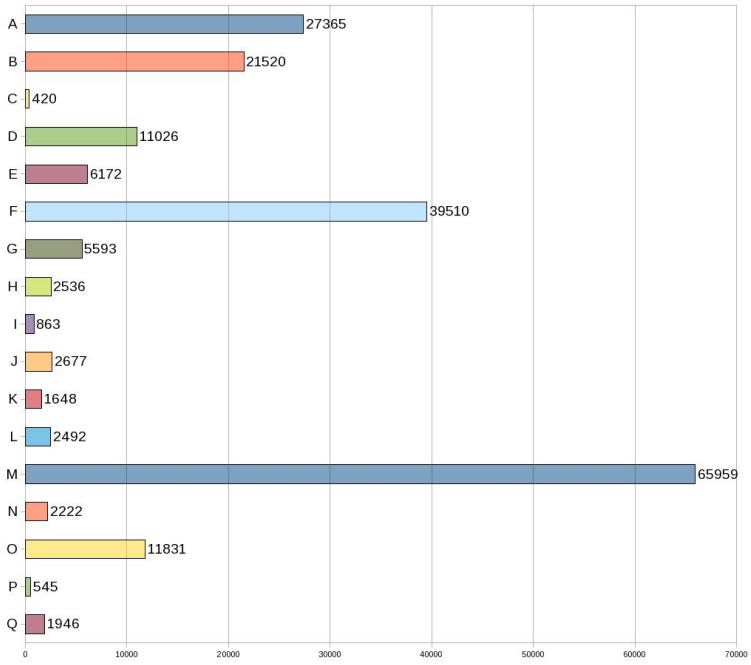
Figure 5.10: Progress of passes with different tolerances. (LIFO policy)

5.2.4 Clustering

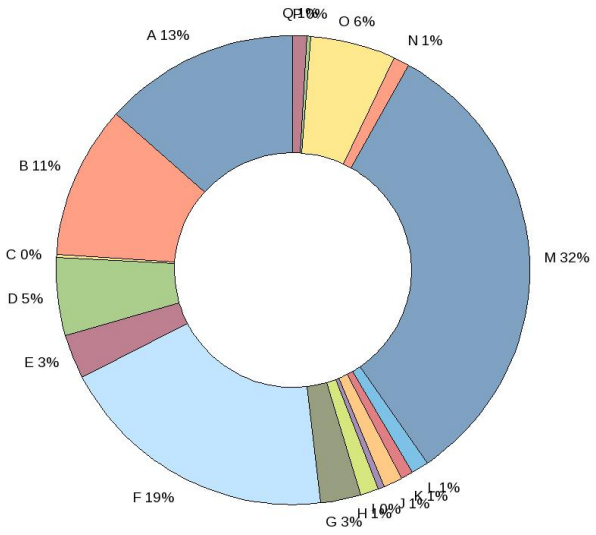
In this part we will see the improvement that clustering can give on the algorithm. As there is no advanced clustering algorithm implemented in the tool we have applied different initial partitions based on the information that we had for the design under test.

In Table 5.7 and Figure 5.11 we can see the 17 main components and their respective sizes for the design under test.

In the following experiments we have initially preallocated these components and have run the tool with that initial partitioning. In these experiments we can see the difference that an automated tool that implements an heuristic algorithm has in comparison to



(a) Component sizes.



(b) Size in cells

Figure 5.11: Relative size of components.

manual partitioning of components. All the experiments have been conducted with target parameters of:

- 50%–50% distribution
- 10% tolerance

Component	Weight
A	27365
B	21520
C	420
D	11026
E	6172
F	39510
G	5593
H	2536
I	863
J	2677
K	1648
L	2492
M	65959
N	2222
O	11831
P	545
Q	1946

Table 5.7: Component weights

- LIFO policy

In Table 5.8 we see the following:

- initial cut and end cut after all passes
- percentage of partition 1 and amount of cells that this corresponds
- the components that partition 1 has. The rest of the components have been placed on partition 2.

The most important observation for clustering is the initial cut. While for the "random" initial partition that the EDIF netlist and the tool provide, the component clustering has 5–6 times less cutted nets.

It is interesting to see from this table that in all cases except the first the algorithm has improved the cut by more than 50%. It is interesting to note that for cases 1–6 the algorithm was *forced* to move nodes from partition 1 to partition 2 as it was out of the $target\ area + tolerance$ boundaries. We must remember that the algorithm does not consider moves from a partition if this condition is broken. It will only move cells from one partition to another until it has fullfilled this hard constraint and then it can select from both partitions.

This can be used to bias the partition distribution as we can provide a very unbalanced initial partitioning solution and set the tolerance and target ratio so that the algorithm will "pull" the cells with the least cost to the empty partition and therefore create good partition from there on.

		Cut	Partition 1(%)	Weight	Components
Clustering 1	Start	729	13	27365	A
	Finish	729	15	31037	
Clustering 2	Start	2651	24	50239	A,B
	Finish	1671	26	53280	
Clustering 3	Start	3183	25	50659	A,B,C
	Finish	1642	26	53443	
Clustering 4	Start	4003	30	61685	A,B,C,D
	Finish	1642	32	64732	
Clustering 5	Start	4197	33	67857	A,B,C,D,E
	Finish	2361	35	70941	
Clustering 6	Start	4497	33	68623	A,B,C,D,E,G
	Finish	2588	35	71892	
Clustering 7	Start	5058	53	110646	A,B,C,D,E,F,G,H
	Finish	2478	53	109102	
Clustering 8	Start	5058	54	111509	A,B,C,D,E,F,G,H,I
	Finish	2478	54	110490	

Table 5.8: Clustering solutions

5.3 Runtime

As the algorithm calculates the gain of all the cells adjacent to the moved cell a very large net connecting many cells would mean that for every move of a node on that net we would have to calculate all the other cell's gain. For global signal as the clock tree and the global reset signal the impact of the runtimes is much bigger. We have implemented early exiting for the recalculation of the gain of the adjacent cells, so that if cells have been found in both partitions the check stops and the cut increases by 1. Still, in order to improve the runtime of our F–M implementation we have removed the clock net as well as the global reset signal. The runtimes of the netlist including clock/reset in comparison to that of the removed clock/reset net have been consistently $\times 10$ bigger.

In Figure 5.12 we can see the difference in runtime for *one* full pass of the algorithm. It is therefore advisable to remove big nets from a netlist.

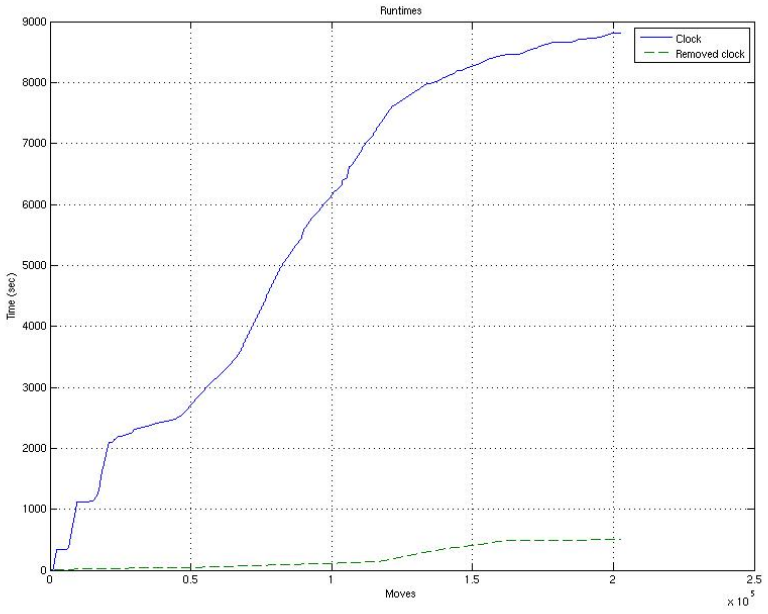


Figure 5.12: Runtime comparison (LIFO policy)

6 Conclusion

The main objective for this thesis was to investigate, implement and test solutions on the partitioning problem of prototyping designs on a custom hardware emulator. This chapter provides a summary of this inquiry, implementation and . We provide an evaluation of the approach that we have selected as well as recommendations on further improvement and future work.

6.1 Summary

In Chapter 2 we presented an overview of the trends of verification technology for system design in order to provide some general information of the platform that we will be using. We briefly saw :

- Instruction Set Simulation
- Cycle Accurate Simulation
- Emulation
- Silicon Prototyping

Given that our working platform was an *emulator*, we saw the advantages and disadvantages that this platform has compared to its "rivals" for design validation. At that point we observed that an important parameter of the performance of an emulator is the capacity of the reconfigurable devices that the emulator and the design under test is mapped. As modern designs have a high complexity to be mapped on one FPGA, partitioning techniques are required.

Having explained the reasons that partitioning is required, we examined the underlying FPGA platform that the emulator is deployed. We presented the architectural, interconnection and topology specifications of the underlying FPGAs that are of interest to us.

As the partitioning quality of the design highly affects the performance of the emulator speed, selection of the partitioning algorithm is crucial. We stated the reasons that netlist partitioning is favored compared to HDL partitioning and we gave the necessary graph background for netlist partitioning algorithms. The available algorithms for netlist partitioning were then presented. After having performed an evaluation of the algorithms according to the data provided by the available literature, the Fiduccia–Mattheyses algorithm was selected as the core algorithm for partitioning our design.

In Chapter 3 we have explained what are our specific issues that arise from partitioning our design in multiple FPGAs. We then gave a detailed description of the F–M core

algorithm and an extensive analysis of the behavior of F–M. We described our emulator specific constraints:

- Multiple FPGA board – Asymmetric topology
- Heterogeneous FPGAs
- Initial conditions for partitioning
- Prelocking cells in FPGAs
- Combinatorial paths – timing constraints

and formulated or proposed solutions that we wrapped around the core F–M algorithm.

In Chapter 4 we presented our implementation approach of the F–M algorithm. We have explained the choices that we have made during the implementation of the software tools that we have developed. We then described the tool functionality and the necessary toolflow modifications that we have made in order to incorporate the partitioning toolset. In the end of the chapter we also gave an explanation of the internals of the partitioning tool.

In Chapter 5 we presented the results that we got from our experiments on our custom emulator. We explain partitioning tool behavior and we observe the strengths and weaknesses of the F–M algorithm. We evaluated the quality of the results of the partitioning tool with different parameters. We tested the tool with different starting partitioning, selection policies and initial clustering and we presented the results of these tests. We also provided a runtime evaluation of the tool and ways to improve on the speed performance as well as the partitioning quality.

6.2 Open issues

Although the tools that we have developed are quite accurate on the cut cost calculation there seem to be some bugs in the prepartitioner and the graph converter which insert some small amount of error. In specific :

- The EDIF parser creates some edges with identical names in different places of the netlist which the graph converter sees as multiple drivers of the same hyperedge.
- It also seems to strip a small amount of cells from the information that they have (library family, weight, etc. . .) and therefore these cells can not be preassigned to a partition. They will though be assigned by the tool but this is still unwanted behavior as the user might want to lock the specific cells.
- Also the prepartitioning is done by pattern matching of the names of the components and some cell names may share common patterns so pattern matching is relatively unreliable.

6.3 Future work

Netlist partitioning has been addressed by F–M with relatively good results. It is evident though that in order to obtain an optimal partitioning the algorithm needs to have information of the design components. This can be done by either providing a relatively good coarse grained initial partitioning or by implementing multilevel Fiduccia–Mattheyses extensions as tools like hMetis[1].

Other approaches on partitioning for custom FPGA boards would be the implementation of multi–resource aware solutions as we have proposed in Chapter 3.3.2.

The problem of providing timing clean partitioning solutions still needs to be resolved by more coherent solutions. We have proposed a possible solution for this but the clustering of "clouds of gates" between sequential elements is not always feasible as is not in our case. The quality of results for our proposal on designs where we can apply this methodology, are still to be tested.

Bibliography

[1] *hMetis, A Hypergraph Partitioning Package.* [cited at p. 40, 73]

[2] Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel circuit partitioning. In *Design Automation Conference*, pages 530–533, 1997. [cited at p. 40]

[3] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: A survey. Technical report, UCLA Computer Science Department, Los Angeles, 1998. [cited at p. 59]

[4] S. Areibi and A. Vannelli. Advanced search techniques for circuit partitioning. In P. M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 77–96. AMS, 1994. [cited at p. 25]

[5] Shawki Areibi and Anthony Vannelli. Circuit partitioning using a tabu search approach. In *ISCAS*, pages 1643–1646, 1993. [cited at p. 25]

[6] Shawki Areibi and Anthony Vannelli. An efficient solution to circuit partitioning using tabu search and genetic algorithms, November 13 1994. [cited at p. 25]

[7] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. In *DAC '89: Proceedings of the 26th ACM/IEEE conference on Design automation*, pages 775–778, New York, NY, USA, 1989. ACM. [cited at p. 22]

[8] Thang Nguyen Bui, F. Thomson Leighton, Soma Chaudhuri, and Michael Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987. [cited at p. 22]

[9] Thang Nguyen Bui and Byung Ro Moon. A fast and stable hybrid genetic algorithm for the ratio-cut partitioning problem on hypergraphs. In *DAC*, pages 664–669, 1994. [cited at p. 26]

[10] Thang Nguyen Bui and Byung Ro Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, 1996. [cited at p. 22]

[11] A. Caldwell, A. Kahng, and I. Markov. Improved algorithms for hypergraph bipartitioning, 2000. [cited at p. 40]

[12] Chung-Kuan Cheng and Yen-Chuen A. Wei. An improved two-way partitioning algorithm with stable performance [VLSI]. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 10(12):1502–1511, 1991. [cited at p. 36]

[13] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 3rd edition, February 2006. [cited at p. 13]

[14] S. Dutt. New faster kernighan-lin-type graph-partitioning algorithms. In Michael Lightner, editor, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 370–377, Santa Clara, CA, November 1993. IEEE Computer Society Press. [cited at p. 22]

[15] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *19th —DAC—*, pages 175–181, 1982. [cited at p. 22, 37]

[16] Luca Formaggio, Franco Fummi, and Graziano Pravadelli. A timing-accurate hw/sw co-simulation of an iss with systemc. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 152–157, New York, NY, USA, 2004. ACM. [cited at p. 5]

[17] Giorgio Gallo, Giustino Longo, Sang Nguyen, and Stefano Pallottino. Directed hypergraphs and applications. [cited at p. 16]

[18] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997. [cited at p. 25]

[19] Fred Glover. Tabu search– part I. *ORSA Journal on Computing*, 1(3):190–206, 1989. [cited at p. 25]

[20] Lars W. Hagen, Dennis J.-H. Huang, and Andrew B. Kahng. On implementation choices for iterative improvement partitioning algorithms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(10):1199–1205, 1997. [cited at p. 38]

[21] John H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975. [cited at p. 26]

[22] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, November-December 1989. [cited at p. 24]

[23] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. Technical report, 1997. [cited at p. 40]

[24] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970. [cited at p. 20]

[25] Ho Young Kim and Tag Gon Kim. Performance simulation modeling for fast evaluation of pipelined scalar processor by evaluation reuse. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 341–344, New York, NY, USA, 2005. ACM. [cited at p. 5]

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983. [cited at p. 24]

[27] Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Computers*, 33(5):438–446, 1984. [cited at p. 24, 37, 38]

[28] Helena Krupnova and Gabriele Saucier. FPGA-based emulation: Industrial and custom prototyping solutions. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *FPL*, volume 1896 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2000. [cited at p. 29]

[29] Jie Liu, Marcello Lajolo, and Alberto Sangiovanni-Vincentelli. Software timing analysis using hw/sw cosimulation and instruction set simulator. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 65–69, Washington, DC, USA, 1998. IEEE Computer Society. [cited at p. 5]

[30] Rose F. Liu. Axcis: Rapid processor architectural exploration using canonical instruction segments. Master's thesis, S.B., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 2005. [cited at p. 6]

[31] Grant Martin Louis Scheffer, Luciano Lavagno. *EDA for IC System Design, Verification, and Testing*, volume 1 of *Electronic Design Automation for Integrated Circuits Handbook*. CRC, 1 edition, March 2006. [cited at p. 7]

[32] MIT. `http://web.mit.edu/6.111/www/f2005/tutprobs/synthesis_noanswers.html`. [cited at p. vii, 10]

[33] D.S. Johnson M.R. Garrey. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. [cited at p. 19]

[34] Stephen C. North. *Agraph Tutorial*. AT&T, Shannon Laboratory, Florham Park, NJ, USA, July 2002. `http://www.graphviz.org/Documentation/Agraph.pdf`. [cited at p. 47]

[35] National Institute of Standards and Technology. `http://www.nist.gov/dads/HTML/bucketsort.html`. [cited at p. 23]

[36] National Institute of Standards and Technology. `http://www.nist.gov/dads/HTML/queue.html`. [cited at p. 37]

[37] Chan-Ik Park and Yun-Bo Park. An efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(11):1686–1694, 1993. [cited at p. 36]

[38] Youssef Saab. Combinatorial optimization by dynamic contraction. *Journal of Heuristics*, 3(3):207–224, 1997. [cited at p. 22]

[39] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, January 1989. [cited at p. 24, 37, 38]

[40] Schuur. Classification of acceptance criteria for the simulated annealing algorithm. *MOR: Mathematics of Operations Research*, 22, 1997. [cited at p. 24]

[41] Navaratnasothie Selvakkumaran, Abhishek Ranjan, Salil Raje, and George Karypis. Multi-resource aware partitioning algorithms for FPGAs with heterogeneous resources. In *Proceedings of the 41st Annual conference on Design Automation (DAC-04)*, pages 741–746, New York, June 7–11 2004. ACM Press. [cited at p. 24]

[42] Harold P.E. Stern and Samy A. Mahmoud. *Communication Systems: Analysis and Design*. Prentice Hall, 2004. [cited at p. 11]

[43] Wikipedia The Free Encyclopedia. `http://en.wikipedia.org/wiki/Bucket_sort`. [cited at p. 23]

[44] Wikipedia The Free Encyclopedia. `http://en.wikipedia.org/wiki/Hardware_emulation`. [cited at p. 7]

Dot language

A

An example of the *.dot* language is shown in A.1. The *".dot"* file is consisted by a header that specifies the type of graph. In Listing A.1 the graph is a non–strict digraph specified in line 1. In lines 2–6 we specify the external attributes and the default values that these get for all the nodes, we can do the same for edges, graphs and subgraphs. For a graph object that has different values than the defaults, the node has to be declared explicitly and has to have the different attributes attached to it.

In the example of Listing A.1, nodes `NODE_0, NODE_1, NODE_8` have weights of $2, 4, 6$ respectively so they have to be declared explicitly, otherwise weight of 1 is assumed. To preserve the information provided from the EDIF netlist we also annotate the type of the cell as well as the library where that type is defined.

From the declaration of nodes we see that `NODE_5` has been preassigned in *partition 1* but is free to be moved by the partitioner whereas `NODE_4` has been locked in *partition 1* and will be excluded from the partitioning procedure.

For the edges of the graph we have only the two special attributes `label` and `edge`. The `key` attribute signifies the name of the edge used internally when we perform a query for an edge name.

Listing A.1: Dot example

```
1  digraph top {
2  node [cell=DEFAULT,
3          lib=DEFAULT,
4          lock=NONE,
5          partition=NONE,
6          weight=1
7  ];
8
9  NODE_0 [weight=2,cell=LUT];
10 NODE_1 [weight=4, cell=FF];
11 NODE_2 [cell=LUT];
12 NODE_3 [cell=LUT];
13 NODE_4 [partition=1, lock=LOCKED, cell=FF];
14 NODE_5 [partition=1, cell=MUX];
15 NODE_6 [cell=FF];
16 NODE_7 [cell=LUT];
17 NODE_8 [weight=6, cell=FF];
18
19
20 NODE_0 -> NODE_2 [label=EDGE_0, key=EDGE_0];
```

```
21 NODE_0 -> NODE_4 [label=EDGE_0, key=EDGE_0];
22 NODE_0 -> NODE_5 [label=EDGE_1, key=EDGE_1];
23 NODE_0 -> NODE_7 [label=EDGE_2, key=EDGE_2];
24
25 NODE_1 -> NODE_8 [label=EDGE_3, key=EDGE_3];
26
27 NODE_2 -> NODE_5 [label=EDGE_4, key=EDGE_4];
28
29 NODE_3 -> NODE_4 [label=EDGE_5, key=EDGE_5];
30 NODE_3 -> NODE_5 [label=EDGE_5, key=EDGE_5];
31 NODE_3 -> NODE_6 [label=EDGE_6, key=EDGE_6];
32 NODE_3 -> NODE_7 [label=EDGE_7, key=EDGE_7];
33 NODE_3 -> NODE_8 [label=EDGE_8, key=EDGE_8];
34
35 NODE_4 -> NODE_5 [label=EDGE_4, key=EDGE_4];
36 NODE_4 -> NODE_7 [label=EDGE_9, key=EDGE_9];
37
38 NODE_5 -> NODE_7 [label=EDGE_10, key=EDGE_10];
39
40 NODE_6 -> NODE_8 [label=EDGE_11, key=EDGE_11];
41 }
```

A visual representation of the graph described in Listing A.1 can be seen in Figure A.1. To produce the graph figures described by the *"dot"* language we have used the *dot* and *dotty* tools provided with *graphviz* package from where we also acquired the library.

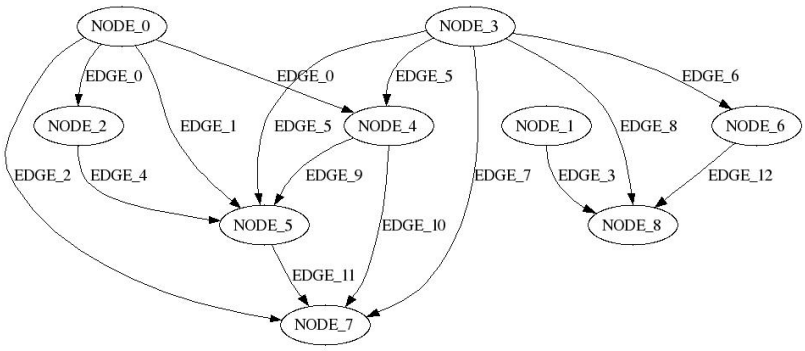


Figure A.1: Graph

Index

Curriculum Vitae

Nikolaos Mitas was born in Kavala, Greece on June 7, 1979. He received his Diploma in Physics in 2003 from University of Ioannina, Greece. In 2005 he joined the Computer Engineering division of the Technical University of Delft, The Netherlands where he started his MSc study in Computer Engineering. During 2005 he has worked as a technical intern in IMEC research center in Belgium on power estimation and power optimization of the ADRES/DRESC reconfigurable processor. In 2007 he started working on his thesis project at Corporate Technology Group of Intel Corporation in Germany Microprocessor Lab. His thesis is titled "Design Partitioning for Custom Hardware Emulation". He joined INTRACOM TELECOM, a part of SITRONICS Telecom, in early 2008 where he is working on backend compilers and computer organization.