Algorithms for Embedded Memory Binding in FPGAs

ALGORITHMS FOR EMBEDDED MEMORY BINDING IN FPGAS

BY

KAVEH GHORBANI ELIZEH, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

© Copyright by Kaveh Ghorbani Elizeh, November 2008

All Rights Reserved

Master of Applied Science (2008)	McMaster University
(Electrical & Computer Engineering)	Hamilton, Ontario, Canada

TITLE:	Algorithms for Embedded Memory Binding in FPGAs
AUTHOR:	Kaveh Ghorbani Elizeh B.Sc., (Computer Engineering) AmirKabir University of Technology, Tehran, Iran
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	x, 108

To Zahra,

for her endless passion for life.

Abstract

Recent advancements in semiconductor fabrication technology have enabled field-programmable gate arrays (FPGAs) with hundreds of embedded memories. Usually, these embedded memories can be configured to work with different widths of address and data buses. In some FPGAs there is also a variety of different types of embedded memories with different capacities and configuration sets. As a consequence, it is becoming cumbersome to bind the data memory of an algorithm to these embedded memories manually. A computer-aided design tool that automates the process of binding embedded memories can save the engineering time for a design, as well as explore different alternatives to bind the data memory with the use of less embedded memories and less amount of peripheral hardware in terms of logic cells of the FPGA.

In this thesis, we first motivate the need for an algorithmic solution to the memory binding problem in FPGAs and explain the design trade-offs. Then we present an exact solution for the problem using a branching method to search the solution space exhaustively. However, due to the large solution space and the plenitude of choices in each step of the algorithm, the runtime of the algorithm is far from being acceptable for realistic problems. To manage the runtime, we have developed a fast heuristic approach. Our experimental results show that the heuristic method can achieve a suboptimal solution, which for small problem instances is shown to be close to the optimal in acceptable runtime. Moreover, when compared to manual solutions, besides substantially improving the implementation time, the heuristic can often enable a more efficient usage of the FPGA logic resources and embedded memories.

Acknowledgements

I would like to thank all the people who helped me while I was working on this research. First, I give a sincere gratitude to my supervisor, Dr. Nicola Nicolici, for all the effort, energy, and patience he put in this research. He taught me many things much more important than research, and it was a pleasure for me to work under his supervision for the past two years.

I also wish to thank my colleagues in Computer-Aided Design and Test Research Group, Adam Kinsman, Henry Ko, Ehab Anis, Jason Thong, Mark Jobes, Rumi Sahi, and Zahra Lak, who always assisted me during the coarse of my study.

My love and appreciation goes to my parents, Behrouz and Fattaneh, who where always beside me and led me to find the true values in life. My sisters Shaghayegh and Sara were always a great source of energy for me, and I am really thankful to them for my success.

I want to acknowledge all my Iranian friends in Hamilton who always cheered me up and spent time with me to ease my life in Canada. Without their help it was impossible to finish this work. Also, I am deeply indebt to Dr. Mehdi Sedighi, the source of my knowledge and passion for digital electronics.

Above all, I am thankful to God, who gave me the strength and the chance to do this work, and for placing these wonderful people in my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital electronics is playing a key role in society. The advancements in digital electronics have made it more useful and reliable over the past few decades. In fact, the transition from electromechanical machines to high performance computers with a fraction of power consumption has enabled us to use digital circuits in a vast variety of areas in modern life. Small and simple circuits in children's toys or more complex designs in cell phones, portable audio players, or laptops are all different examples of the usage of digital electronics. Their reliable operation and the ability to handle complex tasks has facilitated their application in economic sectors that require very high dependability, such as banking or aerospace engineering.

In this chapter we will start with a brief history of digital electronics in section 1.1. In section 1.2 we will describe the Moore's law and its different aspects. The section 1.3 will cover different implementation technologies for digital circuits and their pros and cons. In section 1.4 we will introduce Field Programmable Gate Arrays (FPGAs), which are the platform used for the research presented in this thesis. Finally, a short explanation of thesis organization is provided in section 1.5.

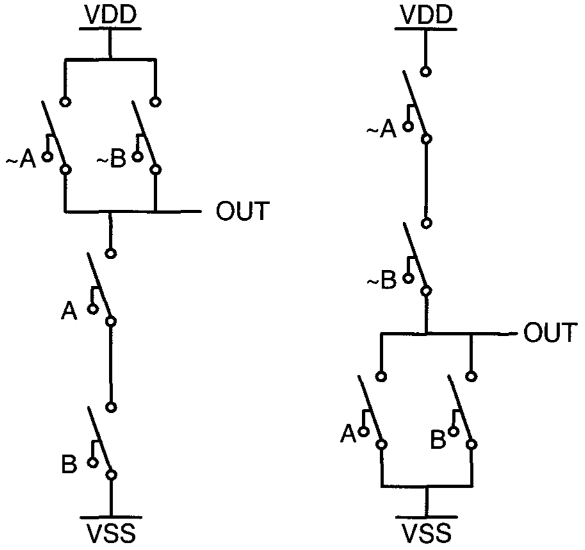


Figure 1.1: NAND and NOR gates designed with switches

1.1 Digital circuits

A digital circuit consists of logic gates and Boolean functions. All of these gates and functions can be built by switches. An example is presented in figure 1.1. Therefore, all digital circuits are based on 0-1 logic because a switch can only have 2 states: "ON" or "OFF".

The invention of the transistor was the beginning of a new era in electronics. Along with their ability to act like a current valve, transistors can be designed and configured to work like electronic switches. Their advantages over mechanical switches (relays) became evident between 1940 and 1960 [24]. Transistors were consuming significantly less power and had a much faster response time than mechanical switches and vacuum tubes. These two important facts were the main motivations for the new branch of electronics which is the "digital electronics".

The potential of digital circuits became evident soon thereafter, and advancements in digital circuit design and logic made them more useful everyday. Digital circuits started to handle more complex problems. Therefore, the number of transistors in a single circuit began to increase. The increase in the number of transistors has a direct effect on the power consumption of the circuit. To handle the power consumption of complex circuits with lots of transistors and also increase the processing speed of the circuit, engineers started to build smaller and smaller transistors. Shrinking the size of transistors has been known as the most effective method to increase the speed and reduce the power consumption of digital circuits in the physical design phase [24]. Also, it helps designers to embed more transistors in a limited area. The traditional transistors, which were components with their dimensions in the range of few inches and three wires attached to them, got phased out by microscopic transistors which were embedded on layers of silicon called "wafers", and engineers began to develop Integrated Circuits (ICs) which were the new generation of electronic circuits. The decreasing size of the transistors over the past few decades is explained by Moore's law in the next section.

1.2 Moore's law

In 1965, Intel's co-founder Gordon E. Moore made an observation about the increasing speed and complexity of digital circuits. His prediction was that the minimum feature size will continue to decrease at a rate of roughly a factor of two per year [20].

In other words, Moore claims that the number of transistors in an integrated circuit doubles every year. Figure 1.2 presents the growing number of transistors in actual digital circuits over the past few decades. This increase in the number of

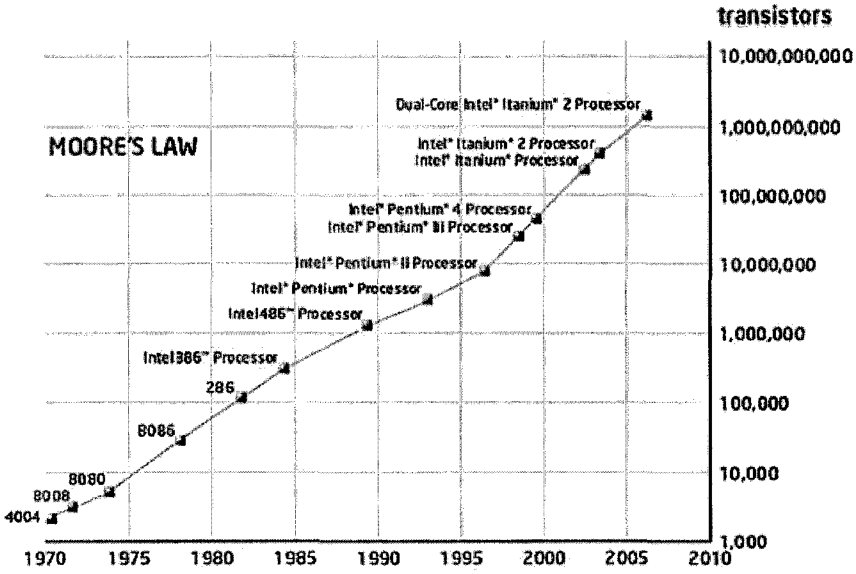


Figure 1.2: Moore's law [20]

transistors is only possible by advancements in reducing the feature size to control the power consumption and area of the chip. Figure 1.3 illustrates the relation between the Moore's law and minimum feature size over the past few decades [16].

In conclusion, we can assume that the Moore's law still holds only because of the advancements in physical design to reduce the size of transistors. The next section discusses different technologies to implement a digital circuit and their pros and cons.

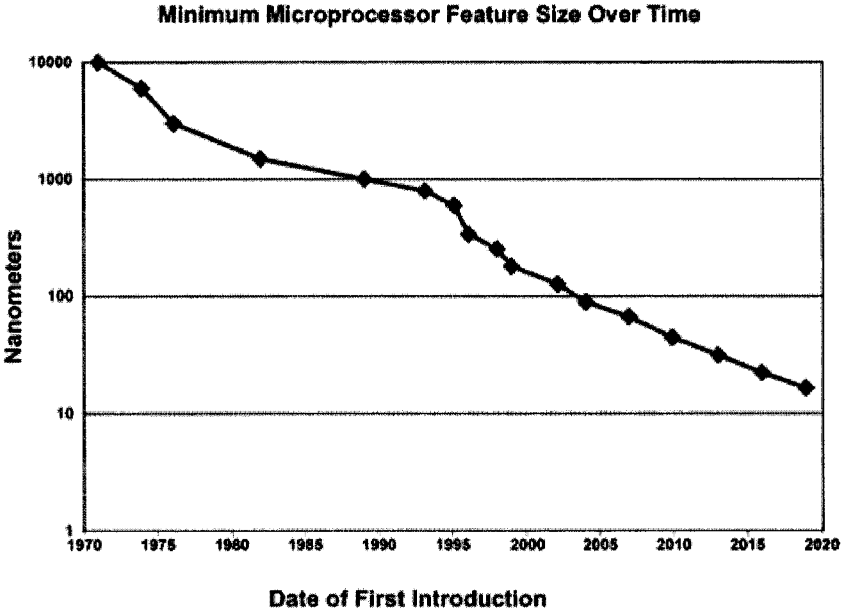


Figure 1.3: Moore's law for minimum feature size [16]

1.3 Implementation technologies for digital circuits

There are a few methods to implement a digital circuit. Each method has its own advantages and disadvantages. Therefore, a designer chooses the implementation technology or even a certain combination of them to meet the specific requirements of a product. The designer should take a number of parameters into account such as:

- Performance
- Cost
- Design time
- Production volume

Basically, we can divide the implementation technologies into four groups [21]:

- Off-the-shelf packages
- Programmable logic designs
- Standard cell based designs
- full-custom designs

We will try to give a brief explanation for each approach and its limitations.

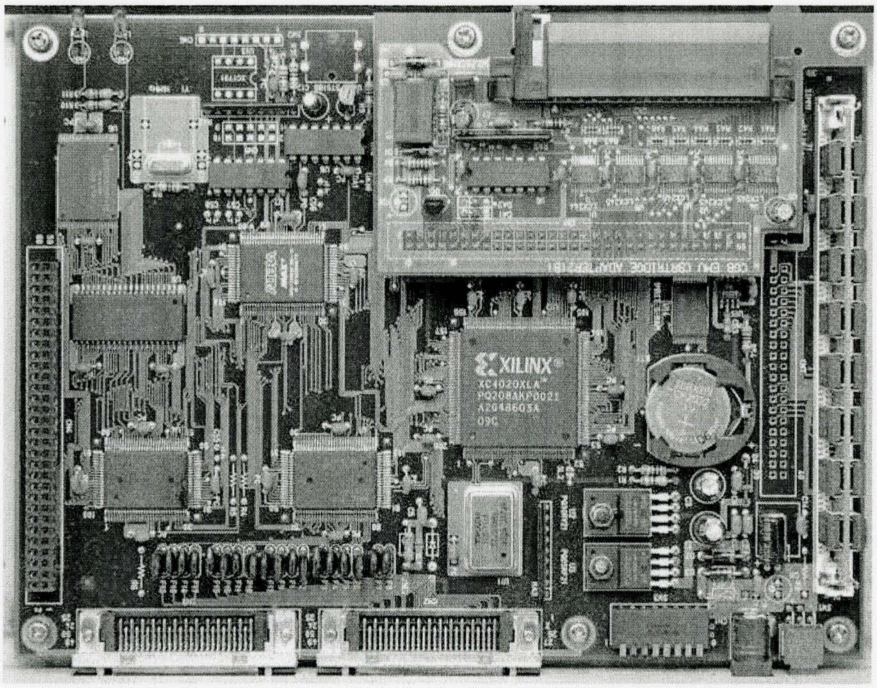


Figure 1.4: Discrete logic method[19].

1.3.1 Off-the-shelf packages

Many useful and essential logic functions are available in market as standalone ICs. One of the ways to design a logic circuit is to use these products in your design and connect them together in a proper way on a printed circuit board (PCB). In this method we are using individual logic parts in our design, so it is also called "Discrete logic". The discrete components in a design can vary from simple logic gates to more sophisticated elements like micro-controllers and programmable logic devices. Figure 1.4 shows a sample design using discrete logic approach.

Discrete logic designs can be very expensive for high product volumes because of the numerous parts which should be bought individually, also the cost for PCB and other peripheral costs increases the product cost. Furthermore, the test process for this family of designs is relatively time-consuming because of the number of devices. Also, they are not very reliable because of the connections outside of the chips. We should add the bulkiness to the disadvantages list of this family of designs.

On the other hand, off-the-shelf packages are flexible for performing partial changes or replacing faulty parts. Also, discrete logic designs can be very economical in low product volumes.

1.3.2 Programmable logic designs

This family of designs can be categorized in the semi-custom approaches. The simplest programmable logic device (PLD) is a read-only memory (ROM). We can place the truth table of a function in a ROM. The inputs of the function are the address lines and the contents of the memory are the outputs for each combination of the inputs. This approach is limited due to the exponential increase in the memory size with respect to increasing the number of inputs.

A more sophisticated approach to design programmable logic is to use programmable logic arrays (PLAs) or programmable array logics (PALs). PALs and PLAs are simple programmable ICs with prefabricated AND-OR or OR-AND logic system. Therefore, they are suitable to implement product-of-sum (PoS) or sum-of-products (SoP). We can program the interconnects from input pins to the inputs of the logic in the first level (ORs in PLAs and ANDs in PLAs). This enables us to create the products in SoP form or the sums in PoS form. Figure 1.5 shows the structure of a PLA and how a designer can program its interconnects. PLAs and PALs can also have registers embedded in them to hold the current state of the circuit and use it in the next clock

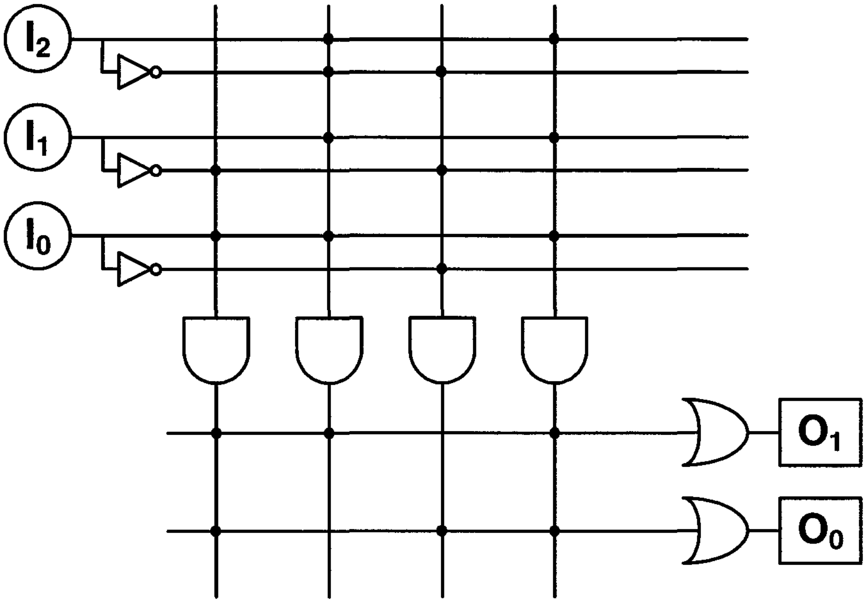


Figure 1.5: Programmable logic array (black dots represent the programmed interconnects).

cycle. Therefore, they can be used to implement state machines.

Today, the most sophisticated programmable logic ICs are field programmable gate arrays (FPGAs). FPGAs have both programmable interconnect network and programmable logic functions ability. We will discuss them more deeply in section 1.4.

1.3.3 Standard cell based designs

Similar to programmable logic designs, this approach is in the category of semi-custom designs. However, standard cell based designs are moving toward more integration. This family of designs consists of rows of standard cells. Each cell performs a fundamental logical function such as: NAND, NOR, register, inverter, and many other more sophisticated functions like single bit adders with different fan-outs. Therefore, a proper combination of different cells can create any digital circuit. These cells are designed with certain standards and parameters in order to match with each other in rows.

To implement a circuit with this approach, the designer should first synthesize the design using a certain cell library. The next step is to place the cells in the design in a proper way which minimizes the complexity of routing. Finally, a routing algorithm should be used to connect the cells in a proper way using different layers of metal.

In this method the masks of the design for different layers are created by computer aided design (CAD) tools, which is more time consuming than the previous approaches. However, predesigned cells are helpful to reduce the duration of the design process with the cost of losing a relatively small percentage of performance.

For small product volumes this method can be very expensive due to the cost of engineering tools and time, and the cost of preparing the masks. However, the increasing the volume of the product will vanish these costs. The performance of this method is higher than the previous ones because of the level of integration. Figure 1.6 shows a design with its three rows of standard cells, and the connections among cells with different metal layers.

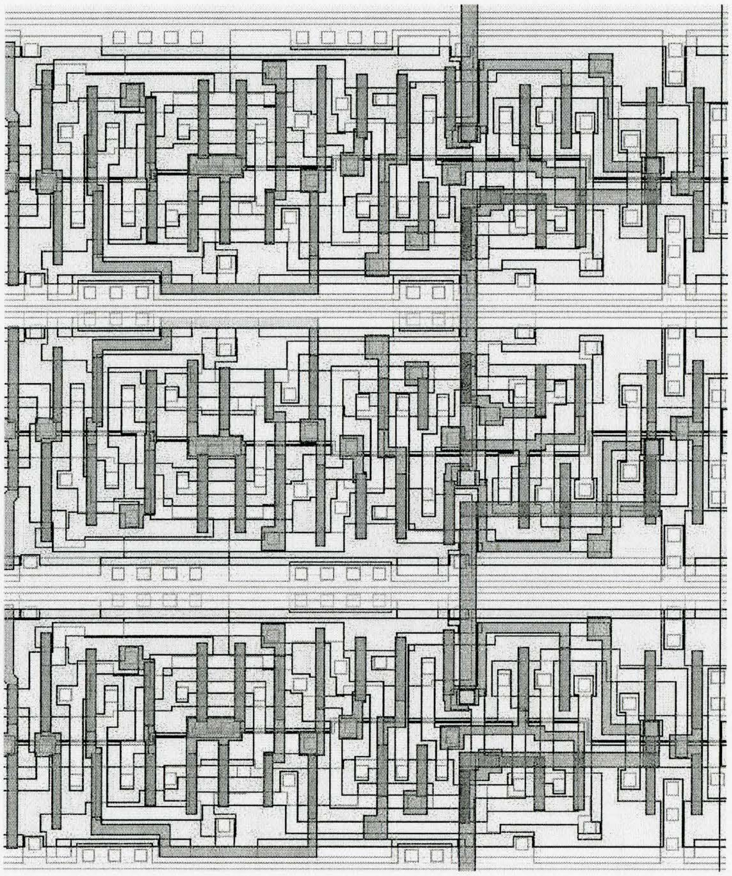


Figure 1.6: Connections between three rows of standard cells[2].

1.3.4 Full-custom designs

This method is the most expensive and time consuming method to design a digital circuit. However, the result has the smallest size and highest performance. Also, this method is the most economical method for high product volumes because the reduction in size of each chip offsets the high cost of mask development.

Unlike the standard cell based method, this method does not use predesigned cells, and the designer should design the physical layers of the design from scratch. Therefore, the engineering cost of this method is higher than the previous one because the designer has to deal with the masks of all layers. This extra effort pays back when it comes to the area of the chip and its performance. By designing all physical layers from scratch, the designer is now able to use all possible methods to enhance the performance of the product. Moreover, the product of this method is the most reliable among all of the digital circuits' development technologies.

We discussed the four main methods to develop digital circuits and their advantages and limitations. Figure 1.7 gives a rough idea about the relation between the cost and product volume for all technologies. Figure 1.8 presents a diagram to compare chip area versus design time for all technologies. A summary of what we discussed in this section so far is available in table 1.1.

It is obvious that in many cases an intelligent design can use a number of these technologies together to take advantage of each one in the best way. Today, many advanced processors have a programmable part which can be configured to do certain tasks much faster. Also, many designs are a combination of both standard cell and full-custom approaches together, and the reason is that in some parts of the design the performance is critical unlike other parts. Therefore, the designers use

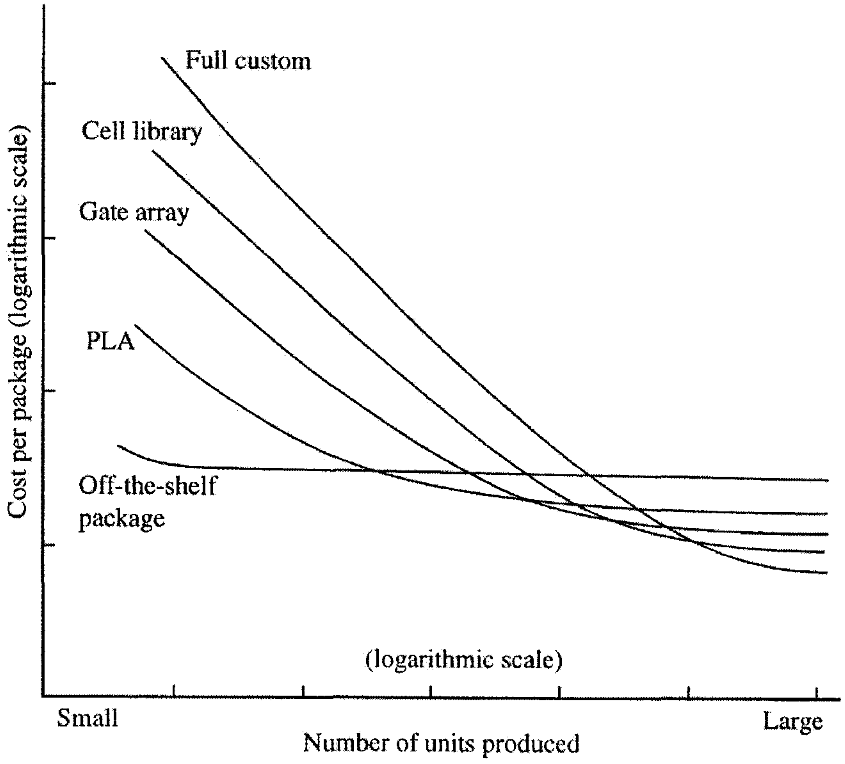


Figure 1.7: Package cost vs. product volume [21]

Table 1.1: Comparison of different task-realization approaches.

Feature	Full-custom	Standard-cell based	Programmable logic	Off-the-shelf package
Performance	Fastest	Fast	Medium	slow
Unit cost				
High volume	Lowest	Low	Medium	Highest
Low volume	Highest	High	Medium	Lowest
Development time	Longest	Long	Medium	Short

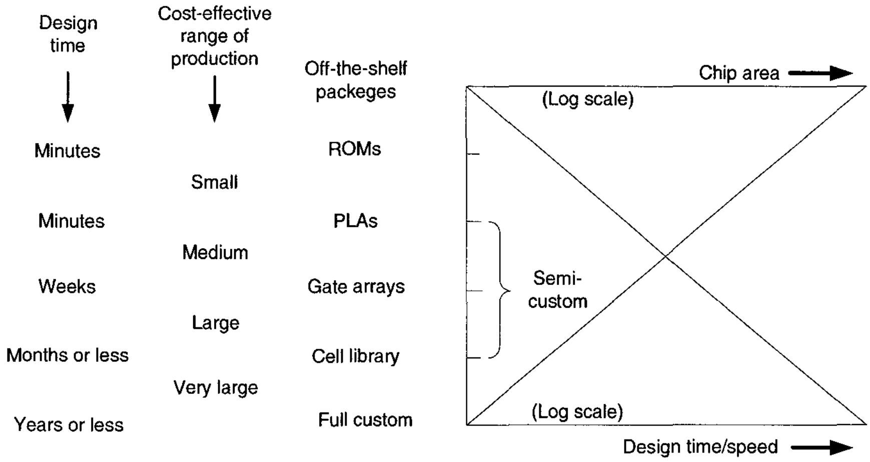


Figure 1.8: Chip area vs. design time [21]

full-custom technology for those parts, and develop the other parts with cell-based methods because this method is much faster while giving a reasonable performance. Another good example for using a combination of technologies in one design is a modern FPGA. These FPGAs have custom designed multipliers and basic signal processing units in them; the reason is that these units are used frequently in digital designs in FPGAs. Therefore, FPGAs try to provide some of these units to enhance the performance of the designs running on them.

1.4 FPGAs

As discussed earlier the advantage of FPGAs over other programmable logics is that they have both programmable logic blocks and programmable interconnects. FPGAs consist of a large set of basic blocks with the ability to be programmed. Each blocks usually has a four to six input look-up table (LUT) along with one or two registers

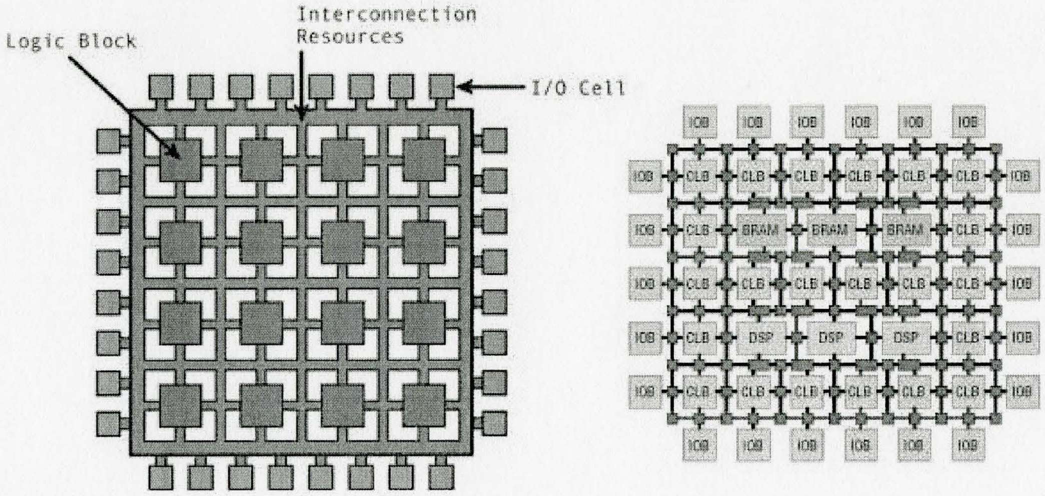


Figure 1.9: FPGA structure[5].

with some peripheral logic. Figure 1.9 shows a FPGA and the position of its logic blocks and interconnects.

Based on the complexity of the logic blocks in a FPGA we can categorize FPGAs into two categories:

- Fine grain FPGAs
- Coarse grain FPGAs

The disadvantage of coarse grain FPGAs is that usually designers can not use the entire embedded logic in each block, so the amount of unused logic increases when we use this kinds of FPGAs. On the other hand, the complexity of the logic blocks helps the designers to use less number of blocks for a certain circuit compared to a fine grain FPGA. It means that coarse grain FPGAs use less interconnect resources

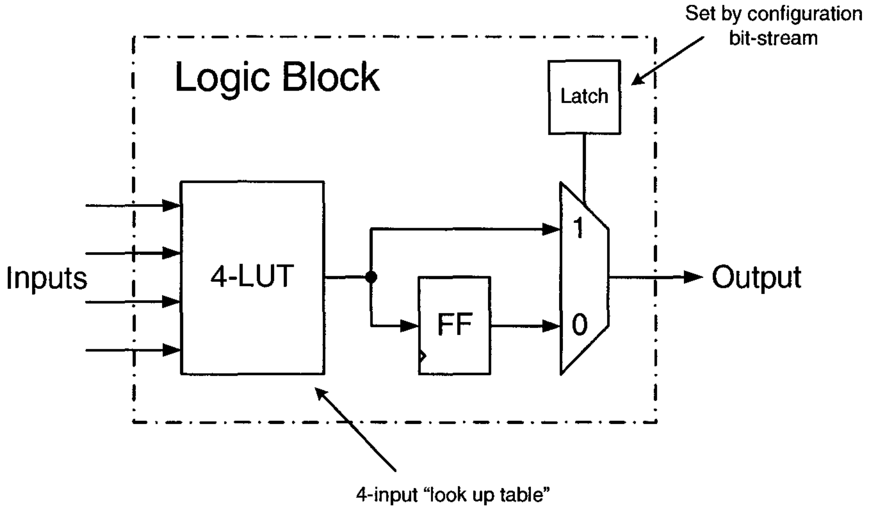


Figure 1.10: 4-input logic block

because of the enhanced ability of each block. Figure 1.10 shows the schematic of a sample 4-input logic block. For more detailed schematics please refer to [1, 29].

The logic blocks in fine grain FPGAs are designed in such a way that they are simple enough that a synthesis tool can use most of the embedded logic in each block. However, the simplicity of the blocks makes the synthesis tool use more of them for a certain task. Therefore, we will have a pressure on the routing resources to connect all the used cells together in a proper way.

We can also categorize the FPGAs based on the technology used to program the interconnects:

- Fuse based FPGAs
- Memory based FPGAs

The advantage of memory based FPGAs is that they can be programmed several times. However, due to the use of memories they are a bit slower than fuse based ones. On the other hand, the disadvantage of the fuse based FPGAs is that we only can program them one time.

Most of modern FPGAs are coarse grain and memory-based. In this case, they can be programmed several times, and having more complex logic blocks reduces the amount of required interconnects which increases their performance. Today, FPGAs have more components embedded in them such as processing units and memories.

Many digital applications use a basic set of arithmetic functions such as multiplication. Many of today's FPGAs have a set of embedded arithmetic functions such as multipliers to help the designers. These multipliers have a limited number of bits. For example in the Stratix II FPGA (manufactured by Altera), each multiplier has 9-bit inputs. However, these multipliers can be chained together to create multipliers with a higher number of bits.

Embedded memories are important components of FPGAs. Almost all FPGAs have embedded memories in them. The architecture of the FPGA is designed in such a way that it has thousands of small embedded memory chunks all over it between logic blocks and interconnects. These memory parts are usually identical in their capacity and dimensions. However, some FPGAs, such as Altera's Statix family, have more than one type. Each type has a different capacity and available bit widths for its address and data. For example, the Stratix III has three types of memories embedded in it.

Memory parts can have several configurations (address and data bus bit width), and in each configuration they can have different properties such as different number

Table 1.2: Memory elements in EP3SL340 FPGA from Stratix III family [1].

Feature	MLABs	M9K Blocks	M144K Blocks
Maximum performance	600 MHz	600 MHz	600 MHz
Quantity	6750	1040	48
Configurations (depth*width)	64*8 64*9 64*10 32*16 32*18 32*20	8K*1 4K*2 2K*4 1K*8 1K*9 512*16 512*18 256*32 256*36	16K*8 16K*9 8K*16 8K*18 4K*32 4K*36 2K*64 2K*72

of ports. Therefore, we can configure each memory block differently. All these types and configurations are there to make FPGAs more flexible and more usable for a large spectrum of applications. Table 1.2 shows the types and configurations for embedded memories of EP3SL340 FPGA from the Stratix III family by Altera.

1.5 Thesis organization

This thesis presents a solution for an emerging problem in CAD tools for FPGAs, which is the memory management problem. Chapter 2 is provides a literature review of similar problems tackled in the past. Also, in this chapter we will formulate the problem we are dealing with in this thesis. Chapter 3 discusses the problem in depth, and gives a solution to it in some special and common cases. Also, in this chapter we discuss an acceptable heuristic method to solve the problem.

Chapter 4 contains a case study along with some experimental and analytical results of the proposed method. Finally, chapter 5 concludes what we discussed in the thesis and covers the future work and open problems.

Chapter 2

Literature Review

In the previous chapter we briefly discussed the digital electronics and the rate of the advancement in this field captured by Moore's law. We have also introduced different implementation technologies and their advantages and disadvantages, which make each of them suitable for a certain family of designs with their specific parameters and requirements.

In this chapter we will address the problem which we investigate in this thesis. To better understand our problem we will introduce the general binding problem in computer aided design (CAD), and its role in behavioral and logic synthesis in section 2.1. Then to create a link between the resource binding and data path of a design, we will discuss data path and memory organization in digital designs in section 2.2.

The connection between the binding problem and data path organization of a design is in physical memory binding. Physical memory binding, which is a step in data memory organization, will be discussed in section 2.3. In section 2.4 we will discuss the reasons why we cannot use the traditional binding approaches to attack the problem of physical binding and why this was not an issue in the past. Finally, we will formulate the problem, and define the important metrics in section 2.5.

2.1 Binding problem in CAD

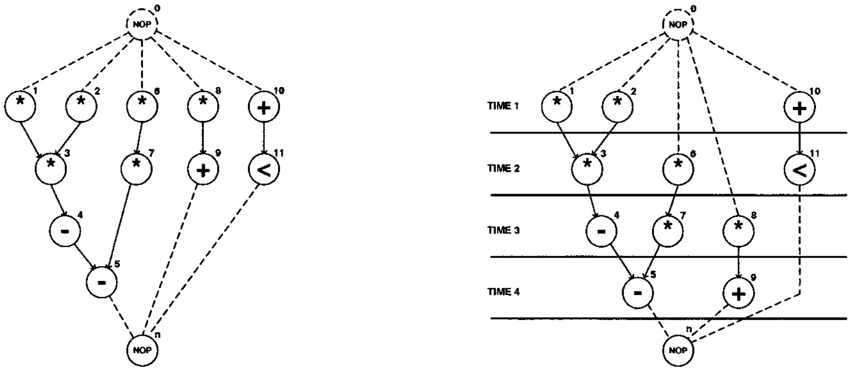
The design flow of a digital circuit consists of different layers of abstraction from behavioral descriptions to physical descriptions. The binding problem can be discussed in two different layers of abstraction: architectural-level and logic-level. The binding problem in the architectural-level, which is a higher-level description, is discussed first. Later, we will focus on the logic-level binding.

2.1.1 Binding in architectural synthesis

Architectural synthesis is the process of constructing a gate-level description of the design from a behavioral model, such as data-flow or sequencing graph. Architectural synthesis consists of two main steps: scheduling and binding.

Scheduling is responsible for the timing of each operation in the design. In other words, we define exactly when to start each operation in terms of clock cycles. There are various methods for scheduling, from basic methods such as “as soon as possible” (ASAP) and “as late as possible” (ALAP) to more sophisticated approaches with different purposes such as minimizing the runtime or minimizing a certain type of a required physical resource. Figure 2.1(a) illustrates the flow of a simple design which consists of 6 multiplications and 5 ALU operations. This design is scheduled with a sample scheduling algorithm in figure 2.1(b).

Resource binding is the process of assigning operations of the design to the physical resources. In this mapping we move from an abstract functional description toward



(a) Before scheduling.

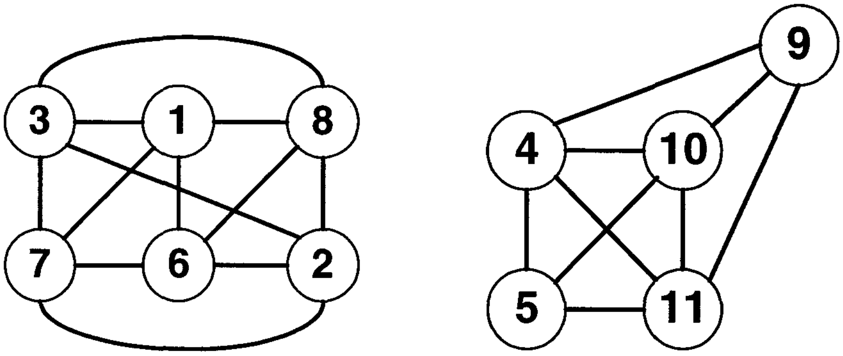
(b) After scheduling.

Figure 2.1: A sample design before and after scheduling[8].

a more feasible physical description. In many cases resource binding is accompanied with resource sharing, which is a crucial task in hardware design. Resource sharing is the process of sharing a physical resource between two or more logical operations which are not concurrent.

Resource sharing has an important role in managing the physical size of a design. By sharing the same resource between multiple non-concurrent operations in the design we can reduce the number of physical resources which results in the reduction of the design area in many cases. The area of a design has a direct effect on its working frequency and power consumption.

Assuming that we have two types of physical resources (multipliers and ALUs), we can create the compatibility graphs of each resource type. The compatibility graphs for this example are presented in figure 2.2. The compatibility graph presents the information about the operations we can assign to the same physical resource because each edge in the graph represents a possible sharing of a physical resource between



(a) Multiplier

(b) ALU

Figure 2.2: Compatibility graphs for resource binding[8].

to operations because they are not concurrent.

There exist different resource binding algorithms with different concerns. Figure 2.3 shows a sample binding of our example based on the compatibility graphs of figure 2.2. Apparently, in this example the binding is done with two multipliers and two ALUs. It is apparent that we could do the binding with only one resource of each type. However, it would make the design slower in terms of latency because we had to stall some operations until the required resource is available.

Besides binding logical operations, register binding is an important part of each design which influences the overall architecture and performance of the design. There are many approaches for register binding, each of them working on different parameters of the design. A helpful tutorial to all of the steps discussed above can be found in [8].

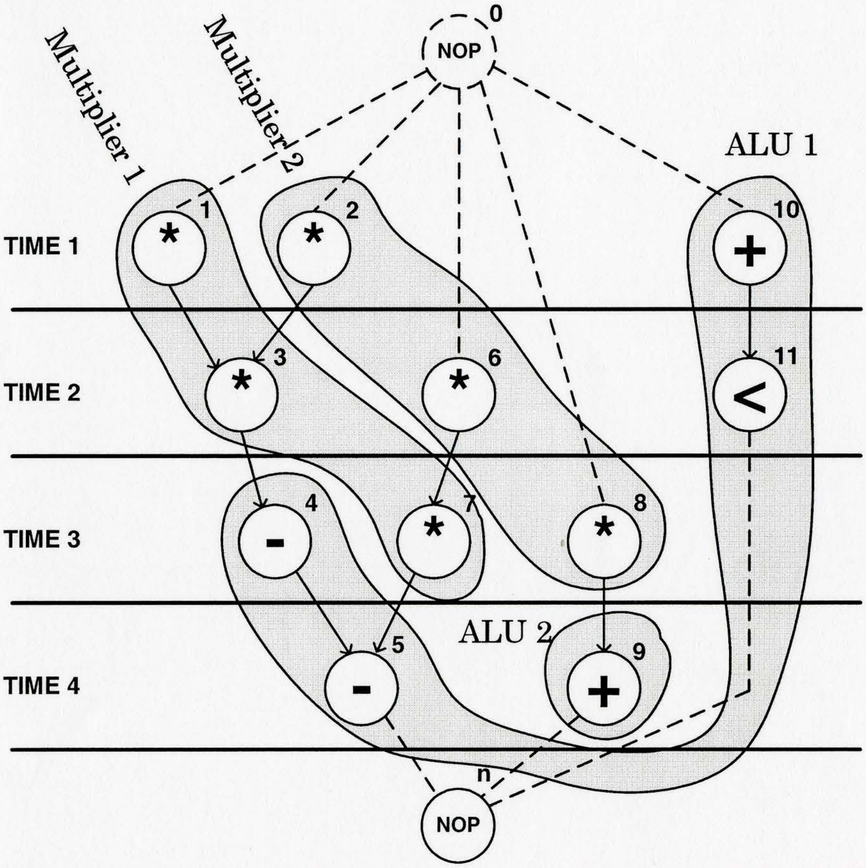
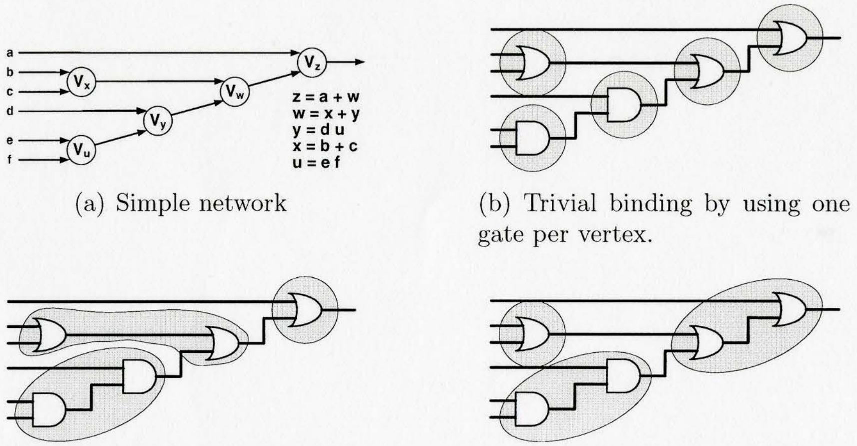


Figure 2.3: A sample binding of our example[8].



(a) Simple network

(b) Trivial binding by using one gate per vertex.

(c) Network cover with library cells including two-input and three-input AND and OR gates.

(d) Alternative network cover

Figure 2.4: Different binding approaches[8].

2.1.2 Binding in logic synthesis

The logical-level problem of binding is library binding, which is often referred to as “technology mapping”. The difference between the architectural-level binding and library binding is that in architectural-level binding we assign each logical operation to a gate. However, in library binding most of the physical resources in the library are more complex than a single logical gate. Moreover, the elements of the library have some other parameters beside their structure which are important in the design, such as area and delay. Figure 2.4 shows the difference between the basic binding problem and the practical library binding.

Library binding is important in standard-cell and FPGA based circuit design because it provides an interface to the physical implementation. Moreover, library binding provides the means for retargeting logic design to different implementation technologies.

A library contains a set of physical resources in different forms with different parameters. Therefore, the binding algorithm should use these parameters to do the binding process with respect to some objectives, such as area, delay, or testability of the design.

The practical approaches to library binding can be classified into two major groups [8]:

- Heuristic algorithms
- Rule based approaches

The heuristic approach to solve the library binding problem is based on its likeliness to code generation in compilers. In both cases we deal with a matching problem, and the choice of optimal matches. The proposed methods to solve the matching problems can be categorized into two major groups:

- Boolean approaches
- Structural approaches

Both of these approaches have been used in the algorithmic library binding. The Boolean method is explained in depth in [3, 30]. For more information about the structural methods please refer to [8].

The rule based approaches for binding are usually used as a complement to algorithmic binding. The basic idea of this method is to refine the circuit step by step.

In other words, these refinements are some local changes to the circuit which do not change the circuit behavior, and just replace an equivalent sub-circuit. Moreover, these rules can have priorities over each other [7, 4, 6].

2.2 Data memory organization

So far we have only discussed the logical functions in a design, which define the control path. In the following section we will focus on the complementary part of digital circuits, which is the data path. The flow of the data in the design is through different functional units, registers and memory units. Therefore, the arrangement of these units and their contents is an important part of the design.

There are several methods for memory architecture, and each method has its own properties and usage. There is a lot of research done in this field, and a proper summary of all different approaches for memory organization can be found in [23, 15, 13].

All these methods focus on different ways to access data from physical memories with minimum cost. They try to manage the storage of the data based on the way the design accesses them with a high level vision. The bottom line of all these approaches is to keep the data which is more frequently accessed within the algorithm closer to the processor and in the memory units with lower access cost. Figure 2.5 shows the different memories in the addressable memory space of a Central Processing Unit (CPU) with different access-times.

In other words, the focus of this area of research is to decide which parts of data to keep in on-chip memories, and how to cache data to reduce cache misses. The proposed approaches in this field perform a study on the algorithm behavior in its

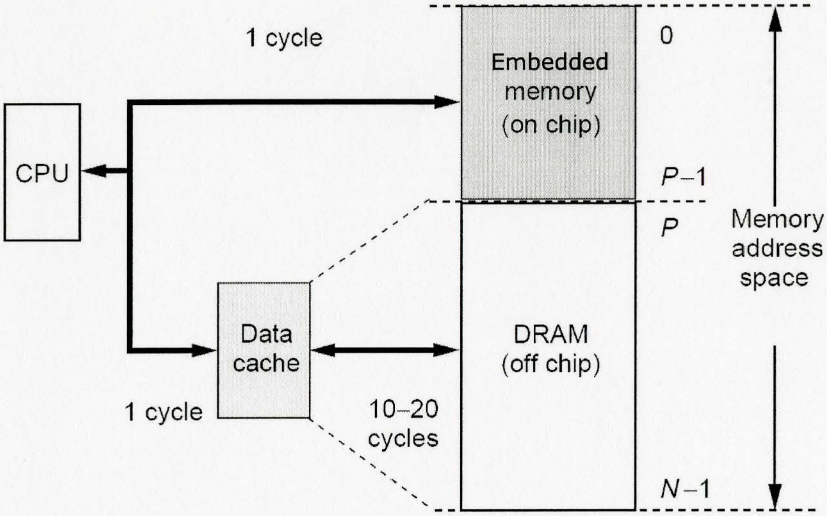


Figure 2.5: Memory levels of a sample design[23].

data accesses, and try to enhance the performance of the data access process [14]. Therefore, they do not work on the details of the physical implementation of the memory architecture, which we refer to as physical memory binding.

2.3 Physical memory binding

Similar to logical resource binding, physical memory binding is crucial in migrating between different implementation technologies because each technology has its own specific parameters in logical and memory units. We can divide the research on memory binding into two main branches:

- Memory binding in ASICs
- Memory binding in FPGAs

In the case of ASICs, the problem is to select suitable memory components from a memory library, and performe an acceptable placement, and design the interfaces to access the memories. In this case there is some research done on how to store data vectors in physical memories, guarantee the access to each data element within an acceptable time, and with an acceptable cost [12, 25, 26, 28].

In the case of FPGAs, the problem is different because the hardware is fixed, and the designer can only decide on a suitable configuration within the given choices as well as the interconnects and peripheral logic around each memory unit. The research in this field is limited to an approach to bind a single data vector to multiple types of physical memories which are not reconfigurable [11]. There is also an Integer Linear Programming (ILP) formulation presented in [22] to give an exact solution to the problem with some simplifying assumptions which affect the real problem radically. The assumptions in this formulation are:

- Different ports of a single physical memory can have completely different configurations.
- All the physical memories used in covering a single data structure are from the same type.

Moreover, this method does not account for the peripheral logical resources nor the minimization of the amount of physical storage we need to use. It only works based on the latency and access-time cost. On the other hand, this method tries to expand itself to the external memory banks and some of the parameters which they imply to the design, which makes it unique in this case.

There are also some industrial efforts to solve the problem of physical memory binding in FPGAs [1]. However, these tools are not powerful enough to handle the complexity of today real-world problems because:

- They work on only one data vector.
- They only use one type of physical memories in each solution.
- They only use one configuration of the used physical memory.
- In many cases the user should provide the suitable configuration to the tool.

Having these limitations, the industrial tools are not able to work on the peripheral logic and leftover minimization. Optimization aside, in some cases these tools are not even able to generate a solution for the given problem.

2.4 Motivation for our work

The main motivation for this research is that the physical memory binding in FPGAs is a new problem in CAD tools for FPGAs, and so far this task was mainly done by engineers. However, the following two reasons are increasing the complexity of this task beyond the ability of an engineer during an acceptable time:

a) The advancements in the fabrication technology have enabled FPGAs with hundreds and even thousands of physical (also called embedded) memories with different types and configurations [1, 29]. However, early FPGAs had very limited embedded memory resources or even none, so we did not need a tool to

Table 2.1: Memory elements in EP2S180 FPGA from Stratix II family [1].

Feature	M512 RAM Block	M4K RAM Block	M-RAM Block
Maximum performance	500 MHz	550 MHz	420 MHz
Quantity	930	768	9
Configurations (depth*width)	512*1 256*2 128*4 64*8 64*9 32*16 32*18	4K*1 2K*2 1K*4 512*8 512*9 256*16 256*18 128*32 128*36	64K*8 64K*9 32K*16 32K*18 16K*32 16K*36 8K*64 8K*72 4K*128 4K*144

map the data vectors to those resources. Table 2.1 shows the types and configurations for embedded memories of EP2S180 FPGA from Stratix II family designed and manufactured by Altera company.

b) The advancements in FPGAs have made them very powerful. Therefore, they are able to process a larger amount of data. The huge parallelism that FPGAs provide today has increased the data volume which we have to feed to them. Consequently, the number and volume of processing data vectors have increased recently. Another important reason for this increase is the migration of different algorithms from floating-point to custom floating-point or fixed-point algorithms. This migration is very critical to fully utilize the provided parallelism by FPGAs because it reduces the size of processing units, so we can fit more of them in a single FPGA chip.

We cannot use the traditional binding algorithms to solve the physical memory binding problem because these two types of binding have some key differences between them. Two examples for these differences are:

- In the physical memory binding problem we have the challenge of reducing the peripheral logic required to guarantee the proper access to all data elements. This problem does not exist in logical resource binding.
- In physical memory binding we want to create a solution with the minimum wasted storage in the physical memories. We have the same problem in library binding in FPGAs because we wish to use all the logic prepared in each cell to reduce the overall number of used cells. However the methods we need for these two minimizations are different. This variation comes from the differences in the way we map data vectors and operations to the physical resources.

In the next section we define the problem which we investigate in this thesis.

2.5 Problem formulation

This thesis addresses the problem of physical memory binding in FPGAs. The physical memories in FPGAs are also commonly referred to as embedded memories. Assuming that we have multiple types of physical memories, and from each type we have a limited number, and each type has a set of different configurations, we want to store in them a set of data vectors called virtual memories.

It is important to stress that efficient use of embedded memories in FPGAs is very important in the overall performance of the design because it enables us to store a larger amount of data close to the processing units. Therefore, it minimizes the access to the external memories which is usually slower compared to on-chip memories. Moreover, the bandwidth of the on-chip memory units is not comparable to the external memories because we have to deal with the limitations of the number of I/O pins of the FPGAs. However, we can access a large amount of data from several embedded on-chip memories in the FPGA without these I/O limitations.

We have defined two cost functions in the process of embedded memory binding in FPGAs:

- Amount of required peripheral logic in terms of logic cells in the FPGA (multiplexer cost)
- Amount of wasted embedded memory bits (leftover)

The peripheral logic size in our case can be translated to the multiplexer cost of the design. In other words, when we divide a virtual memory which is larger than a single physical memory into several physical memories, in some cases we have to choose the physical memory we want to work with based on the provided address, so we will need a multiplexer. Figure 2.6 presents the multiplexers in a sample mapping. Obviously, we need eight 2:1 multiplexers to create this mapping.

An embedded memory binding which is wasteful in logical resources can affect the entire design in an FPGA because it is possible that we have to reduce the size or the number of the processing units in order to save some logic to do the memory binding. However, by performing the binding process with less logical resources, we will be able to use that logic in the processing part of the design to enhance parallelism and increase the processing speed or resolution. Furthermore, when the binding is done

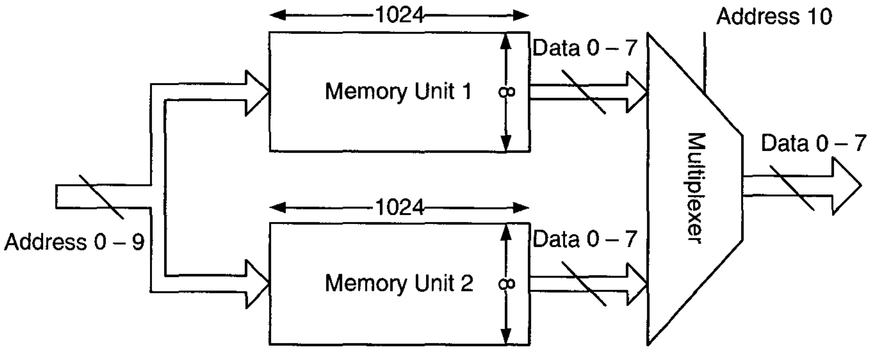


Figure 2.6: Generated peripheral logic in memory mapping

with an extensive use of logic cells in the FPGA, this added logic will increase the timing between the memory and the processing units because it is placed between the memory and logic part of the design.

The leftover is the amount of unused bits of physical memories which we have used to cover the virtual memories. We can define two types of leftover: horizontal leftover and vertical leftover. This categorization will help us later in the design of the algorithms. Figure 2.7 shows the two types of leftovers in a sample coverage.

In this project we have only worked with on chip physical memories because it simplifies the timing properties. The access times of all on chip physical memories in today FPGAs are close to each other, so we can assume that it takes the same number of clock cycles. Moreover, due to limited amount of embedded memory resources, we cannot guarantee the minimization of the cost functions by working on virtual memories individually. Consequently, we should perform the mapping problem for all virtual memories in the problem concurrently.

In order to simplify the reading of this thesis, we should explain some of the terms which we will use in the following chapters: VM stands for "virtual memory",

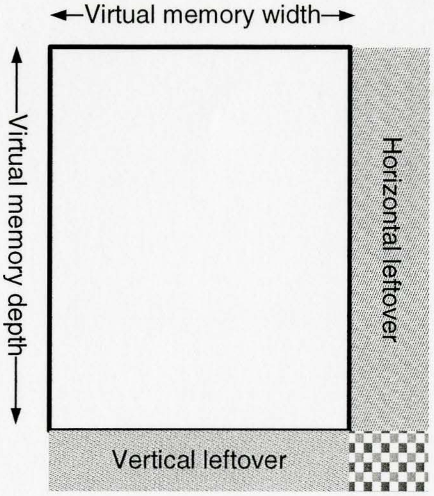


Figure 2.7: Vertical and horizontal leftover

and PM refers to "physical memory". Also, when we discuss the configurations of a physical memory, we refer to the number of their address entries as their *height*, and the number of data bits in each address location is denoted as their *width*. Therefore, we can define the concept of a *short* (wide) or *tall* (thin) configuration based on its height compared to other possible configurations.

Chapter 3

Algorithms for Binding Virtual Memories to Physical Memories

In the previous chapter we discussed the general binding problem in CAD, and more specifically the physical memory binding problem. We also discussed some reasons why the memory mapping process in FPGAs was not a major concern in the past, and why it is getting more complicated for existing and future designs. Furthermore, we summarized the differences between the memory mapping problem in FPGAs and other binding problems in CAD.

In this chapter we present a solution for the physical memory binding in FPGAs through some intermediate steps which are simplified subproblems. These subproblems will help us understand the role of different parameters involved in the complexity of the problem and its solution space. However, the ultimate goal is to solve the problem in the case of multiple virtual memories and multiple types of reconfigurable physical memories.

Because the problem we are tackling (physical memory binding for FPGAs) involves fitting small physical memories (rectangles) into large virtual memories (bins), one may argue that it is similar to the well-studied bin-packing or knapsack problems. We first outline the differences between our problem the above known-problems. The bin packing problem involves fitting a number of rectangular objects in as few bins as possible [10]. The rectangular objects have different sizes, and the dimensions of the bins are defined. We can also relate the problem to the 2-dimensional knapsack problem, which is putting a number of different-sized rectangular objects with different values in a 2-dimensional knapsack with known dimensions to maximize the overall value in the knapsack [18].

However, our problem is different because *we allow the creation of leftover* which is not defined in any of the prior problems. Moreover, the rectangular objects which we deal with are not random-sized because the depth (and in many cases the width) of the physical memories *are powers of 2 and multiples of each other.* The fact that physical memories are in most cases *reconfigurable* (i.e., although the memory capacity stays the same, the aspect ratio changes from one configuration to another) adds an additional dimension to our problem. In addition, the concept of *multiplexer cost* cannot be introduced in any of the prior problems.

Although both problems of bin-packing and 2-dimensional knapsack are known as NP-complete problems, we cannot assume that our problem is also NP-complete. Nonetheless, our experience with it indicates that the problem is intractable. Therefore, due to the lack of a formal proof, we conjecture that the problem is NP-complete. For the rest of this thesis we focus on understanding its solution space and designing effective algorithms that can be adopted in the industrial practice.

Based on the problem definition in the previous chapter, we can list the parameters with a significant role in the complexity of the problem as follows:

- Number of virtual memories,
- Number of different types of physical memories,
- Number of physical memories in each type,
- Number of different configurations for each type of physical memories,

The factors mentioned above help us to notice and understand the subproblems which can help us to design a solution for the problem in its general format. Each subproblem is created with elimination of at least one of the complexity factors of the general problem. In the next three sections of this chapter we will try to discuss the following three subproblems:

- Single virtual memory & one type of reconfigurable physical memory,
- Multiple virtual memories & one type of reconfigurable physical memory,
- Single virtual memory & multiple types of reconfigurable physical memories,

After discussing the above subproblems we will try to attack the problem in its general format, which is "multiple virtual memories & multiple types of reconfigurable physical memories". This is discussed in the fourth section of this chapter. Finally in section 3.5 we will describe a heuristic method to attack the problem in its general format.

3.1 Single VM - Single PM Type

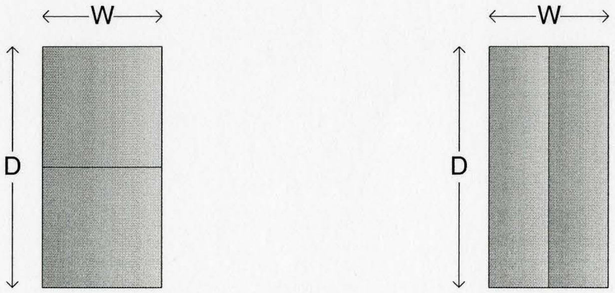
In this problem we intend to map one virtual memory to a number of identical physical memories. Each used physical memory has a configuration chosen from a variety of predefined configurations. The only complexity factor in this case is the decision about the configuration of the used physical memories. It is needless to mention that the capacity of all configurations are equal, and the difference lies in the differences in their dimensions, so as the height of configurations grows their width starts to shrink.

Observation 1: *Using tall configurations reduces the multiplexer cost.*

By using tall configurations we will be able to cover the depth of the virtual memory with a smaller number of physical memories on top of each other. Therefore, we will have a smaller multiplexer cost. Figure 3.1 shows a sample area which we can cover with two different configurations. It can be seen that the first case creates a 2:1 multiplexer which easily can be eliminated by using the taller configuration in second case. However, this method does not always guarantee the minimum leftover.

To attack the problem in this case (one virtual memory and one type of physical memories) we can divide the virtual memory into two areas and work on them separately. This division is done based on the depth of the tallest configuration of the used physical memory.

The first area, which is at the top of the virtual memory, is an area which we can cover by using the physical memories in their tallest possible configuration without creating any vertical leftover. Therefore, we can simply cover all of this area by putting the physical memories in their tallest configuration next to and at the top of each other. This method also creates the minimum amount of horizontal leftover



(a) Formation of 2 to 1 multiplexers due to the chosen configurations.

(b) Elimination of the multiplexers for the same area by changing the configuration of the used physical memories.

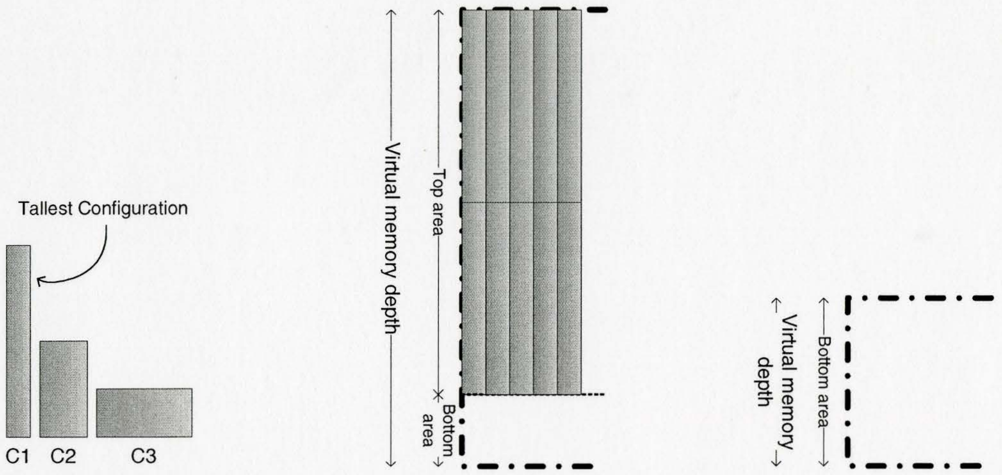
Figure 3.1: Elimination of unnecessary multiplexers.

for this area because tall configurations have smaller widths compared to short configurations. Therefore, the leftover area which is created by the unused width of the physical memories in the last column is minimized in this case.

Observation 2: *Using tall configurations reduces the horizontal leftover.*

This observation is always true assuming that the widths of the physical memory in its different configurations are multiple of each other. This assumption is valid in all the regular configurations of the physical memories of today FPGAs. Regular configurations are the configurations which do not use the allocated parity bits as data bits to increase the capacity of the physical memory, which may affect the reliability of the design in environments that are prone to soft errors.

The second area is the remaining part at the bottom of the physical memory. This area cannot be filled with the tallest configuration without creating any vertical leftover. Figure 3.2 shows an example of different configurations of a sample physical



(a) Configuration set for the physical memory.

(b) Top area consists of two rows of physical memories in their tallest configuration.

(c) Absence of top area because the depth of the virtual memory is already smaller than the tallest configuration of the physical memory.

Figure 3.2: Division of the virtual memory into two parts.

memory and the division of the virtual memory into two areas based on the given configurations.

In conclusion, by dividing the virtual memory into two areas based on the depth of the tallest configuration of the physical memory we can cover the top part with minimum leftover and multiplexer cost. This is simply done by using the physical memories in their tallest configuration to cover the top area. However, the binding process for the bottom area depends on the objective of the algorithm. Hence, this area can be filled with two different strategies:

- Minimizing the amount of leftover.
- Minimizing the multiplexer cost.

In the following subsections of this section we will discuss the methods for binding the bottom part of the virtual memory for minimization of leftover and multiplexer cost respectively.

3.1.1 Binding for Leftover Minimization

As introduced in section 2.5, there are two types of leftovers in a physical memory binding problem: vertical and horizontal. To minimize the overall leftover we have to minimize the sum of these two amounts. In this section we will start with minimizing the vertical leftover in a binding problem, and then we will relate it to the horizontal leftover minimization.

Vertical leftover is the result of using configurations of the physical memory which are taller than the uncovered area at the bottom of the virtual memory. The larger the difference between the depth of the configuration and the depth of the uncovered area, the more leftover will be created. Therefore, we can reduce the vertical leftover by using the configurations which are shorter than the depth of the bottom part.

In other words, it is likely to reduce the vertical leftover by replacing a tall configuration used in the bottom area with a combination of the configurations which are shorter than the depth of the bottom area. It is obvious that this optimization in vertical leftover comes with an increment in the multiplexer cost because we will have more physical memories on top of each other.

Observation 3: *The depths of different configurations of a physical memory are a multiple of each other.*

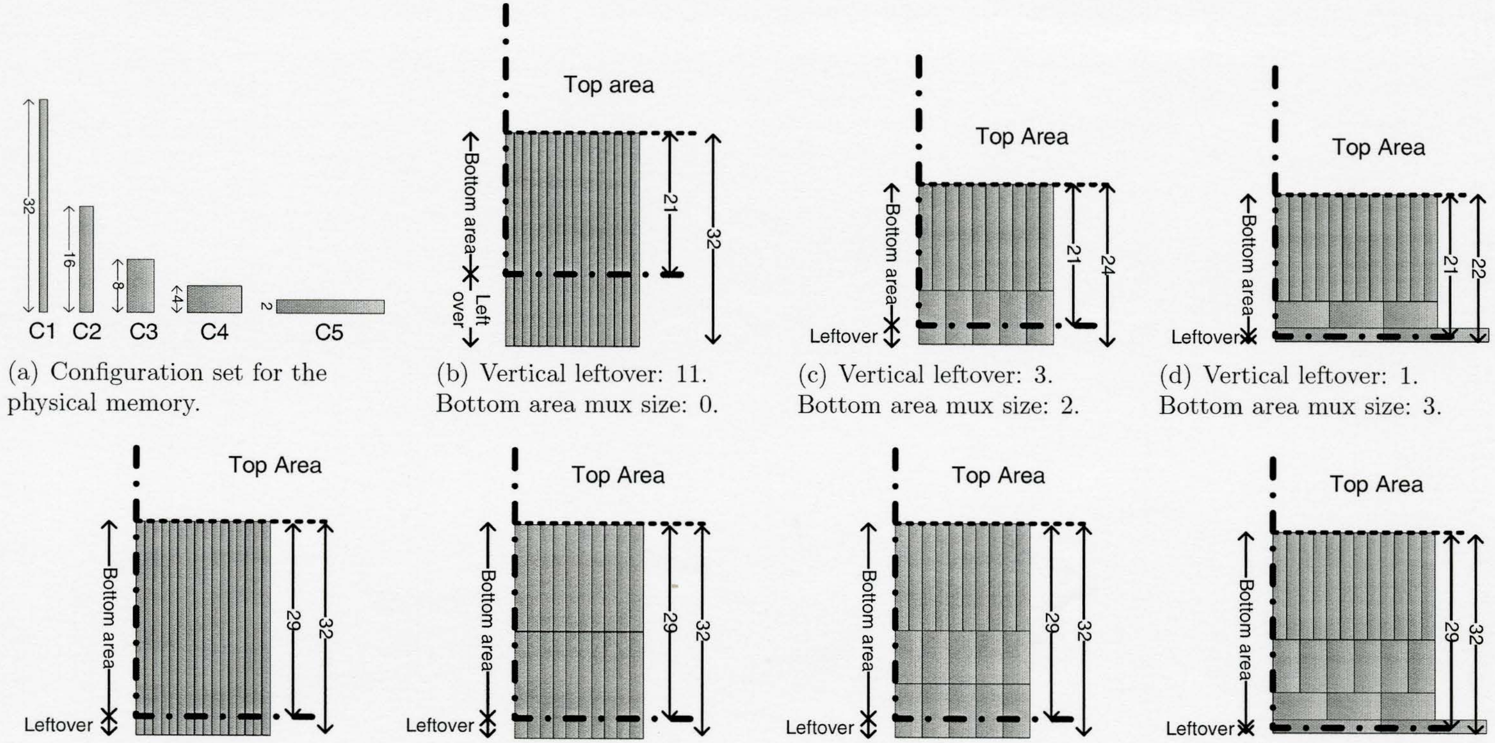
A simple method can help us find the set of configurations which minimize the vertical leftover using the third observation. Each time we want to add a memory to cover the remaining depth of the virtual memory, we have two options:

- Choosing the shortest configuration which is taller than the remaining depth (this option creates leftover).
- Choosing the tallest configuration which does not create leftover (this option decreases the remaining uncovered depth of the virtual memory which we will try to cover in the next level by repeating this decision again).

By choosing the second option every time we can find the desired set of configurations. The only concern is that having two identical configurations on top of each other is not acceptable because it will violate the first observation. Figure 3.3 shows the bottom area of two virtual memories and all the configurations of a certain physical memory which should be used to cover them.

It is apparent that as we proceed in the algorithm, we can come up with less leftover with the penalty of increasing the multiplexer cost. It should be emphasized that in some cases it is impossible to have different ways to cover an empty area. For example, with the given configurations in figure 3.3 and a depth of 31, we only have one option to cover the empty area because all other options lead to two identical configurations on top of each other which violates the first observation.

So far we have worked on the reduction of the vertical leftover in the bottom area of the virtual memory. However, the reduction of the vertical leftover can lead to more horizontal leftover because short configurations have a larger width. This



(a) Configuration set for the physical memory.

(b) Vertical leftover: 11. Bottom area mux size: 0.

(c) Vertical leftover: 3. Bottom area mux size: 2.

(d) Vertical leftover: 1. Bottom area mux size: 3.

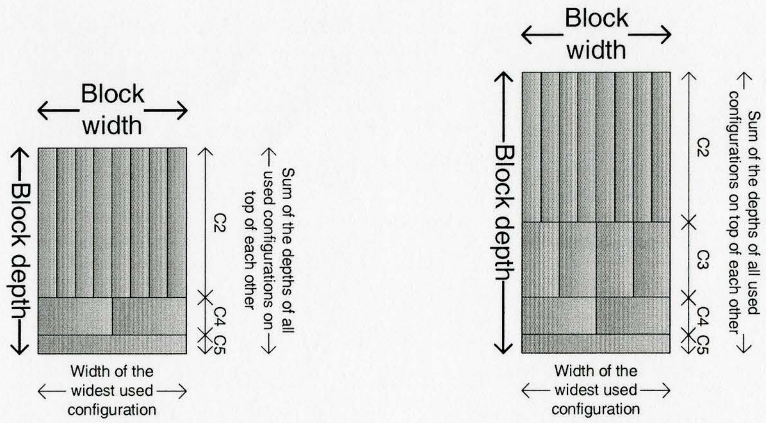
(e) Vertical leftover: 3. Bottom area mux size: 0.

(f) Not acceptable.

(g) Not acceptable.

(h) Vertical leftover: 1. Bottom area mux size: 4.

Figure 3.3: Decreasing the vertical leftover.


Block width
Block depth
C2
C4
C5
Sum of the depths of all used configurations on top of each other
Width of the widest used configuration
Block width
Block depth
C2
C3
C4
C5
Sum of the depths of all used configurations on top of each other
Width of the widest used configuration

(a) Block of physical memories used in figure 3.3(d).

(b) Block of physical memories used in figure 3.3(h).

Figure 3.4: Blocks of physical memories.

increase in the width of the configurations of physical memories can lead to a higher horizontal leftover.

Observation 4: *The widths of different configurations of a physical memory are a multiple of each other.*

In order to solve this problem we can create blocks of physical memories with the best configuration set we have found to minimize the vertical leftover. These blocks have a rectangular shape because the widths of different configurations are a multiple of each other (observation 4). Therefore, the width of the block will be the width of the shortest (widest) configuration which we use in the bottom, and the depth of the block is the sum of depths of the used configurations. Figure 3.4 shows the blocks in the two cases of figures 3.3(d) and 3.3(h).

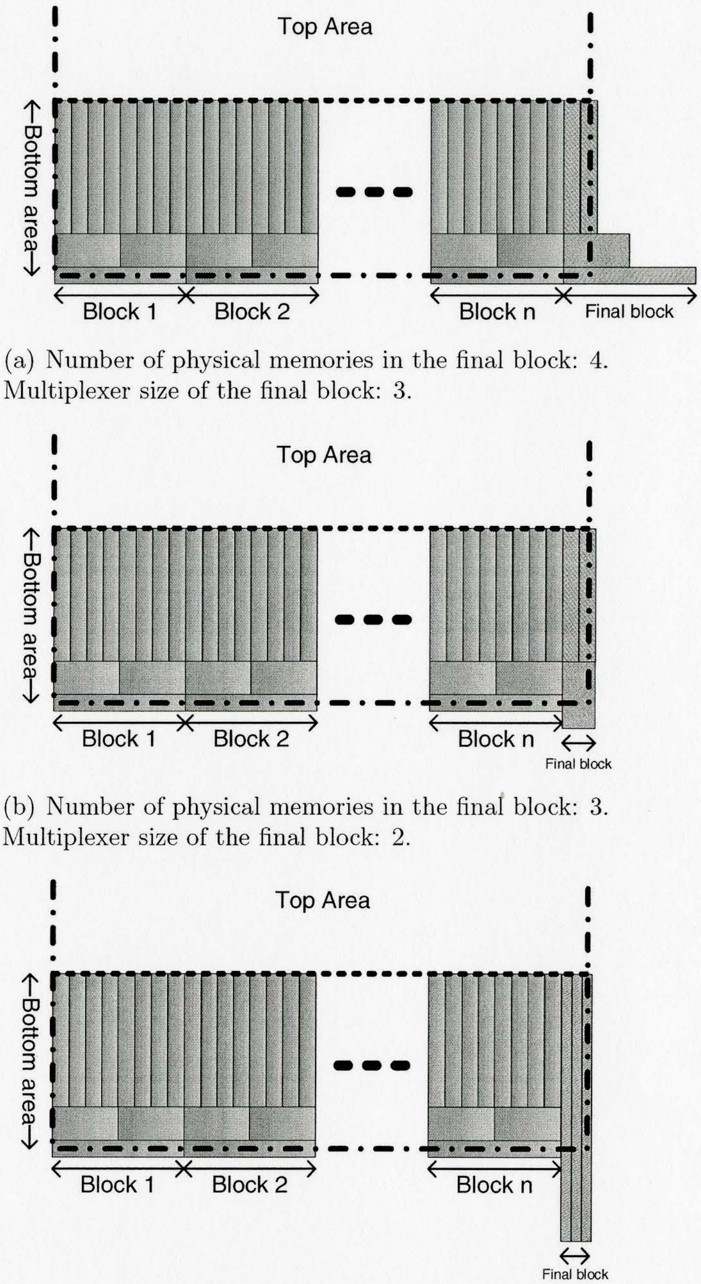
We can put these blocks of physical memories next to each other to cover the bottom area of the virtual memory. These blocks create the minimum possible vertical leftover. However, the final block which we use at the very right side of the virtual memory can create horizontal leftover. In some cases the replacement of this block with other configurations which we have found in the previous step to cover the depth of the region is helpful to reduce the overall leftover. It is obvious that this replacement causes a greater vertical leftover. However, on the other hand the decrease in the horizontal leftover can make up for this in many cases. We have to calculate the leftover, and choose the minimum case. Figure 3.5 shows the use of different configurations instead of the last physical memory block and the effect of each configuration on the overall leftover.

Apparently, the set of configurations used in the figure 3.5(c) results in the best overall leftover and multiplexer size. All the sets of configurations used in the final blocks of figure 3.5 are the results of intermediate steps of reducing the vertical leftover which are visible in figures 3.3(b) to 3.3(d).

To review what we have discussed in this subsection, algorithm 1 presents a pseudo code for the leftover minimization approach in a binding problem with one virtual memory and one type of physical memories. The inputs are:

- "$VM_{dimensions}$": The depth and width of the virtual memory.
- "$PM_{quantity}$": The number of available physical memory resources.
- "$PM_{configuration\_set}$": The list of available configurations (their widths and depths) for the physical memories.

The outputs are the solution to the problem which are the configuration of the



(a) Number of physical memories in the final block: 4. Multiplexer size of the final block: 3.

(b) Number of physical memories in the final block: 3. Multiplexer size of the final block: 2.

(c) Number of physical memories in the final block: 3. Multiplexer size of the final block: 0.

Figure 3.5: Different choices for the final block to reduce the overall leftover.

physical memories we have used in the solution (“$used\_PMs_{configuration}$”) and their exact position in the virtual memory(“$used\_PMs_{disposition}$”).

The “divide” function in the first line of the algorithm performs the procedure of dividing the virtual memory into “top” and “bottom” areas. This division is done based on the depth of the tallest configuration of the physical memory and the depth of the virtual memory, and as illustrated in line 1, these two parameters are the inputs of the “divide” function.

Then, we use the “cover” function in the second line to cover the top area of the virtual memory with the tallest configuration of the physical memory, and we save the coverage for the top area by updating the information of physical memories we have used in the solution by using the “update” function.

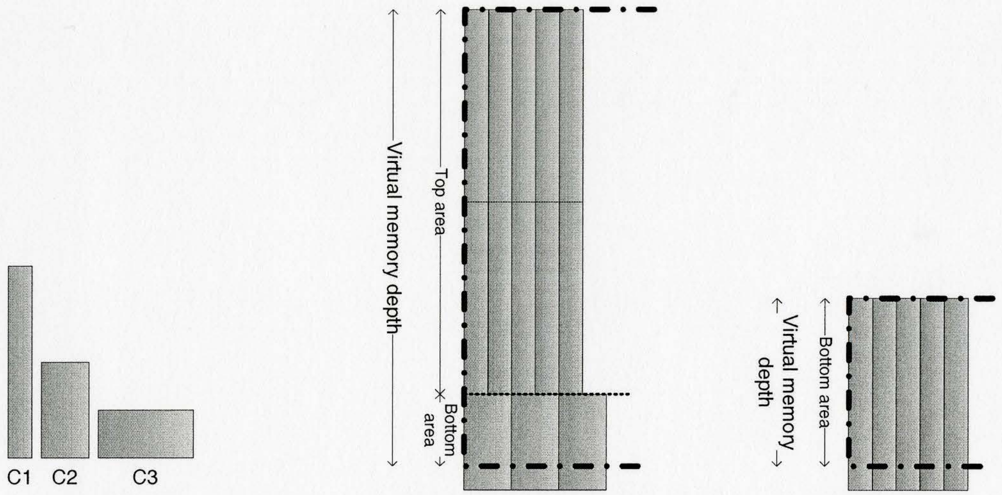
In line 4, the “block_creator” function creates all possible combinations of the physical memory configurations on top of each other to cover the depth of the bottom area of the virtual memory with respect to the first observation, and stores them in “$block_{list}$”. The “while loop” in the fifth line covers the bottom area of the virtual memory with the block with the smallest heights among all blocks in the “$block_{list}$” as long as it does not create any horizontal leftover, which is captured in the condition of the “while loop”.

The “for loop” in line 9 covers the bottom-right part of the virtual memory with one of the blocks in “$block_{list}$” each time. The “leftover_calculator” function in the twelfth line calculates the leftover of the generated solution by subtracting the capacity of the virtual memory from the sum of the capacities of the used physical memories in the solution, and stores the result in “$leftover_{current}$”. Also, the “$leftover_{best}$” stores the leftover of the best solution so far. If the currently generated solution has a smaller leftover than the best solution so far, we will overwrite the best solution and its leftover by the current solution. This procedure is captured in the "if statement" in line 13, and as presented in line 14; the "save" function stores the solution by saving the configuration and position of the used physical memories in the solution. Finally, the algorithm will return these values as its output.

Algorithm 1: Single VM - Single PM type leftover minimization

```
   Input  : $VM_{dimensions}$, $PM_{quantity}$, $PM_{configuration\_set}$
   Output: $used\_PMs_{configuration}$, $used\_PMs_{disposition}$

 1 $(top\_area, bottom\_area)$ = divide($VM_{dimensions}$,
   $PM_{tallest\_configuration} \in PM_{configuration\_set}$);
 2 cover($top\_area$, $PM_{tallest\_configuration} \in PM_{configuration\_set}$);
 3 update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
 4 $block_{list}$ = block_creator($bottom\_area_{depth}$, $PM_{configuration\_set}$);
 5 while $(leftover_{horizontal} = 0)$ do
 6     cover($bottom\_area$, $block_{shortest} \in block_{list}$);      //to reduce vertical leftover
 7     update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
   end
 8 $leftover_{best} = \infty$;
 9 foreach $block \in block_{list}$ do
10     cover($bottom\_area$, $block$);
11     update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
12     $leftover_{current}$ = leftover_calculator($VM_{dimensions}$, $used\_PMs_{configuration}$);
13     if $(leftover_{current} < leftover_{best})$ then
14         save($used\_PMs_{configuration}$, $used\_PMs_{disposition}$); //save current solution
15         $leftover_{best} = leftover_{current}$;
       end
   end
16 Return $used\_PMs_{configuration}$, $used\_PMs_{disposition}$;
```



(a) Configuration set for the physical memory

(b) 3 to 1 multiplexers

(c) No multiplexers

Figure 3.6: Minimum multiplexer cost binding.

3.1.2 Binding for Multiplexer Cost Minimization

When we intend to minimize the multiplexer cost, we can simply choose the shortest configuration which completely covers the depth of the bottom area. Figure 3.6 shows the result of applying this method for two different virtual memories.

In fact, the solution with the minimum leftover gives the minimum number of required physical memories to cover the virtual memory, and the solution with the minimum multiplexer cost is the one with the maximum number of physical memories which we use in a reasonable solution. Therefore, when the number of available physical memories is less than the number we need to create the minimum leftover, the problem does not have a solution. Also, when the given number of physical memories is greater than or equal to the number of required physical memories to come up with the minimum multiplexer cost, we can easily solve the problem without any concerns. However, we need to have a different approach to solve the problem when the given number of physical memories is between the minimum (number of required physical memories for minimum leftover) and the maximum (number of required physical memories for minimum multiplexer cost).

In this case, we have no problem with the minimum leftover solution because it consumes a lower number of physical memories than the given amount. However, when calculating the binding for the minimum multiplexer cost solution we have to deal with a new constraint, which is the number of used physical memories, so we cannot simply accept the solution mentioned at the beginning of this subsection. Instead, we want to use all of the given physical memories in a way that minimizes the multiplexer cost.

As mentioned earlier, we have filled the top part in an optimum way. Therefore, the solution in this case lies in the arrangement of physical memories in the bottom area of the virtual memory.

In subsection 3.1.1 we went through a number of steps to find all the sets of different configurations of physical memories to fill the bottom area of the virtual memory. Also, we defined blocks of physical memories based on each set of configurations. Now, the solution we are looking for is the correct combination and arrangement of these blocks. We may exclude some configurations or change the amount of area they cover to create all possible combinations. Figure 3.7 shows a sample combination of different configurations for the first virtual memory in figure 3.3.

The pseudo code for minimum multiplexer cost binding is presented in algorithm 2. In the first 3 lines of the algorithm we perform the exact procedure that we did in algorithm 1, and cover the top area of the virtual memory. In line 4, the

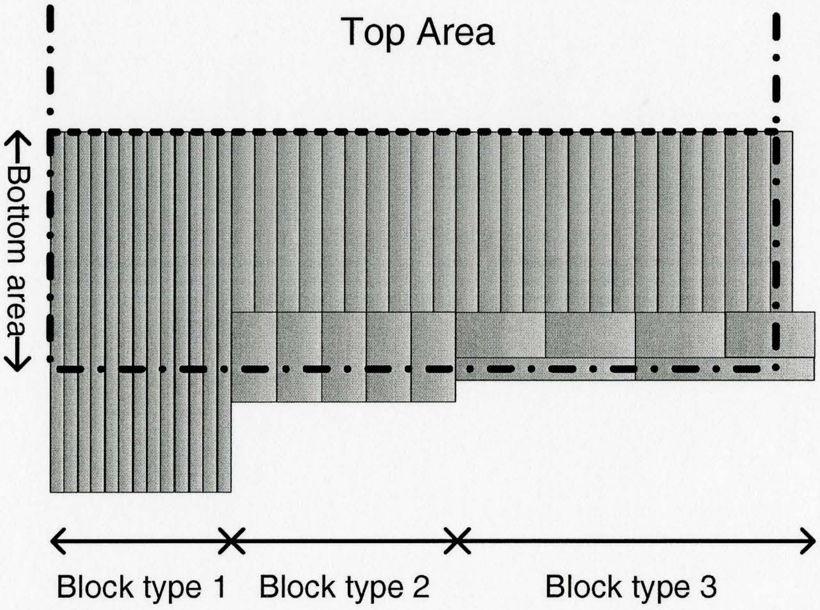


Figure 3.7: A sample combination of different blocks to reduce the multiplexer size

"max_PM_calculator" function calculates the maximum number of required physical memories to solve the problem in the case of minimum multiplexer cost, and stores this number in "$PM_{max\_required}$". Based on this number, in line 5 we choose the suitable strategy for multiplexer cost minimization. If we have sufficient available physical memories we cover the bottom part with the shortest configuration of the physical memory which is taller or equal to the depth of the bottom area ($PM_{suitable\_configuration}$). This strategy is captured in lines 6 to 8.

In the other case, which is captured in lines 9 to 18, we create all blocks of physical memories with respect to the depth of the bottom area, and save them in "$block_{list}$". The "for loop" in line 12 is in charge of creating all possible combinations of blocks next to each other ($block_{set}$) to cover the bottom area. The multiplexer cost

Algorithm 2: Single VM - Single PM type multiplexer cost minimization

```
Input  : $VM_{dimensions}$, $PM_{quantity}$, $PM_{configuration\_set}$
Output: $used\_PMs_{configuration}$, $used\_PMs_{disposition}$

($top\_area$, $bottom\_area$) = divide($VM_{dimensions}$);
cover($top\_area$, $PM_{tallest\_configuration} \in PM_{configuration\_set}$);
update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
$PM_{max\_required}$ = max_PM_calculator($VM_{dimensions}$, $used\_PMs_{configuration}$,
$used\_PMs_{disposition}$);
if (($PM_{quantity}$ - # of $used\_PMs$) $\geq PM_{max\_required}$) then
    $PM_{suitable\_configuration}$ =
    shortest($PM_{configuration} \in PM_{configuration\_set}$) > $bottom\_area_{depth}$;
    cover($bottom\_area$, $PM_{suitable\_configuration}$);
    update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
end
else
    $block_{list}$ = block_creator($bottom\_area_{depth}$, $PM_{configuration\_set}$);
    $multiplexer\_cost_{best} = \infty$;
    foreach $block_{set} \subseteq block_{list}$ do
        cover($bottom\_area$, $block_{set}$);
        update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
        $multiplexer\_cost_{current}$ = mux_cost_calculator($VM_{dimensions}$,
        $used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
        if ($multiplexer\_cost_{current} < multiplexer\_cost_{best}$) then
            save($used\_PMs_{configuration}$, $used\_PMs_{disposition}$); //save current solution
            $multiplexer\_cost_{best} = multiplexer\_cost_{current}$;
        end
    end
end
Return $used\_PMs_{configuration}$, $used\_PMs_{disposition}$;
```

of each generated solution is calculated with "mux_cost_calculator" function using the dimensions of the virtual memory and all used physical memories in the solution, and stored in "$multiplexer\_cost_{current}$". The "if statement" in line 16 compares the multiplexer cost of the current solution with the multiplexer cost of the best solution so far ("$multiplexer\_cost_{best}$"), and overwrites the best solution and its multiplexer cost so far if the current solution gives a smaller multiplexer cost.

3.2 Multiple VMs - Single PM Type

To make the problem we discussed in the previous section more general we can work with multiple virtual memories. Therefore, in this section we will work on minimization of leftover and multiplexer cost in the case of multiple virtual memories and one type of physical memories. The overall multiplexer cost and leftover will be the sum of all multiplexer costs and leftovers for all virtual memories.

In many cases, working on the virtual memories independently can lead to minimization of the overall leftover and multiplexer cost because the resources are identical, so the use of a certain physical memory in a virtual memory does not limit the choices for other virtual memories. However, this is not always the case. We will discuss the problem in its different cases in more details below.

To minimize the overall leftover, we can minimize the leftover in each virtual memory independently. Therefore, the same method introduced in the previous section can be applied to each virtual memory to minimize the leftover in binding process. In the case of minimizing the overall multiplexer cost with adequate number of physical memories, we can apply the same algorithm in the previous section and work on the virtual memories independently.

The minimum number of physical memories we need to solve the problem is the number of used physical memories in all virtual memories when the leftover is minimized. Also, the maximum number of physical memories we will consume to solve the problem is the number of used physical memories in all virtual memories when the multiplexer cost is minimized.

Therefore, when the given number of physical memories is less than the minimum required amount, the problem does not have a solution, and if the given number is larger than or equal to the maximum consumption, we can easily create the solution as discussed above (working on virtual memories independently). However, when we do not have enough number of physical memories to generate the minimum overall multiplexer cost, we cannot work on the virtual memories independently, and this is the only added complexity in this section.

In this case coming up with the minimum leftover solution does not change. However, finding the minimum multiplexer cost solution needs some extra effort. We have to use all the given physical memories to reduce the multiplexer cost as much as possible. In order to do that, we will divide all the virtual memories to top and bottom areas, and fill all top areas by employing the method in the previous section. To fill the bottom areas of the virtual memories we have to create all combinations of physical memory blocks for all the virtual memories. The combinations for each virtual memory are produced from its own blocks, and the blocks are generated with the same method we used in the previous section for each virtual memory.

The pseudo code for generating the minimum multiplexer cost binding in the case of multiple virtual memories and one type of physical memories is presented in algorithm 3. In the first 4 lines of the algorithm we perform the procedures of the previous algorithm on each virtual memory individually. Also, in line 5 the maximum number of required physical memories is calculated for each virtual memory, and the overall number of required physical memories is stored in "$PM_{max\_required}$". In the case of having enough number of physical memories we work on virtual memories individually, and use the method that we use in the case of a single virtual memory for each virtual memory (lines 6 to 10).

However, in the case of having not enough of physical memories, which is captured in the "else" statement in line 11, we create the block list for each virtual memory (VM_{block_list}). The nested loops in lines 15 and 16 create all the possible combinations of the blocks to cover the bottom areas of all virtual memories. The "$block_{set}$" is a list with its size equal to the number of virtual memories, and we store all possible combinations of the combinations of different blocks for different virtual memories in it, so in each location of the "$block_{set}$" we have a "$block_{set}$". We compare each generated solution with the best solution so far in line 20, and save the solution if it has a better multiplexer cost than the best solution so far.

Most of today‘s FPGAs have only one type of embedded memory. Therefore, the presented method can lead to the solution for many of today's problems. However, a new generation of FPGAs is emerging with several different types of embedded memories. This problem is addressed in the next section.

3.3 Single VM - Multiple PM Types

In this section we will try to make the problem more general from another perspective. Some FPGAs have more than one type of embedded reconfigurable physical memories to provide more flexibility in saving the on chip data. This increase in the number of

Algorithm 3: Multiple VMs - Single PM type multiplexer cost minimization

```
Input  : $VMs_{dimensions}$, $PM_{quantity}$, $PM_{configuration\_set}$
Output: $used\_PMs_{configuration}$, $used\_PMs_{disposition}$

foreach (VM) do
    ($VM_{top\_area}$, $VM_{bottom\_area}$) = divide($VM_{dimensions}$);
    cover($VM_{top\_area}$, $PM_{tallest\_configuration} \in PM_{configuration\_set}$);
    update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
end
$PM_{max\_required}$ = max_PM_calculator($VMs_{dimensions}$, $used\_PMs_{configuration}$,
$used\_PMs_{disposition}$);
if (($PM_{quantity}$ - # of $used\_PMs$) $\geq PM_{max\_required}$) then
    foreach (VM) do
        $PM_{suitable\_configuration}$ =
        shortest($PM_{configuration} \in PM_{configuration\_set}$) > $VM_{bottom\_area\_depth}$;
        cover($VM_{bottom\_area}$, $PM_{suitable\_configuration}$);
        update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
    end
end
else
    foreach (VM) do
        $VM_{block\_list}$ = block_creator($VM_{bottom\_area\_depth}$, $PM_{configurations}$);
    end
    $multiplexer\_cost_{best} = \infty$;
    foreach $block_{set}$[1 .. # of VMs] $\subseteq$ $VM_{block\_list}$[1 .. # of VMs] do
        foreach $i \in$ [1 .. # of VMs] do
            cover($VM_{bottom\_area}[i]$, $block_{set}[i]$);
            update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
        end
        $multiplexer\_cost_{current}$ = mux_cost_calculator($VMs_{dimensions}$,
        $used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
        if ($multiplexer\_cost_{current} < multiplexer\_cost_{best}$) then
            save($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
            $multiplexer\_cost_{best} = multiplexer\_cost_{current}$;
        end
    end
end
Return $used\_PMs_{configuration}$, $used\_PMs_{disposition}$;
```

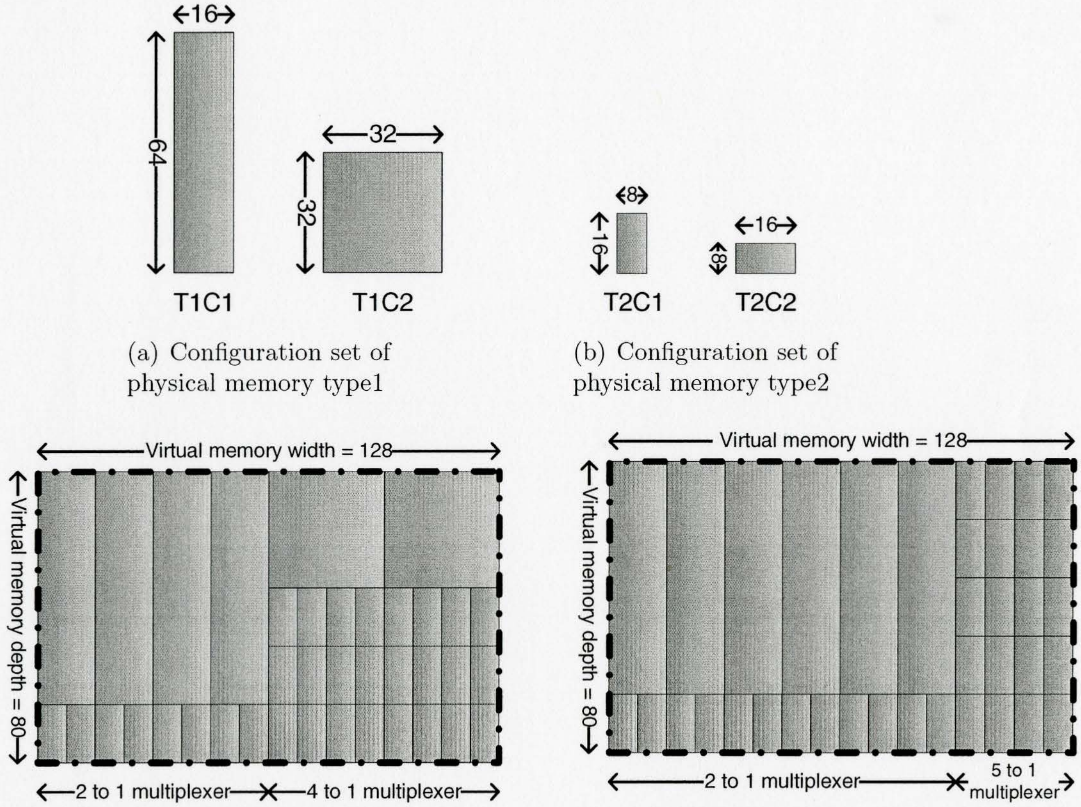
options can increase the complexity of the algorithms which we will use to find the bindings for minimum multiplexer size and leftover.

In the two previous cases we only dealt with choosing the proper configuration of the resources. However, in this case, in addition to the previous decision, we have to decide on the number of each type of physical memories in the solution too. Moreover, each type has its own number and configuration set. Unlike the previous sections preference of the tallest configuration of physical memories does not always result in reduction of the multiplexer cost. Figure 3.8 shows a comparison between using two different configurations in the case of having more than one type of physical memories. It is shown that the taller configuration does not necessarily result in a smaller multiplexer cost.

Assuming C_n is the cost of a single bit n:1 multiplexer, we can compare the multiplexer cost in figure 3.8(c) and 3.8(d) as follows:

$$64 \times C_2 + 64 \times C_4 \quad \textit{versus} \quad 96 \times C_2 + 32 \times C_5$$
$$\Longrightarrow \quad 64 \times C_4 \quad \textit{versus} \quad 32 \times C_2 + 32 \times C_5$$

It is not possible to compare the two sides of the equation unless we have the exact values of C_2, C_4, and C_5. Therefore, we cannot prove that the use of tall configurations always leads to a lower multiplexer cost. In fact, tall configurations can improve the multiplexer cost locally, but in many cases due to the lack of available large physical memories we will have to cover some areas in the absence of large memories with smaller physical memories (e.g. the right area in figure 3.8(d)). Therefore, this increase in multiplexer size in other areas can increase the overall multiplexer cost. On the other hand, as illustrated in figure 3.8(c), by using the configurations of large physical memories which are shorter than the shortest configuration which covers the


16
64
32
32
T1C1
T1C2
8
16
16
8
T2C1
T2C2
Virtual memory width = 128
Virtual memory depth = 80
2 to 1 multiplexer
4 to 1 multiplexer
Virtual memory width = 128
Virtual memory depth = 80
2 to 1 multiplexer
5 to 1 multiplexer

(a) Configuration set of physical memory type1

(b) Configuration set of physical memory type2

(c) A sample placement with tall and short configurations

(d) A sample placement with only tall configurations

Figure 3.8: Comparison between two placements with and without short configurations.

entire depth of the empty part in the virtual memory, we can distribute them to a larger multiplexer cost area in the virtual memory. Therefore, it is critical to account for all configurations of physical memories when we have more than one type of them.

In order to attack the problem in this case and come up with the best leftover or multiplexer cost solution, we have to create all possible solutions and compare them with each other. First, we have to sort the physical memories based on their capacity in a decreasing order, and start from the largest physical memory. In each step of the algorithm we work with only one type of physical memories. The first job is to find the maximum number for the current type in our solution. We simply find this maximum assuming that we want to cover the entire uncovered area with this type in minimum multiplexer cost format. After finding the maximum number we proceed with the algorithm by creating different branches of solutions. In each branch we have a different number for current type between zero and the maximum. This will help us to create all solutions with different numbers of the current physical memory.

In the next step we have to create all possible combinations of configurations for each assigned number, and finally we have to create all possible combinations of placements for each combination of the configurations. The only limitation is that we do not violate the first observation except for the tallest configurations. It is also accepted that for some sets of configurations we may not be able to come up with a feasible placement, so we simply ignore these sets.

After finishing this step each generated solution has covered a certain area of the virtual memory. Now we select the next physical memory type in the list and repeat every step of the algorithm for the remaining part of the virtual memory. The final type of the physical memory will be processed differently. We can reuse the algorithms we had in the section 3.1. However, in this case the empty area is not necessarily rectangular anymore.

The pseudo code for what we discussed in this section for solving the problem of single virtual memory and multiple types of physical memories is presented in algorithm 4. In this algorithm we have 2 sets of outputs, one for the minimum leftover solution ($best\_solution_{leftover}$) and one for multiplexer cost solution ($best\_solution_{muliplexer\_cost}$), and each of these solutions consists of the used physical memories type and configuration ($used\_PMs_{type\&configuration}$) and their exact position in the virtual memory ($used\_PMs_{disposition}$). Also, the physical memory inputs of the algorithm are the different types of physical memories (PM_types) and the number of each type ($PM\_types_{quantity}$) and the set of configurations assigned to each type ($PM\_types_{configuration\_set}$).

The leftover of the best leftover solution is stored in "$leftover_{best}$", and the multiplexer cost for the best multiplexer cost solution is saved in '$multiplexer\_cost_{best}$'". We sort the physical memory types based on their capacity in line 3, and the "for loop" in line 4 uses the first n-1 types. In line 5 we calculate the maximum number of the current type of the physical memories that we may require in the solution based on the empty area in the virtual memory and the capacity of the current type. The empty area of the virtual memory can be calculated based on the dimensions of the virtual memory and the dimensions of the used physical memories so far. The "for loop" in line 6 guarantees that we assign all different numbers for the current type in the solutions, and in line 7 we create all the combinations of the configurations for the assigned number of the current type using the "config_combination_generator" function, and store them in "$configs\_list$". Each entry of the "$configs\_list$" is a combination of configurations of the assigned number of physical memories called "$configs\_set$". In line 9, the "position_combination_generator" function creates all possible sets of placements for each "$configs\_set$", and stores them in *pos_list*. Each entry of the *pos_list* consists of the set of positions for the set of configurations in hand called "*pos_set*". In the core of these nested loops we update the "$used\_PMs_{configuration}$" and "$used\_PMs_{disposition}$" as a temporary solution. Then, we call the routine for 1 virtual memories and multiple types of physical memories in line 12 to cover the remaining empty parts in the virtual memory. After calculating the multiplexer and leftover cost of the generated solution we save the solution if it is better than what we have achieved so far in lines 15 to 20 in terms of leftover or multiplexer cost.

3.4 Multiple VMs - Multiple PM Types

Binding a number of virtual memories to multiple types of reconfigurable physical memories is the most general and complicated problem we want to discuss in this thesis. Moreover, the algorithm which is capable to solve this problem can solve all the subproblems we discussed previously.

When we are working with multiple types of physical memories we cannot work on the virtual memories independently to find the minimum overall leftover or multiplexer cost because the use of each resource in each virtual memory limits the number of this particular resource in other virtual memories. Therefore we have a new dimension in the algorithms for the current problem compared to section 3.3. The overall approach is the same. However, we have to work on all virtual memories at the same time.

While creating all combinations of assigned physical memories of the current type

Algorithm 4: Single VM - Multiple PM types - exhaustive solution search

```
Input  : $VM_{dimensions}$, $PM\_types$, $PM\_types_{quantity}$, $PM\_types_{configuration\_set}$
Output: $best\_solution_{muliplexer\_cost}$, $best\_solution_{leftover}$

$multiplexer\_cost_{best} = \infty$;
$leftover_{best} = \infty$;
sort($PM\_types$);
foreach ($PM\_type \neq PM_{smallest\_type}$) do
    $PM_{max\_required}$ = max_PM_calculator($VM_{dimensions}$,
    $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$, $PM\_type$);
    foreach ($i \in [0, min(PM\_type_{quantity}, PM_{max\_required})]$) do
        $configs\_list$ = config_combination_generator($i$, $PM\_type_{configuration\_set}$);
        foreach ($configs\_set \in configs\_list$) do
            $pos\_list$ = position_combination_generator($configs\_set$, $VM$);
            foreach ($pos\_set \in pos\_list$) do
                update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
                call($1VM - 1PM$);
                $multiplexer\_cost_{current}$ = mux_cost_calculator($VM_{dimensions}$,
                $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$);
                $leftover_{current}$ = leftover_calculator($VM_{dimensions}$,
                $used\_PMs_{type\&configuration}$);
                if ($multiplexer\_cost_{current} < multiplexer\_cost_{best}$) then
                    $best\_solution_{muliplexer\_cost}$ =
                    $used\_PMs_{type\&configuration}, used\_PMs_{disposition}$;
                    $multiplexer\_cost_{best} = multiplexer\_cost_{current}$;
                end
                if ($leftover_{current} < leftover\_best$) then
                    $best\_solution_{leftover}$ =
                    $used\_PMs_{type\&configuration}, used\_PMs_{disposition}$;
                    $leftover_{best} = leftover_{current}$;
                end
            end
        end
    end
end
Return $best\_solution_{muliplexer\_cost}$, $best\_solution_{leftover}$;
```

to virtual memories, instead of a single number we produce an array of numbers. The size of the array is equal to the number of virtual memories, and each number is the number of assigned physical memories of that type to its index virtual memory, and the sum of entries is the total number of used physical memories of that type in current solution. We work on all virtual memories at the same time through all steps of the algorithm.

The pseudo code for generating all possible solutions for the general problem is provided in algorithm 5. As illustrated in lines 5 and 6, in the case of multiple virtual memories we calculate the maximum required number of physical memories of the current type for each virtual memory individually. Therefore, assigning the number of used physical memories of the current type in the current solution is different in this case. First we create all the possible numbers of assigned physical memories to the overall solution ('i') in line 7, and after that, we create all the combinations of the assigned numbers to virtual memories with the sum of 'i'. This task is done by "number_combination_generator" function and the results are stored in "*assigned_list*". Each entry of the "" consists of the set of assigned numbers to virtual memories called "*assigned_set*".

The rest of the algorithm is similar to algorithm 4. The only difference is that in this case the "leftover_calculator" function calculates the leftover of the generated solution by subtracting the sum of the capacities of virtual memories from the sum of the capacities of all used physical memories in the solution.

Algorithm 5: Multiple VM - Multiple PM types exhaustive solution search

```
Input   : $VMs_{dimensions}$, $PM\_types$, $PM\_types_{quantity}$, $PM\_types_{configuration\_set}$
Output  : $best\_solution_{muliplexer\_cost}$, $best\_solution_{leftover}$

$multiplexer\_cost_{best} = \infty$;
$leftover_{best} = \infty$;
sort($PM\_types$);
foreach ($PM\_type \neq PM_{smallest\_type}$) do
    foreach ($VM$) do
        $VM_{PM\_max\_required}$ = max_PM_calculator($VM_{dimensions}$,
        $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$, $PM\_type$);
    end
    foreach ($i \in [0, min(PM\_type_{quantity}, \sum VM_{max\_required})]$) do
        $assigned\_list$ = number_combination_generator($i$, # of VMs);
        foreach ($assigned\_set \in assigned\_list$) do
            foreach ($j \in assigned\_set$) do
                $configs\_list$ = config_combination_generator($j$, $PM\_type_{configurations}$,
                $VM_j$);
                foreach ($configs\_set \in configs\_list$) do
                    $pos\_list$ = position_combination_generator($configs\_set$, $VM_j$);
                    foreach ($pos\_set \in pos\_list$) do
                        update($used\_PMs_{configuration}$, $used\_PMs_{disposition}$);
                        call($nVM - 1PM$);
                        $multiplexer\_cost_{current}$ = mux_cost_calculator($VMs_{dimensions}$,
                        $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$);
                        $leftover_{current}$ = leftover_calculator($VMs_{dimensions}$,
                        $used\_PMs_{type\&configuration}$);
                        if ($multiplexer\_cost_{current} < multiplexer\_cost_{best}$) then
                            $best\_solution_{muliplexer\_cost}$ =
                            $used\_PMs_{type\&configuration}, used\_PMs_{disposition}$;
                            $multiplexer\_cost_{best} = multiplexer\_cost_{current}$;
                        end
                        if ($leftover_{current} < leftover\_best$) then
                            $best\_solution_{leftover}$ =
                            $used\_PMs_{type\&configuration}, used\_PMs_{disposition}$;
                            $leftover_{best} = leftover_{current}$;
                        end
                    end
                end
            end
        end
    end
end
Return $used\_PM\_types_{configuration}$, $used\_PM\_types_{disposition}$;
```

3.5 A Heuristic Approach

In the previous section we tried to discuss the complexity of the general problem, which was binding multiple virtual memories to multiple types of reconfigurable physical memories. Moreover, we presented a method to explore the entire solution space to find the desired solution for the problem. However, due to the complexity of the problem, coming up with an exact solution by exploring the entire solution space does not seem to be a practical method because of its runtime. Therefore, we have to use heuristic methods to achieve a suboptimal solution within an acceptable time.

In this section we will discuss two different heuristic approaches for the two parts of the problem, which are leftover and multiplexer cost minimization respectively. The following two subsections explain the steps of the heuristic approaches in each case for the problem of binding multiple virtual memories to multiple types of physical memories.

3.5.1 Heuristics for Leftover Reduction

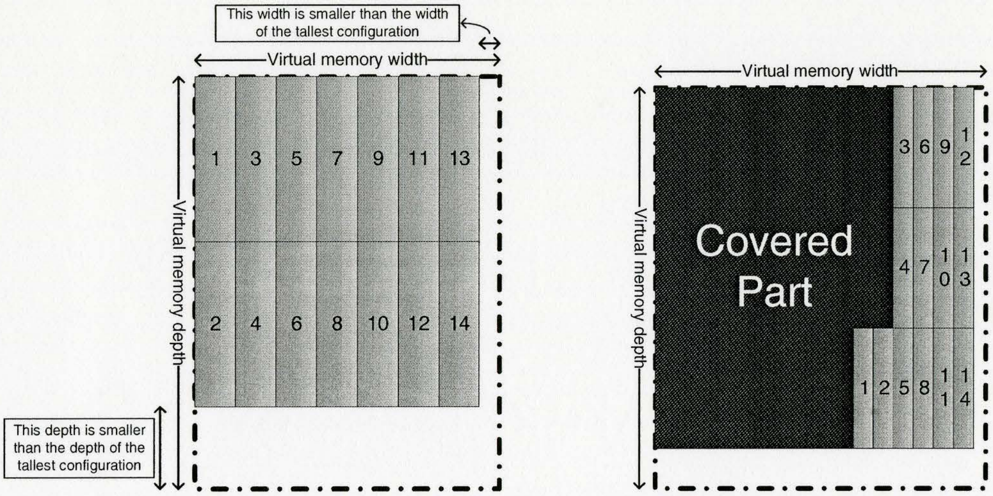
The basic idea of the heuristic algorithm for leftover reduction is to use larger types of physical memories as long as they do not create any leftover. Therefore, it sorts the types of physical memories based on their capacity and starts with the largest type to fill the virtual memories. The algorithm tries to work on all the virtual memories at the same time, and in each step it only works with one type of physical memory.

Filling the uncovered parts of virtual memories with a type of physical memories has three steps with three different strategies. As mentioned earlier, we start with the largest type of physical memory, and apply these three steps on the virtual memories using the current type. Then, the output is given to the next part which applies the same three steps to the remaining empty parts using the next type of physical memories. Here, it is very important to know when to move from one strategy to another and from one type to the next type. We will discuss the three strategies and the conditions for switching between them in the algorithm:

Step 1: In this step we will only use the tallest configuration of the physical memory to cover the empty parts of the virtual memories. However, we do not allow any types of leftover. We sort the virtual memories based on their depth increasingly, and start with the shortest one, and whenever the current virtual memory cannot accommodate more physical memories of the current type in their tallest configuration without any leftover, we move to the next virtual memory.

In this step we fill the virtual memories column-wise. Figure 3.9 shows two sample virtual memories and the way we fill each one. It is also important to mention that the empty area is not necessarily rectangular always because of the parts that the previous physical memories have covered so far in the virtual memory. Also, we always have to keep track of the number of available physical memories and the number that we have used so far in the algorithm.

It is important to know that the cost of a multiplexer grows linearly with respect to the number of its inputs, so reducing the number of inputs in a large multiplexer saves the same amount of logic compared to a small multiplexer. Therefore, covering an empty part in a virtual memory can be done column-wise or row-wise without any visible difference in the overall multiplexer cost. Moreover, starting with the shortest virtual memory does not imply any advantages or shortcomings in the multiplexer cost of the solution compared to the case


This width is smaller than the width of the tallest configuration
Virtual memory width
Virtual memory depth
1
3
5
7
9
11
13
2
4
6
8
10
12
14
This depth is smaller than the depth of the tallest configuration
Virtual memory width
Virtual memory depth
Covered Part
3
6
9
12
4
7
10
13
1
2
5
8
11
14

(a) First used type in the virtual memory.

(b) Virtual memory has been covered partially by previous types of physical memories.

Figure 3.9: Filling the virtual memories in the first step.

that we start with the tallest virtual memory.

If we finish this step, and the current type of physical memories is still available, we can go to step 2. Otherwise, we have to go to the next type of physical memories and run the step 1 for it.

Step 2: In this step of the algorithm we will work on the bottom part of the virtual memories which is not covered yet. To start, we sort the virtual memories based on the depth of their bottom uncovered part, and start with the virtual memory with the deepest uncovered part in its bottom area. The strategy in this step is again not to create any type of leftover. Therefore, we have to use other configurations of the physical memory which do not create any horizontal or vertical leftover; again, we prefer tall configurations, and that is why we try to work on deeper uncovered parts first.

We continue the strategy in this step until we run out of physical memories of the current type, or there are no more spots left in the virtual memories which we can cover without creating any leftover. As of this point in the algorithm the use of this type of physical memory creates leftover, so we prefer not to use it, and go for smaller types which create less leftover. However, the problem is we may have not enough of those smaller types to cover all parts of all virtual memories. Therefore, we need an estimation of the number of smaller physical memories (remaining types) which are needed for the rest of solution of the problem, and if we need more smaller physical memories than are available, we have to use more of the current type with the penalty of creating leftover, which leads us the step 3.

The estimation function is a very simple algorithm which at the beginning calculates the number of current physical memory type required to cover the remaining parts of all virtual memories for minimum multiplexer cost solution. It has been discussed in section 3.1 that the minimum multiplexer cost solution gives the maximum required number of physical memories to come up with an acceptable solution for the problem.

We can estimate the overall memory capacity we need to solve the rest of the problem by multiplying the number we calculated above to the capacity of the current type of the physical memories. Then, by comparing this number with the sum of available capacities for all smaller types of physical memory, which is the overall capacity we have still available we can conclude whether we have enough physical memories in other types or not. In the case of having not enough physical memories in other types we have to continue using the current type with the cost of additional leftover. Adding extra physical memories of the current type is discussed in step 3.

Step 3: We keep executing the strategy in this step until we make sure that the number of physical memories in smaller types is adequate to cover the remaining empty parts in all virtual memories. In this step, we know that adding each physical memory creates some leftover. Therefore, we create a list of all the candidate spots for the current type of physical memory and the leftover created for each spot. Then, we sort this list increasingly based on the created leftover, and add the physical memories one by one. After adding each physical memory we rerun the estimation function to decide whether we should still work with the current type or whether we can move to the next type.

It is also obvious that we can only run this step while we have available physical memories of the current type. Therefore, we should keep track of the used and available numbers of physical memories in the current type. After finishing this step we can move to the next type of physical memories and start from step 1 for that type.

This algorithm should be used for the first n-1 types of physical memories, and we can reuse the algorithms in section 3.2 for the last type of physical memories because of its desirable runtime and optimum results. The only difference is that the empty areas in this case are not necessarily rectangular.

Algorithm 6 provides a pseudo code to summarize what we discussed for heuristics to reducing the leftover in a solution. Lines 3 to 6 represent the first step of the algorithm, where we only use the tallest configuration of the physical memory to cover the virtual memory without creating any leftover. In line 7 we sort the virtual memories based on the depth of their uncovered bottom part, and in lines 8 to 11 we try to use the tallest possible configuration of the current type of the physical memories to cover the empty parts at the bottom of the virtual memories without creation of leftover. In line 12 we have the estimation function (estimate) which calculates the estimated number of required physical memories in smaller types to complete the solution to the problem using the capacities of the smaller types of physical memories and the area of the empty parts in all virtual memories. The result is stored in in "$PM\_next\_type_{required\_number}$". The comparison of this number with the actual number of available physical memories in smaller types provides the condition for the third step of the algorithm. Based on this condition we add a single physical memory in a place in virtual memories with minimum implied leftover each time, which is done by "add" function. Finally, the "$nVM-1PM$" function is called to cover the remaining uncovered parts of virtual memories (if there is any) with the smallest type of physical memory which is the last type.

3.5.2 Heuristics for Multiplexer Cost Reduction

Similar to the leftover heuristics, the heuristic approach for multiplexer cost reduction consists of a number of steps. The basic idea in this case is to use large physical memories in their tall configurations as long as they give us a better multiplexer cost compared to the smaller types. The challenge is how to use the larger types more

Algorithm 6: Leftover reduction heuristic

```
Input  : $VMs_{dimensions}$, $PM\_types$, $PM\_types_{quantity}$, $PM\_types_{configuration\_set}$
Output: $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$

sort($PM\_types$);
foreach ($PM\_type \neq PM_{smallest\_type}$) do
    foreach ($VM$) do
        while ($leftover = 0$) do
            cover($VM$, $PM_{tallest\_configuration}$);                //step 1
            update($used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$);
        end
    end
    sort($VMs$, $bottom\_area_{depth}$);
    foreach ($VM$) do
        while ($leftover = 0$) do
            cover($VM$, $PM_{tallest\_possible\_configuration}$);       //step 2
            update($used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$);
        end
    end
    $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$,
    $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$, $PM\_next\_types$);
    while ($PM\_next\_type_{required\_number} < PM\_next\_types_{available\_number}$) do
        add($VMs$, $PM\_type$, "min leftover");                        //step 3
        update($used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$);
        $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$,
        $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$, $PM\_next\_types$);
    end
end
call($nVM - 1PM$);
Return $used\_PM\_types_{configuration}$, $used\_PM\_types_{disposition}$;
```

efficiently to reduce the overall multiplexer cost of the solution.

Again, we sort the physical memory types decreasingly based on their capacity, and go through the following steps for each type. Also, we try to work on all virtual memories at the same time.

Step 1: This step is exactly the same as the first step in the leftover reduction because filling empty parts with the tallest configuration of the physical memory without creating any type of leftover is the most efficient way to use a physical memory to reduce multiplexer cost.

We have to keep track of the used and available physical memories, and we have to stop this step if we have consumed all the physical memories of the current type. After filling every possible spot with this step, if we have more physical memories of this type available we will go to the next step.

Step 2: In this step we sort all the virtual memories based on the depth of the empty part in their bottom area decreasingly, and only work on the ones which the current physical memory can cover their entire bottom area depth. It means smaller types of physical memories cannot cover the entire depth of the bottom area of these virtual memories with their tallest configuration. We start from the virtual memory with the deepest bottom part, and try to fill its bottom empty area with the configuration which covers the entire depth. We put the physical memories in this configuration next to each other in the bottom area of the virtual memory as long as we do not create any horizontal leftover.

After working on each virtual memory we move to the next one as long as there are physical memories of the current type left. When we finished working on all the eligible virtual memories for this step, and we still have more physical memories of this type available we can go to the step 3.

Step 3: In this step we work on the empty parts at the right hand side of the virtual memories. These are the parts which we did not cover in the first step due to creation of horizontal leftover. We sort the virtual memories based on the width of the uncovered area on their right hand side, and start with the virtual memory with the widest area. In this step we only work with the tallest configuration of the physical memory, and do not allow creating of vertical leftover. Therefore, we will work only on the virtual memories which we worked on in the first step (the ones deeper than the depth of the tallest configuration).

After this step if the current type of physical memories is still available we can move to the step 4. Otherwise, we should move to the next type of physical memories, and start from the first step with them.

Step 4: In the second step we worked on the virtual memories which the smaller types of physical memories could not cover the entire depth of their bottom empty area with their tallest configuration. In this step we can perform the same procedure for the virtual virtual memories for whom we have a proper configuration of the current physical memory. It means that there exists a configuration for the current type with the same depth as the desirable configuration of the smaller type.

If the depth of the shortest configuration of the current physical memory is taller than the desirable configuration of the smaller type to cover the bottom area of the virtual memory, we can assume that this type is not a good choice

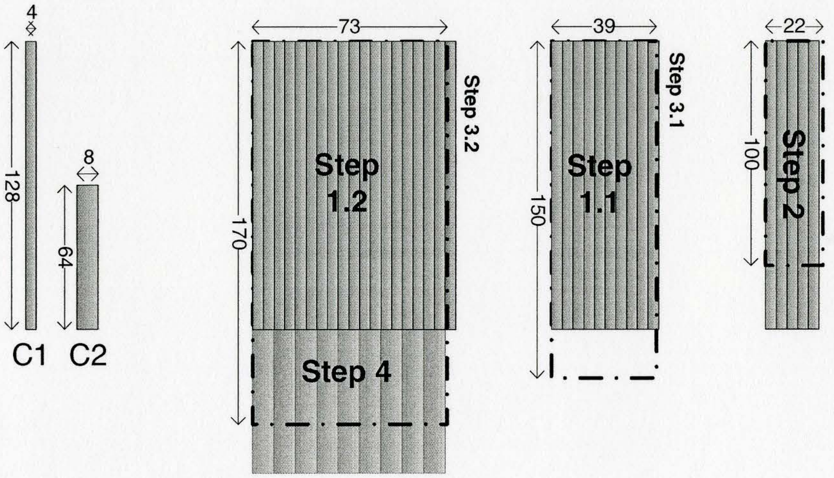


Figure 3.10: Steps 1 to 4 of the multiplexer cost heuristic algorithm

for covering the bottom area of the virtual memory. We do not allow any horizontal leftover in this step too, and as mentioned before, we sort the virtual memories based on the depth of their uncovered bottom area decreasingly.

After completion of this step if we still have the current type available we will go to the next step. Figure 3.10 shows the configuration set of a sample physical memory, and the way we use it in the first 4 steps of our heuristic approach for the 3 sample virtual memories. We have numbered the steps in such a way that it represents the priority of each virtual memory in each step. For example, step 1.1 is prior to step 1.2.

Step 5: In step 2 we did not allow the creation of horizontal leftover because it was possible that we could use the current type of the physical memories in a more efficient way to reduce the overall multiplexer cost. In this step we are willing to use the current type despite of the leftover it creates to reduce the multiplexer cost for the areas that we left in step 2 to avoid horizontal leftover. Therefore, we sort the virtual memories we worked on in step 2 based on the depth of their empty bottom area, and add the physical memory which creates the horizontal leftover to them one by one to reduce the multiplexer cost.

The spots which we worked on in this step were the only remaining spots which we could come up with a efficient usage for the current type of physical memories. From now on the use of this type does not have any advantage over smaller types in terms of the multiplexer cost reduction. Moreover, this type creates more leftover compared to the smaller types. Therefore, we prefer not to use this type in the solution anymore. However, to make this decision we need to make sure that the overall capacity of the remaining physical memory types is sufficient to cover all the remaining empty parts in all the virtual memories.

In this case we can use the estimation function which we discussed in step 2 of the leftover reduction algorithm in subsection 3.5.1 to decide whether we can go to the next type or we should continue using the current type due to lack of overall capacity of smaller physical memory types. In the case that we should continue with the current type we will move to step 6, which is discussed below.

Step 6: We are running this step because we know that we do not have sufficient smaller physical memory types to cover all the remaining uncovered parts of all virtual memories. Therefore, we have to use more instances of the current type although it would not be an ideal choice if adequate number of smaller physical memory types were available. This step works on the virtual memories which we processed in step 4, and we left their bottom-right part uncovered due to creation of horizontal leftover. We sort these virtual memories based on the depth of their uncovered bottom part, and start from the deepest one. After adding each instance of the current type to each virtual memory, we rerun the estimation function to make sure whether we should continue with the current type or not.

Step 7: In this step we try to work on the virtual memories which we did not process their bottom area during the previous steps because the depth of the area was small and suitable for the smaller types of physical memories.

First, we sort the virtual memories based on the depth of their uncovered bottom area decreasingly, and start from the virtual memory with the deepest bottom area. It is obvious that due to the small depth of the bottom area we will use the shortest configuration of the current physical memory to cover it.

It is very important to know that we do not allow creation of horizontal leftover in this step. Moreover, after adding each physical memory we rerun the estimation function to make sure that we need to continue in this step or we can move to the next type of physical memories. When we finish adding physical memories in this step, and adding any other physical memory will result in creation of horizontal leftover, and the estimation function shows that we still need to use the current type, we should move to the next step assuming that there are more physical memories left from this type. Therefore, we have to keep track of the used physical memory resources in different steps.

Step 8: The only uncovered spots left in all virtual memories in this step are the ones that we left uncovered in the last step because of the creation of the horizontal leftover. In this step we sort the virtual memories based on the area of their uncovered part in their bottom right decreasingly. We start to cover these spots with the current type of the physical memories as long as the estimation function requires us to do that, and we have physical memories from the current type.

There are 3 possibilities in this case:

- We run out of the current type, so we have to move to the next type, and ignore the result of the estimation function.
- The result of estimation function does not change during this step, so we will fill all the empty parts with the current type and complete the solution.
- The result of estimation function changes during this step, so we move to the next physical memory.

Steps 5 to 8 of the multiplexer cost reduction heuristic are presented in figure 3.11 which shows same physical and virtual memories of figure 3.10, and the way we continue our heuristic method on them.

The algorithm we presented in this section should be used for all except the last type of physical memories, and we should reuse the algorithm introduced in section 3.2 for the final type of physical memories. However, we should consider that the empty area in this case is not necessarily rectangular.

Algorithm 7 summarizes what we discussed in this section in a pseudo code format. In the first step which is captured in lines 3 to 5 we work on all virtual memories starting from the shortest ($VM_{shortest}$) to the one with the highest depth ($VM_{tallest}$). Lines 6 to 8 represent the second step which works on the virtual memories from the

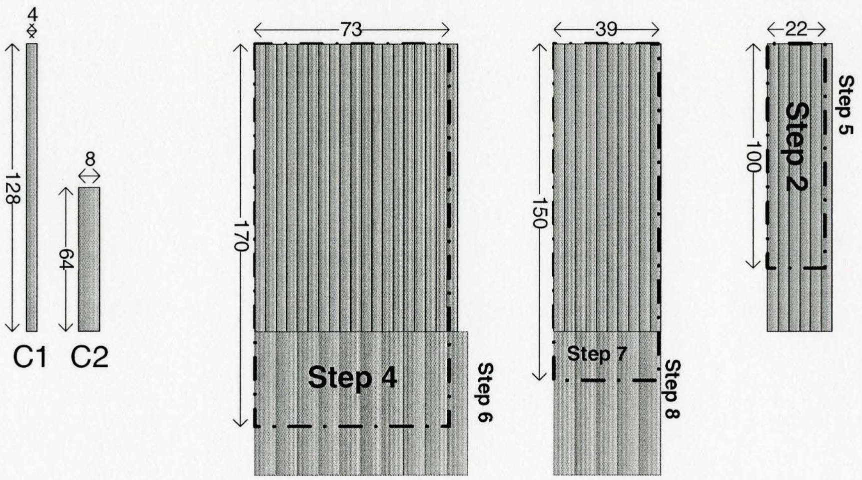


Figure 3.11: Steps 5 to 8 of the multiplexer cost heuristic algorithm

one with the deepest bottom area ($VM_{tallest\_bottom\_area}$) to the ne with the shortest bottom area ($VM_{shortest\_bottom\_area}$), and fills the bottom part of the virtual memories with the shortest configuration of the current physical memory which covers the entire depth of the bottom part ($PM_{tallest\_possible\_configuration}$) as long as it does not create any horizontal leftover (h_leftover).

The third step of the algorithm is coded in lines 9 to 11 which works on the virtual memories from the one with the widest empty right side area ($VM_{widest\_right\_area}$) to the one with the thinnest uncovered area on its right side ($VM_{thinnest\_right\_area}$). In line 12 we start the fourth step, and find the depth of the shortest configuration which can cover the entire depth of the empty part at the bottom of each virtual memory in line 13 by using the "proper_conf_finder" function, and store the result in "$VM\_proper\_configuration\_depth$". If this value is available in the depths of the configurations of the current type of physical memories ($PM\_configuration\_set_{depth}$), we use the current type in this step for the particular virtual memory.

Step 5 is captured in lines 16 to 19 which covers the bottom part of the virtual memories in the case of the depth of the bottom part is larger than the heights of the tallest configuration of the next type of physical memory $(PM\_nexttype_{tallest\_configuration})$. Line 19 represents the estimation function which we will use in the final 3 steps of the algorithm which is the same as the estimation function in the leftover algorithm. However, we have presented its arguments in a more abstract way here to shorten the pseudo code. Therefore, the "*used_PMs*" represents the type, configuration, and the disposition of the used physical memories in the solution.

Step 6 is coded in lines 22 to 26, and lines 27 to 30 are for step 7, and finally, step 8 is captured in lines 31 to 34. It should be noted that the final 3 steps of the algorithm add only one physical memory to the virtual memory which we work on each time. Also, steps 7 and 8 only work on the virtual memories which have not been processed in steps 2,4, and 5 $(VM \notin VMs\_steps2, 4, 5)$. Finally, the "$nVM - 1PM$" function is called to cover the remaining uncovered parts of virtual memories (if there is any) with the smallest type of physical memory, which is the last type.

Algorithm 7: Multiplexer cost reduction heuristic

```
Input  : $VMs_{dimensions}$, $PM\_types$, $PM\_types_{quantity}$, $PM\_types_{configuration\_set}$
Output : $used\_PMs_{type\&configuration}$, $used\_PMs_{disposition}$
sort($PM\_types$);
foreach ($PM\_type \neq PM_{smallest\_type}$) do
    foreach ($VM$ from $VM_{shortest}$ to $VM_{tallest}$) do
        while ($leftover = 0$) do
            cover($VM$, $PM_{tallest\_configuration}$);    //step 1
        end
    end
    foreach ($VM$ from $VM_{tallest\_bottom\_area}$ to $VM_{shortest\_bottom\_area}$) do
        while ($h\_leftover = 0$) & ($VM\_bottom\_area_{depth} > next\_PM\_type_{tallest\_configuration}$) do
            cover($VM$, $PM_{tallest\_possible\_configuration}$);    //step 2
        end
    end
    foreach ($VM$ from $VM_{widest\_right\_area}$ to $VM_{thinnest\_right\_area}$) do
        while ($v\_leftover = 0$) do
            cover($VM$, $PM_{tallest\_configuration}$);    //step 3
        end
    end
    foreach ($VM$ from $VM_{tallest\_bottom\_area}$ to $VM_{shortest\_bottom\_area}$) do
        $VM\_proper\_configuration\_depth$ = proper_conf_finder($VM_{bottom\_area}$, $PM\_types_{configurations}$);
        while ($h\_leftover = 0$) & ($VM\_proper\_configuration\_depth \in PM\_configuration\_set_{depth}$) do
            cover($VM$, $PM_{proper\_configuration}$);    //step 4
        end
    end
    foreach ($VM$ from $VM_{tallest\_bottom\_area}$ to $VM_{shortest\_bottom\_area}$) do
        if ($bottom\_area_{depth} > PM\_nexttype_{tallest\_configuration}$) then
            $VM\_proper\_configuration\_depth$ = proper_conf_finder($VM_{bottom\_area}$,
            $PM\_types_{configurations}$);
            cover($VM$, $PM_{tallest\_possible\_configuration}$);    //step 5
        end
    end
    $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$, $used\_PMs$, $PM\_next\_types$);
    while ($PM\_next\_type_{required\_num} < PM\_next\_types_{available\_num}$) do
        foreach ($VM$ from $VM_{tallest\_bottom\_area}$ to $VM_{shortest\_bottom\_area}$) do
            $VM\_proper\_configuration\_depth$ = proper_conf_finder($VM_{bottom\_area}$,
            $PM\_types_{configurations}$);
            while ($VM\_proper\_configuration\_depth \in PM\_configurations_{depth}$) do
                add($VM$, $PM_{proper\_configuration}$);    //step 6
                $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$, $used\_PMs$, $PM\_next\_types$);
            end
        end
        foreach ($VM \notin VMs\_steps2, 4, 5$) do
            while (No added $h\_leftover$) & $PM\_next\_type_{required\_num} < PM\_next\_types_{available\_num}$ do
                add($VM$, $PM_{tallest\_possible\_configuration}$);    //step 7
                $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$, $used\_PMs$, $PM\_next\_types$);
            end
        end
        foreach ($VM \notin VMs\_steps2, 4, 5$) do
            while ($PM\_next\_type_{required\_num} < PM\_next\_types_{available\_num}$) do
                add($VM$, $PM_{tallest\_possible\_configuration}$);    //step 8
                $PM\_next\_type_{required\_number}$ = estimate($VMs_{dimensions}$, $used\_PMs$, $PM\_next\_types$);
            end
        end
    end
end
call($nVM - 1PM$);
Return $used\_PM_{configuration}$, $used\_PM_{disposition}$;
```

Chapter 4

Experimental Results

In the previous chapter we have tried to understand the complexity of the problem by going through some intermediate steps and subproblems to discover the role of different parameters involved in the overall process. Then, we gave a method to search the entire solution space to find the optimum solutions for minimum leftover and minimum multiplexer cost. Since the runtime of the exhaustive search method was not acceptable, we introduced two different heuristic approaches to come up with suboptimal solutions for the problem to reduce leftover or multiplexer cost in an acceptable runtime.

In this chapter we will start with a case study on soft tissue modeling to observe the dimensions of the problem in a realistic environment, and we will evaluate the results of our heuristic against a manual memory binding approach for the problem. In the second section of this chapter we will provide some experimental results and a comparison between our heuristic methods and the exact solution for a set of hypothetical examples. Furthermore, we will try to analyze the results, and provide evidence for them.

4.1 Case Study

Soft tissue modeling has a vital role in virtual surgery because it gives an artificial feeling of the tissue to the surgeon. This modeling uses Finite Element Modeling (FEM) analysis and has to be done in real-time. The FEM analysis uses iterative algorithms to solve a set of equations to find the position (or force) of each node in the tissue. The most compute-intensive part in the iterative algorithms is a sparse matrix-vector multiplication which is an intermediate step in the solution generation process. The goal is to accelerate this matrix-vector multiplication to increase the overall speed of the FEM analysis, and be able to meet the real-time constrains introduced by the environment and the nature of the problem.

The memory architecture has a vital role in the overall performance and structure of the architecture. Therefore, in this case study we can use our heuristic methods and compare them with the manual solution which we originally designed for the problem. Moreover, if such a tool was available when we started this hardware accelerator project, we could have checked the applicability of several choices for the processing core of the architecture in a fraction of time.

To generate a model for a tissue, we employ a meshing algorithm first. A meshing algorithm creates a set of nodes on the surface and inside the tissue and calculates the dependencies between nodes by measuring their movements and forces with respect to each other. The meshing algorithm creates a 2-dimensional sparse matrix for the tissue, which holds all physical information such as density, hardness, and softness of each part of the tissue. This model can be processed to simulate the reflections of the tissue to an external force or displacement.

To attack the problem we first start with the properties of the sparse-matrix:

- The matrix is symmetric.
- The matrix has blocks of non-zero elements in size of 3×3.
- The meshing algorithm creates a limited number of neighbors per node to manage the condition number of the generated matrix [9, 27]. Therefore, we can assume that in this case, the number of non-zero elements in each row and column is bounded. In this particular type of matrices, the number of neighbors is limited to 18.

Assuming we have a model with 'n' nodes, the size of the matrix will be "$3n \times 3n$". In the iterative solution, we have to multiply this matrix by an intermediate vector with the size of "$3n \times 1$". We will call the matrix 'A' and the vector 'd', and the result of multiplication will be stored in vector "Ad" with the dimension of "$3n \times 1$". An acceptable number of nodes for a simple model is about 500, and a real-time model should be updated 100-1000 times per second [17].

In order to meet the real-time constraints of the problem, we should go for highly parallel solutions which leads us to use several multiplier units concurrently. FPGAs are a suitable candidate for design and development of highly parallel solutions because they have a large number of embedded multipliers and embedded physical memory units to store data on chip.

For the purpose of this project we used a Stratix II EP2S60 FPGA with the specifications given in table 4.1.

We are aiming to fit 512 nodes in this FPGA. Therefore, the size of the matrix 'A'

Table 4.1: Specifications of Stratix II EP2S60 FPGA [1].

Feature	M512 RAM Block	M4K RAM Block	M-RAM Block	Embedded 18 × 18 Multipliers
Quantity	329	255	2	144
Configurations (depth × width)	512 × 1 256 × 2 128 × 4 64 × 8 64 × 9 32 × 16 32 × 18	4K × 1 2K × 2 1K × 4 512 × 8 512 × 9 256 × 16 256 × 18 128 × 32 128 × 36	64K × 8 64K × 9 32K × 16 32K × 18 16K × 32 16K × 36 8K × 64 8K × 72 4K × 128 4K × 144	9 × 9 18 × 18 36 × 36

is "1536 × 1536". The maximum number of non-zero elements will be the maximum number of neighbors for each node (18) times the number of nodes (512) times the number of non-zero elements for each neighborhood between 2 nodes (9), which is "$18 \times 512 \times 9 = 82944$". This can be translated to the maximum number of non-zero elements per row/column of $82944 \div 1536 = 54$. Also, the bit width of the matrix elements is 12, and vector elements are in 16 bits. The result of multiplication is stored in 16 bits.

The solution which we want to work on processes the neighbors of each node in 2 consecutive clock cycles. Therefore, it has to process 9 neighbors in each clock cycle, and we know that each neighborhood is a set of 9 numbers. Therefore, we need 81 multipliers in parallel to calculate the result of the matrix-vector multiplication, and the whole process of multiplication will take 1024 clock cycles.

We have 9 multiplication clusters, and each cluster is in charge of processing one neighbor of the current node. Therefore, there are 9 multipliers in each cluster. Figure

4.1 shows a multiplication cluster, and the way we feed the values of matrix 'A' and vector 'd' into it. As presented in this figure, there are 9 distinct values of matrix 'A' coming into the cluster and each one is fed to a multiplier. However, we have only 3 distinct vector 'd' values going to each cluster, and each value is connected to 3 multipliers because those multipliers are in the same column, and need the same operand from the vector.

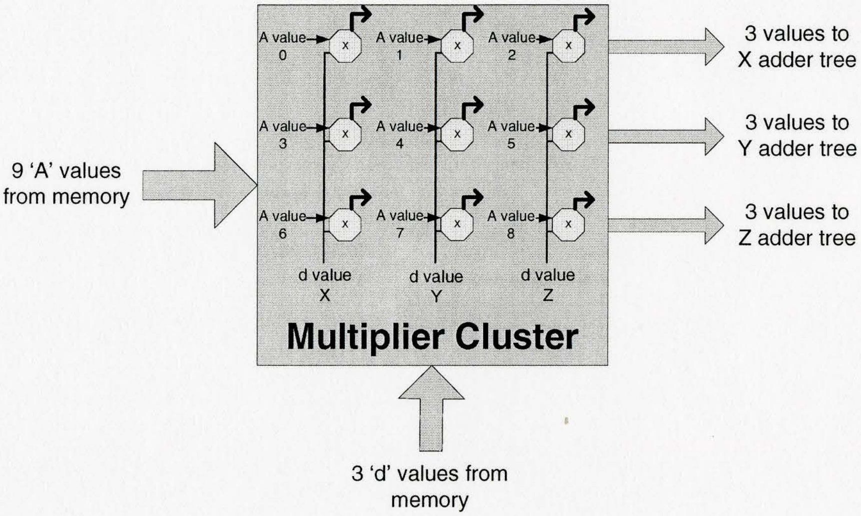


Figure 4.1: A cluster of multipliers.

To process 9 nodes concurrently, all the clusters should work in parallel, and their outputs should be connected to 3 different adders. Each adder is in charge of summing one row of the multiplication results. As each node (3 rows) is processed in 2 clock cycles, the number of adder inputs should be 28, where 27 are for the values of the current cycle and 1 for the sum in previous clock cycle which we use every other clock cycle. Figure 4.2 shows the way we connect the group of clusters together and to the adder.

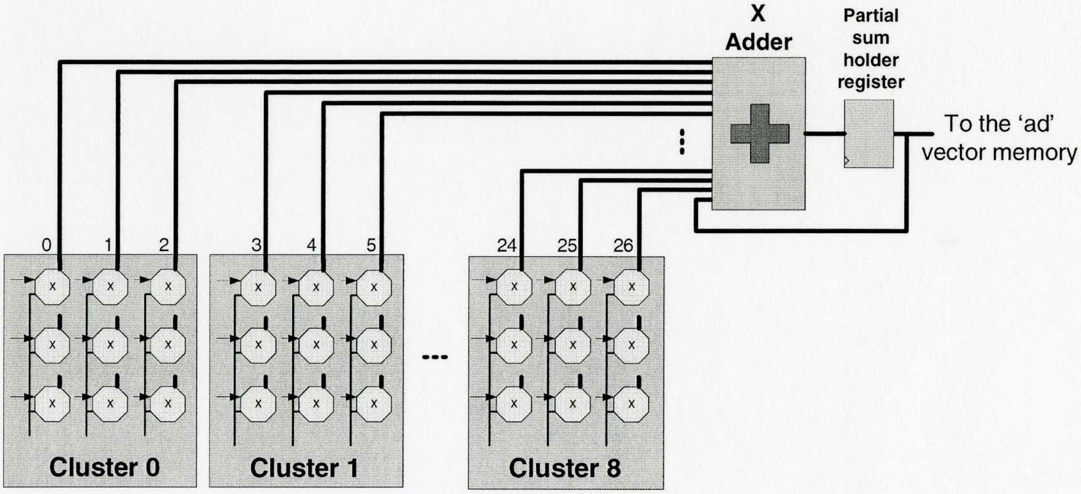


Figure 4.2: Connection of the clusters to the X adder (Y and Z adders are not shown. They are connected to the second and third row of the multipliers in clusters).

After designing the processing core, we have to design and develop the memory architecture around this unit. In this step we can take advantage of the sparsity of matrix 'A' and store only non-zero elements of the matrix. However, this comes with the cost of keeping track of the position of non-zero elements because we need their position to perform the multiplication properly.

We have a maximum of 18 neighbors for each node, and each neighborhood implies a 3 × 3 non-zero block. Therefore, we can assume that every 3 consecutive rows of the matrix have the same placement for non-zero element values and each non-zero element is followed by two other ones in a row. Therefore, we can read 3 consecutive rows of matrix 'A' at the same time and do the multiplication. As we are only storing non-zero elements of the matrix, we should save the index of the non-zero block we want to work on because we have to find the relative spot in the vector 'd' as the second argument of the multiplication. We have 512 nodes, so we need 9 bits of address to look up the respective value for the current elements of the matrix in the vector.

We have 9 clusters, and each cluster should receive 9 values from matrix 'A' every clock cycle. As we have 512 nodes, we need 9 bits of index for each cluster which helps us to look up the respective vector 'd' values for each cluster. Therefore, we need $9 \times 9 \times 12 = 972$ bits to feed the 'A' values to the cluster and $9 \times 9 = 81$ bits as the address for looking up proper value in vector 'd'. Figure 4.3 shows the storage strategy we use to store the non-zero values of matrix 'A' and their indices which are used as addresses to lookup vector 'd' elements (as the second operand of the multiplications).

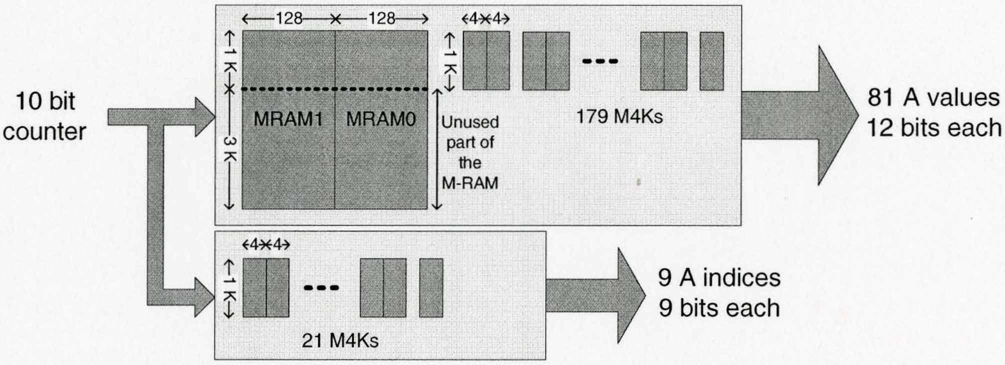


Figure 4.3: Memory architecture for storing matrix 'A'.

This memory architecture is accessed linearly from address 0 to 1023 (each 2 addresses are for the neighbors of 1 node) because we do the multiplication row wise, unlike the memory architecture for vector 'd' which is accessed randomly. As we always work with 3 consecutive rows of matrix 'A', we always have to lookup 3 consecutive values of vector 'd' which we call 'x', 'y', and 'z'. Therefore, we store the vector in such a form that every address holds 3 consecutive values, so by each access we will be able to retrieve all values at once. Figure 4.4 shows how we store vector '*d*'.

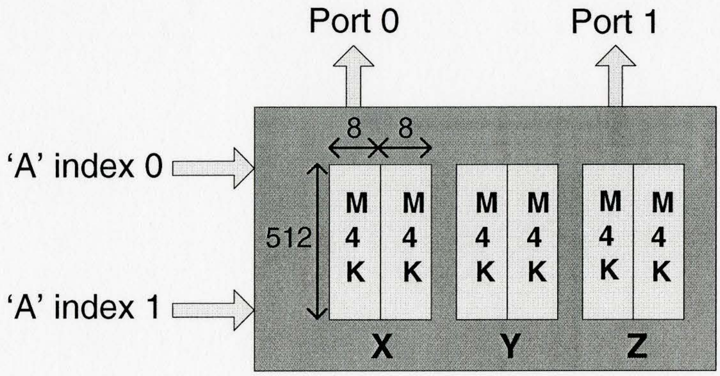


Figure 4.4: Memory architecture for storing vector '*d*'.

All the memory elements in the Stratix II FPGA we use are dual port. Therefore, each storage of vector '*d*' can provide 2 values in every clock cycle. To feed all the clusters we need to have 5 copies of the vector '*d*' storage architecture, and each copy can provide inputs to 2 clusters. As illustrated in figure 4.5, the address inputs of the vector '*d*' storage are provided by the index outputs of the matrix '*A*'.

The multiplication results are stored in vector '*Ad*' which has a similar architecture to vector '*d*'. We should write the outputs of the 3 adders every other clock cycle in this vector. The address is provided by using the value of the same counter we use to read the matrix values. Figure 4.6 shows the architecture for the storage and the input signals for vector '*Ad*'.

Due to the level of complexity of this case study, the exhaustive search solution takes several weeks to finish. Therefore, we will compare our heuristic methods to

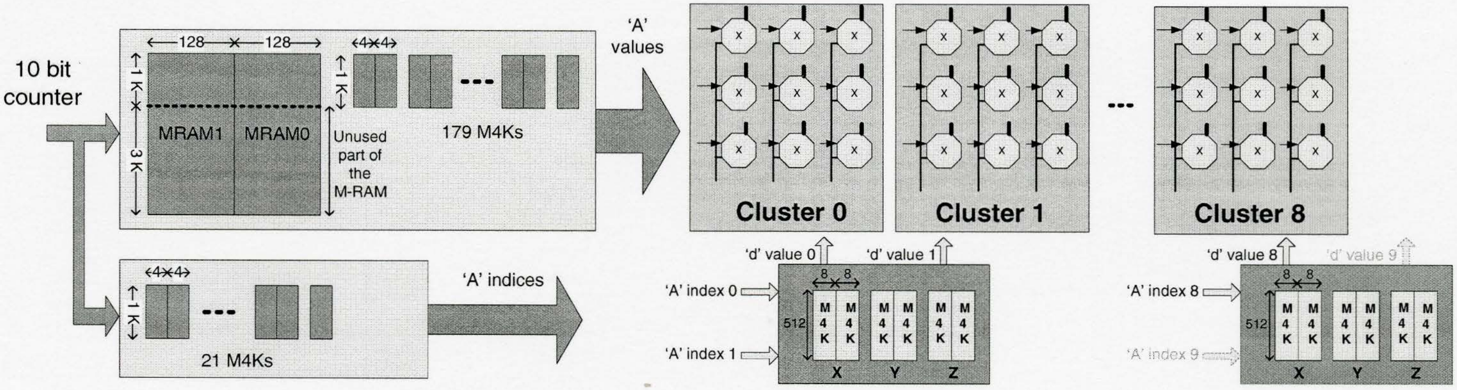


Figure 4.5: The overview of the architecture.

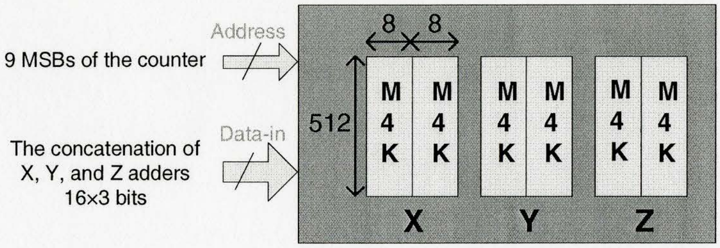


Figure 4.6: Memory architecture for storing vector '*Ad*'.

a manual solution which we designed for this problem before performing this research. Table 4.2 presents the comparison between the 2 cases, and obviously using the heuristic approaches saves us a tremendous amount of time, and it gives more efficient results in this case.

Table 4.2: Heuristic algorithm and a manual solution comparison.

Solution	Runtime	Multiplexer cost	Leftover	Used M-RAMs	Used M4Ks	Used M512s
Manual	48 hrs	0	789504 bits	2	236	0
Heuristic	8 s	0	396288 bits	1	255	104

In the next section we will provide some experimental results in both of the heuristic and exhaustive solution space search approaches, and will analyze them to illustrate the weaknesses of our heuristic approach. The results also will easily convince us that the exhaustive solution space search method is not a practical approach to solve the problem for complex real-world applications.

4.2 Experimental Results

In this section we report the results of running the heuristic algorithms and the exhaustive solution search method for a number of experiments with different sets of virtual memories and physical resources. Each set of virtual memories consists of 2 to 3 virtual memories selected from a pool of memories. Table 4.3 shows the sets used in the experiments along with the dimensions of each memory.

Table 4.3: Virtual memory sets.

Virtual memory set	**Virtual memories**	**Dimensions**
V1	VML1 VMXS	4500 × 7 100 × 20
V2	VML1 VMS2	4500 × 7 700 × 90
V3	VMM2 VMS1	1500 × 30 700 × 15
V4	VMM1 VMS2	1500 × 11 700 × 90
V5	VML1 VMM1 VMS2	4500 × 7 1500 × 11 700 × 90
V6	VML1 VMM2 VMS1	4500 × 7 1500 × 30 700 × 15
V7	VML1 VMS2 VMXS	4500 × 7 700 × 90 100 × 20

We use two types of physical memories with different quantities and configuration sets for each experiment. Tables 4.4 and 4.5 show the different configuration sets of the physical memory types that we use in the experiments.

Table 4.4: Configuration sets of the first physical memory type (M8K).

2 configurations for M8K	**4 configurations for M8K**	**6 configurations for M8K**
1024 × 8 512 × 16	2048 × 4 1024 × 8 512 × 16 256 × 32	8192 × 1 4096 × 2 2048 × 4 1024 × 8 512 × 16 256 × 32

Table 4.5: Configuration sets of the second physical memory type (M512).

2 configurations for M512	**5 configurations for M512**
64 × 8 32 × 16	512 × 1 256 × 2 128 × 4 64 × 8 32 × 16

The number of small physical memory type (M512) in all experiments is 500, and the number for the large type (M8K) is 5 or 10. The experiments are named with the following format:

V[*set number*]_ N[*quantity of large type*]_ C[*number of configurations for large type, small type*]

For example, V1_ N10_ C4,5 means: using the first set of virtual memories in table 4.3 and 10 physical memories of the first type which has 4 configurations (table 4.4), and 500 physical memories of the small type with 5 configurations (table 4.5). With this explanations, we can now proceed to presenting the tables of the results.

Table 4.6: Experiments on V1 with 2 configurations for M512.

	Exhaustive search						**Heuristic**					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V1_N5_C2,2	32	12068	5924	15.03	48	0	32	12068	12068	26.48	32	0
V1_N10_C2,2	21	23844	5924	15.03	48	0	21	23844	12068	26.48	445	0
V1_N5_C4,2	27	10532	5924	15.03	48	0	27	10532	10532	23.92	27	0
V1_N10_C4,2	7	15652	5924	15.03	48	0	7	15652	15652	31.84	7	0
V1_N5_C6,2	18	11044	5924	15.03	38	0	101	30500	30500	47.66	101	0
V1_N10_C6,2	0	32036	5924	15.03	38	1	0	32036	30500	47.66	101	0

Table 4.6 presents the results of experiments on the first set of virtual memories. These experiments are all done with 2 configurations for the small physical memory type, and the effect of increasing the number of available physical memories in large types and increasing the number of their configurations is obvious.

In the case of the exhaustive search solution, the results show that increasing the quantity of the large physical memory type results in reducing the multiplexer cost which makes sense because large types can cover a larger depth of the virtual memory. Also, when we have more options for configuration of the large physical memory types we can use tall configurations to reduce the multiplexer cost, which is apparent in table 4.6. When we have an adequate number of small physical memory types, the leftover of the solution is defined by these small types. Therefore, changes in the quantity and the number of configurations of the large type does not affect the leftover of the design. However, when we have more options for the large types (quantity and configuration set) we may be able to reduce the multiplexer cost of the minimum leftover solution, which is the case in the last 2 rows of the table.

The heuristic approach results are close to the optimum in some cases. However, it is obvious that increasing the number of options does not guarantee better results for the heuristic algorithm. Moreover, the leftover overhead in the case of using the heuristic algorithm in this set of experiments is about 10-30% because of an assumption in the heuristic algorithm which is not always correct. We have assumed that small types of physical memories are more suitable in the solutions which work on reducing the leftover simply because we should be able to cover small uncovered parts with them with less implied leftover. However, this is dependent on the configurations of the small types. As an example we can refer to the case when the thinnest configuration of the small type is wider than some configurations of the large type. This is the exact case that we have in the set of experiments in table 4.6. In this set of experiments, the heuristic leaves the right end part of the virtual memories for the small type, and does not use the large type for them. However, the large type has thinner configurations than the small type, and is more suitable to reduce the horizontal leftover.

Table 4.7 contains the results of the same experiments with 5 configurations for

Table 4.7: Experiments on V1 with 5 configurations for M512.

	Exhaustive search						**Heuristic**					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V1_N5_C2,5	21	2340	1316	3.78	28	0	21	5412	5412	13.91	21	0
V1_N10_C2,5	21	2340	1316	3.78	28	0	21	5412	5412	13.91	21	0
V1_N5_C4,5	7	5412	1316	3.78	16	0	7	5412	1316	3.78	16	0
V1_N10_C4,5	7	5412	1316	3.78	16	1	7	5412	1316	3.78	16	0
V1_N5_C6,5	4	12068	1316	3.78	10	3	8	19236	1316	3.78	10	0
V1_N10_C6,5	0	26404	1316	3.78	10	7	0	26404	1316	3.78	10	0

the small type of physical memories. The first notable issue is that the leftover in both cases of exhaustive search and heuristic algorithm is decreased. The reason is that some of the extra configurations in this case for the small type are taller than what we had in previous experiments. Therefore, we are able to reduce the horizontal leftover more effectively with the current set of configurations. Moreover, the heuristic algorithm works much better with the current set of configurations for the small type because the current set of configurations holds the property we have assumed in our heuristic approach. Therefore, when we leave a part of virtual memory to be filled with a smaller type of physical memory, we can be confident that the smaller type is more suitable than the current type for this relatively small part.

The first half of table 4.8 contains the results for experiments on the second set of virtual memories (V2) using 2 configurations for the small type of physical memories, and the second half is for the case with 5 configurations for the small type. Obviously, the increase in the number of types of the small physical memory type makes a visible growth in the runtime of the exhaustive search method because of the added number of options. The other observation we can have from the results is that the number

Table 4.8: Experiments on V2.

	Exhaustive search						Heuristic					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V2_N5_C2,2	340	9436	9436	9.079	340	0	561	9436	9436	9.079	561	0
V2_N10_C2,2	116	11996	9436	9.079	118	0	481	35036	35036	27.05	481	0
V2_N5_C4,2	340	9436	9436	9.079	340	1	421	25820	25820	21.46	421	0
V2_N10_C4,2	116	11996	9436	9.079	118	14	309	46300	29916	24.05	325	0
V2_N5_C6,2	330	9436	9436	9.079	330	4	345	34012	34012	26.47	345	0
V2_N10_C6,2	106	11996	9436	9.079	108	336	201	38108	38108	28.74	201	0
V2_N5_C2,5	78	14044	1244	1.299	118	17	78	14044	1244	1.299	118	0
V2_N10_C2,5	38	26844	1244	1.299	118	41	38	26844	26844	22.12	38	0
V2_N5_C4,5	78	14044	1244	1.299	106	2139	82	8924	1244	1.299	106	0
V2_N10_C4,5	38	26844	1244	1.299	106	17353	42	21724	5340	5.349	106	0
V2_N5_C6,5	78	14044	1244	1.299	100	8842	98	19164	1244	1.299	100	0
V2_N10_C6,5							66	34012	5340	5.349	100	0

of configurations and the quantity of memories have a direct effect on the run-time of the exhaustive search algorithm. However, the heuristic method does not seem to be affected with this parameters. Tables 4.9 and 4.10 show the results of applying both solution methods on the third and forth set of virtual memories (V3 and V4) respectively.

So far we have reported the results of the algorithms on the virtual memory sets with 2 virtual memories. The following 3 tables (4.11, 4.12, and 4.13) contain results for the virtual memory sets with 3 virtual memories (V5, V6, and V7).

As presented in these tables, the number of virtual memories has a direct effect on the run-time of the exhaustive solution search method. Also, there is a visible growth in the run-time of the heuristic method when the number of virtual memories and the number of configurations of the small type of physical memories are increased. However, compared to the run-time of the exact solution method, the results of the

Table 4.9: Experiments on V3.

	Exhaustive search						Heuristic					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V3_N5_C2,2	132	4404	3892	6.553	135	0	172	6964	4404	7.352	204	0
V3_N10_C2,2	30	10036	3892	6.553	135	1	30	10036	4404	7.352	204	0
V3_N5_C4,2	132	4404	3892	6.553	135	0	220	20788	3892	6.553	252	0
V3_N10_C4,2	0	26420	3892	6.553	105	0	0	26420	3892	6.553	252	0
V3_N5_C6,2	132	4404	3892	6.553	135	0	220	20788	3892	6.553	252	0
V3_N10_C6,2	0	26420	3892	6.553	105	4	0	26420	3892	6.553	252	0
V3_N5_C2,5	30	6964	820	1.456	77	15	37	5940	1332	2.344	45	1
V3_N10_C2,5	30	6964	820	1.456	77	18	30	9012	1332	2.344	45	1
V3_N5_C4,5	18	13108	820	1.456	77	217	25	11572	820	1.456	77	1
V3_N10_C4,5	0	26420	820	1.456	77	471	0	26420	820	1.456	77	1
V3_N5_C6,5	18	13108	820	1.456	77	1155	25	11572	820	1.456	77	1
V3_N10_C6,5	0	26420	820	1.456	77	7232	0	26420	820	1.456	77	1

Table 4.10: Experiments on V4.

	Exhaustive search						Heuristic					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V4_N5_C2,2	248	12148	12148	13.26	248	0	248	12148	12148	13.26	248	0
V4_N10_C2,2	83	20340	12148	13.26	134	0	152	35188	32628	29.1	200	0
V4_N5_C4,2	244	8564	8564	9.725	244	0	248	12148	12148	13.26	248	0
V4_N10_C4,2	72	20340	8564	9.725	118	3	152	37236	16244	16.97	200	0
V4_N5_C6,2	244	8564	8564	9.725	244	2	248	12148	12148	13.26	248	0
V4_N10_C6,2	72	20340	8564	9.725	118	59	152	37236	16244	16.97	200	0
V4_N5_C2,5	61	13684	884	1.1	101	50	69	11124	884	1.1	101	1
V4_N10_C2,5	21	26484	884	1.1	101	96	29	23924	21364	21.18	101	1
V4_N5_C4,5	61	13684	884	1.1	101	3373	69	12660	884	1.1	101	1
V4_N10_C4,5	21	26484	884	1.1	101	12704	29	25460	4980	5.895	101	1
V4_N5_C6,5	61	13684	884	1.1	101	18222	69	12660	884	1.1	101	1
V4_N10_C6,5							29	25460	4980	5.895	101	1

Table 4.11: Experiments on V5.

	Exhaustive search						**Heuristic**					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V5_N5_C2,2	468	17000	17000	13.28	468	0	689	17000	17000	13.28	689	0
V5_N10_C2,2	228	17000	17000	13.28	228	1	593	40040	37480	25.24	641	0
V5_N5_C4,2	468	17000	13426	10.79	500	4	549	33384	33384	23.12	549	0
V5_N10_C4,2	224	13416	13416	10.78	224	184	405	37480	37480	25.24	405	0
V5_N5_C6,2	458	17000	13416	10.78	490	25	473	41576	41576	27.25	473	0
V5_N10_C6,2	214	13416	13416	10.78	214	6140	281	43624	43624	28.21	281	0
V5_N5_C2,5	89	14440	1640	1.456	129	6774	97	11880	1640	1.456	129	19
V5_N10_C2,5	49	27240	1640	1.456	129	25966	57	24680	22120	16.62	129	21
V5_N5_C4,5							101	8296	1640	1.456	117	12
V5_N10_C4,5							61	21096	5736	4.914	117	21
V5_N5_C6,5							109	19560	1640	1.456	111	16
V5_N10_C6,5							85	33384	3688	3.216	111	21

Table 4.12: Experiments on V6.

	Exhaustive search						**Heuristic**					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V6_N5_C2,2	456	8744	8744	9.133	456	0	613	11816	9256	9.616	645	0
V6_N10_C2,2	115	9256	8744	9.133	163	0	128	11816	9256	9.616	645	0
V6_N5_C4,2	442	16936	8744	9.133	449	1	457	25128	25128	22.41	457	0
V6_N10_C4,2	115	9256	8744	9.133	113	26	265	49192	25128	22.41	457	0
V6_N5_C6,2	412	33320	8744	9.133	446	5	429	33320	33320	27.69	429	0
V6_N10_C6,2	105	9256	8744	9.133	123	937	333	33320	33320	27.69	333	0
V6_N5_C2,5	51	8744	1576	1.779	105	2677	58	5672	2088	2.344	73	18
V6_N10_C2,5	51	8744	1576	1.779	105	5576	58	10792	2088	2.344	73	17
V6_N5_C4,5	42	9768	1576	1.779	93	65396	49	8232	1576	1.779	93	22
V6_N10_C4,5							25	18984	1576	1.779	93	21
V6_N5_C6,5							53	20008	1576	1.779	87	16
V6_N10_C6,5							33	33320	1576	1.779	87	18

Table 4.13: Experiments on V7.

	Exhaustive search						**Heuristic**					
	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)	Min Mux (LB)	Lo of min Mux (bit)	Min Lo (bit)	Min Lo %	Mux of min Lo (bit)	Run time (s)
V7_N5_C2,2	360	10508	10508	9.82	360	1	581	10508	10508	9.82	581	0
V7_N10_C2,2	136	13068	10508	9.82	138	2	501	36108	36108	27.23	501	0
V7_N5_C4,2	360	10508	10508	9.82	360	7	441	26892	26892	21.79	441	0
V7_N10_C4,2	136	13068	10508	9.82	138	152	329	47372	30988	24.31	345	0
V7_N5_C6,2	350	10508	10508	9.82	350	43	365	35084	35084	26.66	365	0
V7_N10_C6,2	126	13068	10508	9.82	128	4872	221	39180	39180	28.88	221	0
V7_N5_C2,5	78	14604	1804	1.835	118	458	78	14604	1804	1.835	118	3
V7_N10_C2,5	38	27404	1804	1.835	118	1352	38	27404	27404	22.12	38	2
V7_N5_C4,5	78	14604	1804	1.835	106	54083	82	9484	1804	1.835	106	3
V7_N10_C4,5							42	22284	5900	5.762	106	3
V7_N5_C6,5							98	19724	1804	1.835	100	2
V7_N10_C6,5							66	34572	5900	5.762	100	3

heuristic method are acceptable regarding its run-time.

The number of configurations of the smallest type of physical memories affects the run-time of the heuristic method because we have reused the algorithm in section 3.2 in our heuristic method. Therefore, the heuristic algorithm applies a branching method in the binding of the smallest physical memory type to generate an acceptable coverage.

Chapter 5

Conclusion

In this thesis, we have provided a solution for the physical (embedded) memory binding in FPGAs, which is an emerging problem in hardware due to the significant increase in the amount of embedded memories in FPGAs. Moreover, today's FPGAs offer their embedded memories in a variety of capacities and a set of configurations. Therefore, it is essential to automate the process of physical memory binding in FPGAs to enable the optimization of the required resources (embedded memories and logic cells) to bind a data memory in a FPGA, besides saving engineering time.

The introduced CAD tool is able to manage the complexity of a problem with several data memories and different types of embedded memories with different capacities and configuration sets. At the beginning, we designed and developed an exact solution algorithm using a branching strategy to search the entire solution space of the problem. The run-time of our exact algorithm was not acceptable, so we developed a heuristic method with an acceptable run-time. However, the optimality of the calculated solution was not guaranteed anymore.

The experimental results show that the in many cases the heuristic approach achieves an acceptable solution compared to the exact solution of the problem in a fraction of the run-time of the exact method. Moreover, run-time aside, in some cases it can outperform a manual design due to its resource usage.

5.1 Future Work

The first part which requires more study and research in this project is the set of decisions in the heuristic approach, which can be made more efficient. The increase of the performance of the heuristic approach is one of the most important open problems in the way of making this research more applicable.

As illustrated in the case study in section 4.1, such a tool which automates the memory architecture design for FPGAs, can be useful in the CAD tools for special cases of architectural synthesis, such as FEM analysis accelerators. This is because the CAD tool can calculate the required amount of on-chip memory elements for each candidate architecture and thereby tune the amount of parallelism and the size of processing cores with respect to the available memory resources. This a direction that clearly justifies the importance of such a tool, and requires more research.

Also, we can enable the algorithm to work with irregular configurations of the embedded memories too. However, the use of irregular configurations should be optional because they reduce the reliability of the design due to use of the parity bits as data bits. Moreover, the tool can become more advanced if it takes the number of ports of each embedded memory and the types of ports into account, and decide on the type of required embedded memories for each data memory based on the number and the type of ports as well.

In the current solution we map each virtual memory to a number of physical memories. However, in some cases it is possible to map two or more virtual memories to one physical memory, assuming that the physical memory is large enough to accommodate all virtual memories and more importantly the virtual memories are not accessed at the same time more than the number of ports of the physical memory. Therefore, we have to deal with dependencies in the high level algorithm which we want to implement in hardware. We can even make the problem more general by mapping 'n' virtual memories to 'm' physical memories.

Finally, we can consider the timing properties of the physical memories to support working with memories with different access times, so the tool will be able to handle working with different types of physical memories with different access times at the same time, and account for various clock cycles to access different memories. This can improve the overall clock speed of the design and consequently improve the performance of the design. Moreover, by having this ability, we can expand the tool to work on off-chip memories as well.

Bibliography

[1] Altera-Co. Url = www.altera.com, 2008. [Online; accessed 1-April-2008].

[2] C. Batten. Url = http://www.cag.lcs.mit.edu/ cbatten/6371/, 2002. [Online; accessed 1-April-2008].

[3] L. Benini and G. DeMichelli. A survey of boolean matching techniques for library binding. *ACM Transaction on Design Automation of Electronic Systems*, 2(3):193–226, July 1997.

[4] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. pages 236–243. International Conference on Computer Aided Design, 1995.

[5] D. Buell and T. El-Ghazawi. High-performance reconfigurable computing. *Computer*, March 2007.

[6] A. Chowdhary and J. P. Hayes. Technology mapping for field-programmable gate arrays using integer programming. pages 346–352. International Conference on Computer Aided Design, 1995.

[7] J. Cong and Y. Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Transaction on Design Automation of Electronic Systems*, 1(2):145–204, April 1996.

[8] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1st edition, 1994.

[9] Q. Du, Z. Huang, and D. Wang. Mesh and solver co-adaptation in finite element methods for anisotropic problems. *Wiley InterScience*, 21(4):859–874, March 2005.

[10] M. R. Garey and R. L. Graham. An analysis of some packing algorithms. pages 39–47. Combinatorial Algorithms, 1973.

[11] P. K. Jha and N. D. Dutt. Library mapping for memories. pages 288–292. European Design and Test Conference, 1997.

[12] K. S. Khouri, G. Lakshminarayana, and N. K. Jha. Memory binding for performance optimization of control-flow intensive behavioral descriptions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(5):513–524, May 2005.

[13] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. Automated data dependency size estimation with a partially fixed execution ordering. pages 44–50. International Conference on Computer Aided Design, 2000.

[14] C. Kulkarni, F. Catthoor, and H. DeMan. Cache optimization for multimedia compilation on embedded processors for low power. *ACM/IEEE Proc. Parallel Processing Symp. (IPPS)*, pages 292–297, April 1998.

[15] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. de Man. Cache conscious data layout organization for embedded multimedia applications. pages 686–693. Design, Automation, and Test in Europe, 2001.

[16] S. Leibson. *Designing SOCs with Configured Cores*, chapter 1. Morgan Kaufmann, 1st edition, 2006.

[17] R. Mafi, S. Sirouspour, B. Moody, B. Mahdavikhah, K. G.Elizeh, A. Kinsman, N. Nicolici, M. F. Ghiam, and M. Dan. Hardware-based parallel computing for real-time haptic rendering of deformable objects. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008.

[18] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons Inc, revised edition, 1990.

[19] S. Megadrive. Url = http://arcade.ym2149.com/megadrive/index.html, 2008. [Online; accessed 1-April-2008].

[20] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[21] S. Muroga. *The VLSI Handbook*, volume 38, chapter 46. CRC Press and IEEE Press, 2nd edition, 2006.

[22] I. Ouaiss and R. Vemuri. Hierarchical memory mapping during synthesis in fpga-based reconfigurable computers. pages 650–657. Design, Automation, and Test in Europe, 2001.

[23] P. R. Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. DeGreef. Data memory organization and optimizations in application-specific systems. *IEEE Design and Test*, 18(3):56–68, May 2001.

[24] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits.* Prentice-Hall, 2nd edition, 2003.

[25] L. Ramachandran, D. D. Gajski, and V. Chaiyakul. An algorithm for array variable clustering. pages 262–266. European Design and Test Conference (EDAC), March 1994.

[26] H. Schmit and D. E. Thomas. Synthesis of application-specific memory designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):101–111, March 1997.

[27] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CS-94-125, Carnegie Mellon University, 1994.

[28] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, and H. de Man. Flow graph balancing for minimizing the required memory bandwidth. pages 127–132. International Symposium on Systems Synthesis, November 1996.

[29] Xilinx-Co. Url = www.xilinx.com, 2008. [Online; accessed 1-April-2008].

[30] C.-H. Yang, C.-C. Tsai, J.-M. Ho, and S.-J. Chen. Hmap: a fast mapper for epgas using extended gbdd hash tables. *ACM Transactions on Design Automation of Electronic Systems*, 2(2):135–150, April 1997.

Index