UNIVERSITY OF CINCINNATI

9th APRIL , 20 01

I, AMIT KASAT **,**

hereby submit this as part of the requirements for the degree of:

MASTER OF SCIENCE

in:

COMPUTER ENGINEERING

It is entitled:

Memory Synthesis for FPGA-Based Reconfigurable Computers

Approved by:

Dr. Ranga Vemuri

Dr. Harold W. Carter

Dr. Karen Tomko

Memory Synthesis for FPGA-Based Reconfigurable Computers

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering

April 2001

by

Amit Kasat

B.E. (Electronics and Instrumentation Engineering)
Devi Ahilya University, Indore, India
June 1998

Thesis Advisor and Committee Chair: Dr. Ranga Vemuri

Abstract

A design executing on Reconfigurable Computer (RC) typically reads from and writes to physical memories on the RC. For data intensive applications like Digital Signal Processing (DSP), Image Processing, Pattern Recognition, etc. memory reads and writes constitute a large portion of the total design execution time.

With the advent of on-chip memories available on various FPGA devices, a complete hierarchy of physical memories is now available on a RC. Different types of memories provide different access latencies, storage capacities, multiple ports etc. An intelligent usage of these memories can lead to significant improvement in the read/write latency of the design. Most automated synthesis tools targeted for RCs do a trivial form of memory mapping, which does not make use of this memory hierarchy. In order to exploit the memory hierarchy, more sophisticated logic partitioning and memory mapping tools are required.

This thesis presents an automated memory mapping methodology during high level synthesis flow. By memory mapping, we mean performing a detailed assignment of various data structures, which are part of the design, to the physical memories available on the RC. We use *Tabu Search* meta-heuristic to find a good mapping for various logical memories of the design onto physical memories available on the RC. We present a heuristic, called *Rectangle Carving*, to map a single logical memory onto the RC. Tabu search calls this heuristic at every iteration to get new solutions. To ensure correct functionality for the memory mapping, additional control logic is required. This logic is used to resolve potential memory access conflicts, and to make the details of memory mapping transparent to the accessing logic, thus keeping the implementation of the logic independent of memory mapping.

Quality of memory mapping is closely related to the way logic partitioning is done on the board. We present an integrated methodology to perform both logic partitioning and memory mapping together. A tabu search formulation is used to do the task. This helps in getting good overall design mapping in very little time.

The execution time of the tool on benchmark examples is found to be very small. For design containing 100 logical memories, the stand alone memory mapper took less than 150 seconds. The heuristic is produces results within 3.5% of the near optimal results produced by the ILP-approach. The spatial partitioner took less than 800 seconds for designs having 100 compute tasks and 100 logical memories.

I dedicate this work to my parents

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Ranga Vemuri. Working with him was a great learning opportunity for me. His guidance, suggestions and encouragement was the driving force for this work. I

I am greatful to Dr. Harold Carter and Dr. Karen Tomko for giving their valuable suggestions and comments on my thesis work and for taking time off their busy schedule for the thesis defese.

This research is supported in part by US Air Force, Wright Labs, WPAFB, under contract number F33615-97-C-1043. I thank Mr. Al Scarpelli for providing valuable suggestions. The comments and suggestions provided by Ms. Kerry Hill and Mr. Darrell Barker were very helpful.

I would like to extend special thanks to Iyad. I enjoyed and learnt a lot while working in close coordination with him. He was always present for help and suggestions. I recall the long and intense discussions with him at various times of my stay here. I have fond memories of the great time I had with Srini here. I can not imagine spending last year and a half without him being here. He was a constant source of help and support. I would like to thank Sree for her constructive suggestions and motivation and helping me sail through the tough times here. I would also like to thank Rajesh for his help and suggestions.

It was great to work with Madhu and Siva for the review meeting. It was nice to have friends like Vijay, Sairavi, Jawad and Manish. I would also like to thank Ela, Karam, Margret, and other *DDELites* for their help.

I am thankful to my roommates Mahesh and Swaroop to bear with me during the last few months of my stay here. I also recall the great time I had with Sid, Sathya and Neema. I am lucky to have made lots of friends in Cincinnati with whom I had a great time.

Last but not the least, I would like to express my gratitude to my parents for encouraging me with my endeavors and for being immensely patient and supportive towards me during my stay here. Without them, it would not have been possible for me to make it so far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reconfigurable devices like *Field Programmable Gate Arrays (FPGAs)* have been the focus of attention because of the quick design turn-around time they allow. Most designers utilize *FPGAs* as a platform for prototyping of designs. For prototypes, the focus is on functionality rather than on performance. However, with the increasing pressure on time to market, and the tremendous increase in the density and complexity of FPGA devices available now, they have become a viable contender for being used in the final design itself. Today, FPGAs provide more than 3 million device equivalent logic capacity.

A *Reconfigurable Computer (RC)* is a hardware platform which can be reused by configuring the device for new design. This is possible by the virtue of its programmable devices. A RC comes ready to use, with the complete design environment. A wide variety of commercially available tools targeting FPGAs provide the complete design flow right from synthesis to place and route. Together, these have made reconfigurable platforms a feasible alternative to *Application Specific Integrated Circuits (ASICs)* for a range of applications.

1.1 Reconfigurable Computing Platforms

The aim of any reconfigurable platform is to provide the ability to put, with ease, user logic design on to the device. A non-trivial, RC is attached to a processor called the *host controller*. This machine is responsible for providing configuration bitstream, control signals, data for the design etc. In a typical RC, FPGAs provide the computational power. Memories are used to provide input and output data for the design. These memories can be shared between FPGAs or they can be local to a single FPGA. The host machine accesses the memories either directly or through the FPGA. For a multi-FPGA RC, a set of interconnection network is present for communication between various design parts mapped to different

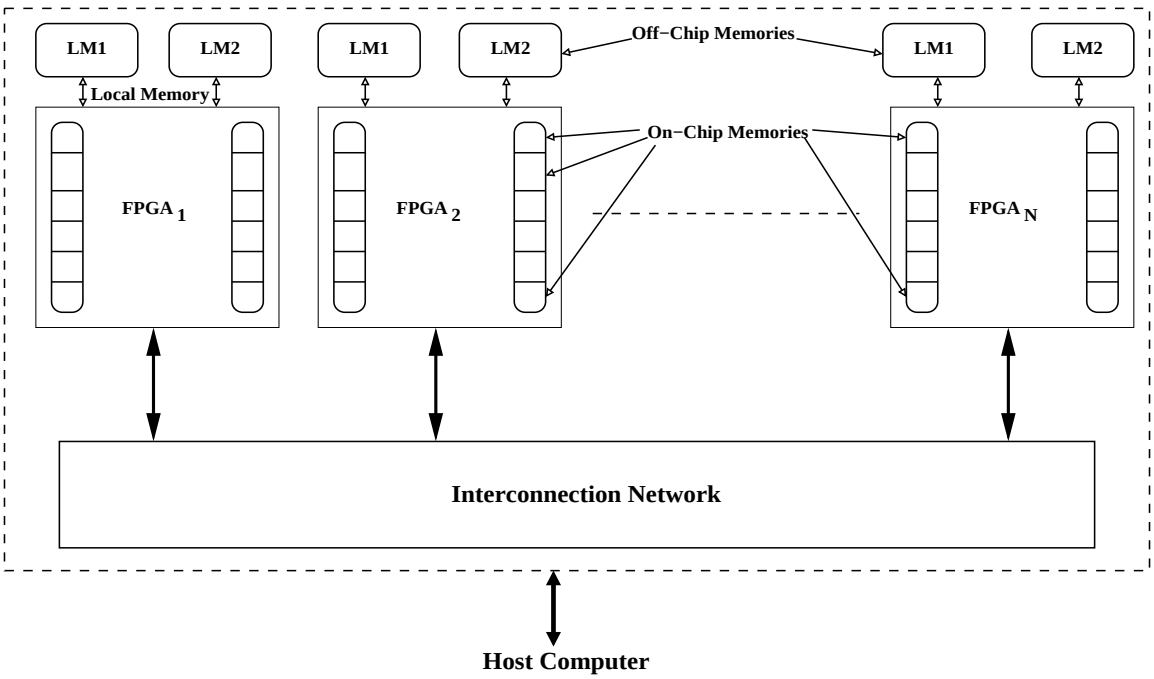


Figure 1.1: *A Typical FPGA Based Reconfigurable Platform*

FPGAs. This interconnection may be programmable to provide additional flexibility to accommodate needs for a range of applications. A logical architecture of a typical RC is shown in Figure 1.1. Many RCs are available from both academia and industry. Some examples of commercial RCs are Annapolis Microsystem's Xilinx 4013 based *WildForce* [4] and Virtex based *WildStar* [5] series, and Altera's RIPP-10. University of Cincinnati's RACE [26] and University of Southern California's SLAAC-1V [32] are examples of RCs developed in the academia.

1.2 Memory Features on a RC

A design, down-loaded onto the FPGA, needs some inputs, upon which it can operate and produce some outputs as the result. This can be looked upon as essential communication of the design with its environment. The input and output to the design can be given in following ways :

- Input is made part of the design itself and output is directed to either some device on the RC (e.g. display) or is read back by the controller immediately. In this case, inputs are synthesized, along with the design, onto the device. This is not a feasible option for a non-trivial design. Besides, the whole synthesis has to be carried out again for every set of input data.
- The host controller gives inputs to the design as and when required. Besides overhead logic for

handshaking between the design and the host, this approach will make the design very slow.

- The host controller writes inputs into some memories on the RC. The design can read from and write to these memories when required. The host can read-back these memories in the end. This approach can handle large amount of input and output data for the design, without slowing it down.

Clearly, the last approach is the most feasible and practicle. All contemporary RCs come with large sizes of memories which the logic inside the FPGA can access.

There are lots of variations in the type of memories found an RC. Some of the memories and their features found on the RC are described below.

1. **SRAM** (Static Random Access Memory)

   Cheap cost and easy availability makes SRAMs the most widely used memories, found on almost all RCs. They are also one of the biggest memories present on a RC. A typical SRAM based memory is around 32-bits wide and can have a depth from a few hundred K-words to a few Mega-words. Typically, SRAMs have read and write latencies of 1 clock cycle each. However, in most cases, FPGA logic can not access these memories directly. A *Memory Interface*, provided by the designer of the RC, controls all accesses to the memory. An interface serves many purposes, like, safeguarding memory against accidental simultaneous accesses by parallely executing logic, making actual pin location details transparent to the user, providing simpler access to memory port signals, making the design portable across device sizes by providing an interface for each device size etc. These memory interfaces generally govern the access latencies of SRAM, For example, in the *WildForce* [4] series of RCs, the interface provides a read latency of 2 clock cycles and a write latency of 1 clock cycle. In the *WildStar* [5] series of RCs, the interface provides read latency of 3 clock cycles. Even for the same interface, there can be different modes of accessing the memories. The *SLAAC-1V* series of RCs have two modes. One is the *pipelined* mode in which although the read latency is 4 clock cycles, the memory can be accessed at a clock frequency of up to 133 MHz. On the other hand, in the *flowthrough* mode, latency is only 3 clock cycles. However, the maximum memory clock frequency can be only 66 MHz. Thus we see that even for the same type of memory, the RC-designers can provide different access latencies, which practically makes them of different types. Figure

   1.2 shows various read and write latencies.

2. **ZBT** (Zero Bus Turnaround Memory)

   ZBTs are memories which allow memory operations on two consecutive clock cycles. The logic can read-back the data from the same memory location to which it wrote in the previous clock cycle.

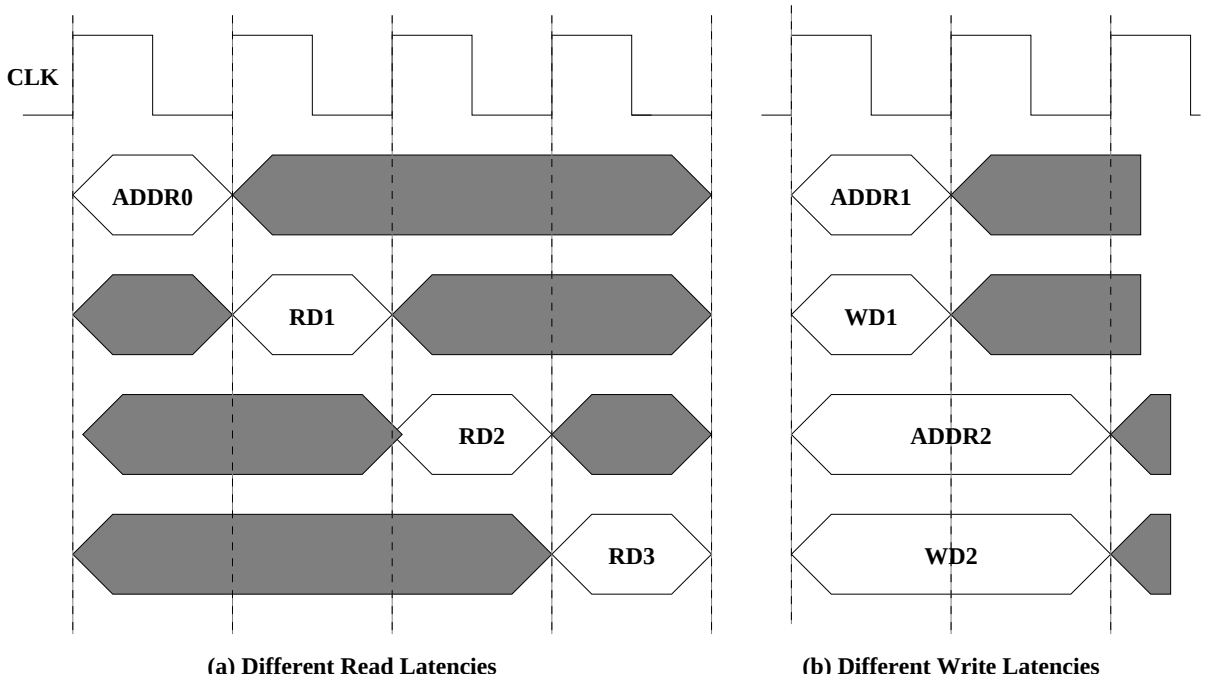


Figure 1.2: *Different Read and Write Latencies*

SRAMs typically require a *dead* clock cycle between two operations. These types of memories are useful for applications which process a large amount of data and where access latencies can make a big impact on the overall design latency.

3. **On-Chip Memories**

   Most contemporary FPGA devices have memories on the chip. Such on-chip memories provide a different memory architecture. There are large number of small-size instances which provide fast access. The *Xilinx-Virtex* series of devices have small memory structures called the *Block RAMs (BRAMs)* [35] which are present on the chip. These memories can be used in either synchronous or asynchronous mode. In synchronous mode, they have a read and write latency of 1 clock cycle each. There are large number of BRAMs available on the chip, e.g XCV3200 has 108 BRAMs. Similarly, *Altera FLEX 10K* [3] series of devices have *Embedded Array Blocks* and *Altera APEX E* [2] devices have *Embedded System Blocks*. Table 1.1 shows the number of on-chip memories present on different families of reconfigurable devices.

4. **Different Number of Ports**

   Besides difference in size and speeds, memories can also differ in the number of ports through which they allow access, e.g. BlockRAMs in Xilinx Virtex FPGAs have 2 ports each. Multiple ports, which allow simultaneous access to same memory space, can be exploited to speed up designs.

Device	Logic Area(#CLBs)	#Block RAMs
XCV50	384	16
XCV100	600	20
XCV150	864	24
XCV200	1176	28
XCV400	1200	40
XCV1000	6144	64
XCV3200	16224	104

Table 1.1: *Number of Block RAMs on various devices of Virtex-E series*

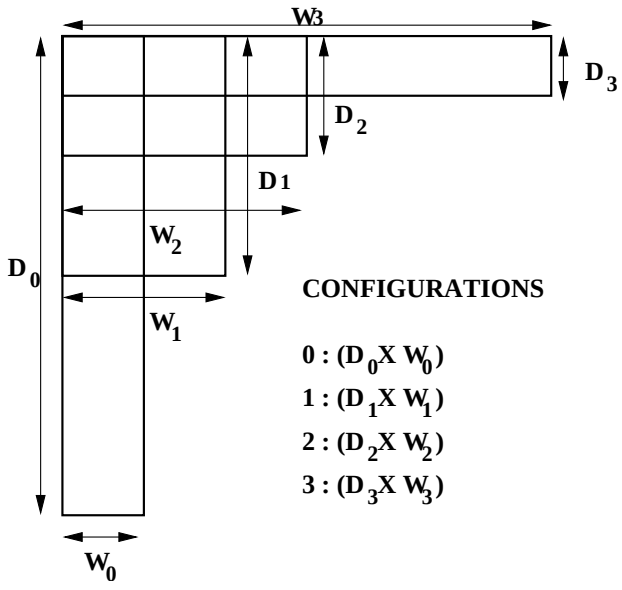


Figure 1.3: *Different Configurations for same storage space*

5. **Different Configurations**

   For certain types of memories, each port of an instance can be configured in a different way. In other words, each port provides a different view of the same physical storage space. This feature is most common for on-chip memories. e.g. Virtex BlockRAMs can be configured to have different Depth-Width combinations Table 1.2 shows the storage capacity and various configurations for memories on different devices. Depending on the type of data structure to be mapped, the memory can be configured in different ways, thus providing opportunity to use it in better ways. Figure 1.3 shows how changing configuration changes the memory accesses.

6. **Pins Traversed**

   The maximum clock speed at which a design operates depends on the length of the longest routed

Device	Max Number of Bits	Configurations
Xilinx Virtex	4096	4096×1, 2048×2, 1024×4 512×8, 256×16
Xilinx Virtex-II	18432	16384×1, 8196×2, 4096×4 2048×9, 1024×18, 512×36
Altera FLEX 10K	2048	2048×1, 1024×2 512×4, 256×8,
Altera APEX E	4096	4096×1, 2048×2, 1024×4, 512×8, 256×16

Table 1.2: *Different configurations for on-chip memories*

wire. Generally, a signal which crosses the chip boundary and traverses across pins causes the most delay. On a RC, different memories require different number of pins to be traversed before reaching the memory ports. On-chip memories do not require any wire to be routed outside the chip and hence can lead to higher clock speeds. Some off-chip memories are connected directly to the FPGA. Some others are connected through interconnect devices, like a cross-bar, thus increasing the path an accessing logic needs to traverse.

1.3 Motivation

Since quick design turn around time and flexibility are one of the most attractive features of a RC platform, an environment which provides ease of designing is necessitated. The environment should be automated at various stages so that the need for designer's interaction, and hence the chances of mistakes, is minimized. A *High Level Synthesis (HLS) Flow* is a suitable option. A typical HLS flow takes a behavioral specification of the design as input. If the estimated area of the design is more than the total area available over all the devices, it has to be *temporally partitioned* [16]. For small designs, there will be only one temporal partition. If there are more than one FPGAs on the RC, each temporal partition may have to be further partitioned so that each part fits one device. Besides, various logical memories accessed by any logic in the spatial partition has to be mapped to some physical memory instance. Together, this can be termed as *spatial partitioning* [27]. This temporal and spatial partitioning results in a set of logic partitions. Each partition is estimated to fit on a single device. Design space exploration [23] can provide an implementation of each logical design segment such that the overall design has improved logic area and

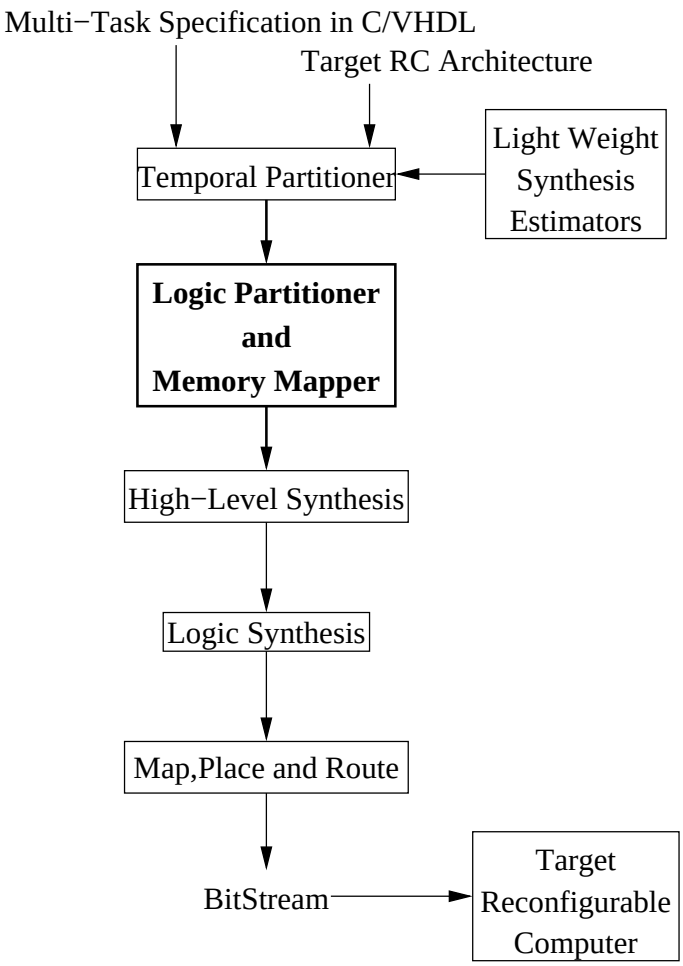


Figure 1.4: *A Typical High Level Synthesis Flow for Reconfigurable Platforms*

latency characteristics. The output of a HLS flow is a structural specification at *Register Transfer Level (RTL)* [19]. These RTL specifications are then synthesized for the specific device to get gate level specifications. Synthesis is followed by mapping the gate level design onto the component-library available on the device. The mapped components are finally placed at some physical location and the interconnect signals are routed. The place and routed [36] design is converted into a bit-stream, which is used to configure the FPGA.

Figure 1.4 shows various steps involved in taking a design through a HLS Flow

The architecture of any RC is already decided at the time of its usage. Thus, there are fixed resources which can be used in certain ways. Any technique which attains better utilization of resources will results in better performance without incurring any extra cost. Thus, it is of utmost importance to exploit the available resources to extract best possible performance.

The previous section lists the features in which different types of physical memories can differ. With in-

crease in design sizes, the number of different logical data structures used in a design varies from a few to a few hundreds. The task of deciding which data structure of the design should be mapped to which physical memory becomes a non-trivial one. Before the arrival of on-chip memories, the total number of physical memories were very few. A RC would typically have between 1 to 10 physical memories, each being single-ported and providing same access latencies. Thus memory mapping could be done by hand. However, with the arrival of on-chip memories, there can be a few hundred physical instances available, having variations in features. With decreasing ratio of number of pins available per logic area available, it will become imperative to utilize *on-chip* memories as storage space. Not only do they alleviate pin requirements, but also provide faster access. Thus the problem of memory mapping becomes a combinatorial problem, its complexity directly increasing with increase in number of logical memories and physical instances. It is virtually impossible to do a good memory mapping by hand. For very large designs, it might be difficult to do even a constraint satisfying memory mapping by hand. For *Data Intensive Applications* like DSPs, Image Processing, Speech Recognition etc., significant improvements can be achieved by using a memory mapper which takes into account these factors. Besides, for proper working of any memory mapping, additional logic needs to be inserted into the design. There clearly is a necessity for an automated memory synthesis framework.

In this thesis, we present a heuristic memory synthesis methodology that tries to improve the overall design parameters like latency, routing resources required, design clock cycle etc. targeted towards RC platforms.

1.4 Related Work

Memory synthesis is the process of mapping various logical data structures used in the design to some appropriate physical instances. As outlined in [14], the process of mapping *logical to physical memory* can be divided into two steps: *(i)* translating the storage requirements onto logical memories, i.e. forming the data structures needed by the design, and *(ii)* mapping the logical memories onto the physical memories of the hardware; i.e. assigning the data structures to the memory banks.

Memory mapping can be broadly classified into two categories, based on the target of the synthesis, *Memory Mapping for Custom Hardware* and for *Predefined Hardware*.

1.4.1 Memory Mapping for Custom Hardware

A lot of research has been done on the problem of memory synthesis for *Application Specific Integrated Circuits (ASICs)*. For ASICs, the problem is that of mapping various logical memory data structures

onto a predefined set of library components. The optimization goals include minimizing the number of different physical memory components used and placing the chosen components so as to minimize routing requirements and signal delay. Minimizing the resources required is as important as optimizing various performance parameters. The constraints are at more abstract level, in terms of the overall area available, design latency desired, number of available pins for off-chip communication etc. Restrictions also arise out of the availability of components in the library.

In research done as part of the *data path synthesis* problem, the tools map each logical memory to some specific component from the library and connect them to the accessing logic. In the context of high level synthesis, many researchers tried to form groups of variables. These groups are then partitioned to form data segments. Some researchers did not take interconnection cost into consideration [31], while others have taken this cost also during variable grouping [18]. An *Integer Linear Programming (ILP)* approach has been used in [1, 6] to group registers to form multi-port memory modules. Interconnection cost incurred in routing address and data buses from memory modules to the accessing logic has been of significant importance in these approaches. Minimizing the number of memory modules required was an important way of reducing the interconnection cost and thus satisfying the pin constraints. In other approaches, [15] concentrate on minimizing the area while finding a legal packing of the logical segments into the physical segments.

1.4.2 Memory Mapping for Predefined Hardware

In the case of synthesis for RC platforms, an important difference is that the resources available are already fixed. Trying to minimize resource utilization may not be the best approach. The goal is to optimize performance while satisfying the constraints posed by the RC architecture.

In [7,34], researchers try to use on-chip memories available on FPGA devices for implementing logic. The approach does not exploit them as storage space for logical memories. [33] considers on-chip memories for memory mapping. In [11], the same technique is improved for dual-ported on-chip memories. In both cases, the framework considers only one type of physical bank and does not handle both single and dual ported at the same time.

In [21], Ouaiss and Vemuri approach the problem using ILP formulation. They target memory mapping for RC architecture. The framework considers all instances simultaneously and gives optimal mapping. However, as the problem size increases, the ILP formulation takes a long time to converge. In the extension of this work in [20], mapping is done in two stages. First, *Global Mapping* and *Detailed Mapping*. Global mapping predicts how mapping would be done if a logical memory is mapped to a particular type of physical memory. This prediction is done for all logical memory and physical memory type combinations.

Based on the predicted mapping, an optimal assignment of physical memory type is done for each logical memory. During *detailed mapping*, for each type of physical memory, all instances of that type and all logical memories assigned to it in the earlier stage are considered. Mapping a logical memory to specific instance(s) and port(s) is performed. This results in much faster execution time. However, if a logical memory is bigger than an instance, this approach allows the logical memory to be split only in some predefined manner, decided during the preprocessing stage. Thus, some of the solutions are excluded. This restriction may prohibit the mapper to find a solution where one may exist.

1.4.3 Integrated Spatial Partitioning and Memory Mapping

If a design is bigger than an individual logic device, it has be *spatially partitioned* . Spatial Partitioning is very closely linked to memory mapping. The quality of memory mapping depends heavily on where the accessing logic is placed. If accessing logic is placed in a device such that address and data signals need to be routed across chips, the clock period of the final design will significantly deteriorate. In addition, it will also consume the scarce interconnection resources. [17] and [8] present very early work done in the field of logic partitioning. However, they deal with a very fine level of granularity which is not suitable during high level synthesis. In [27], Srinivasan and Vemuri handle the two problems at the same time. However, the memory mapping part is simplified and does not consider on-chip memories or multi-ported memories. It assumes all physical instances to be of the same type. It further assumes only one physical instance local to every FPGA. Each logical memory was required to fit into a physical instance. Splitting across multiple instances was not permitted.

1.5 Overview of the Thesis

In the previous sections of this chapter, we presented the various steps involved during the high level synthesis flow targeted towards RCs. We motivated the use of a memory mapping methodology in the flow and showed the impact it can have on design quality.

The focus of this thesis is the memory synthesis technique used to improve design latencies. It can be done at various stages

- *Stand-alone Memory Mapping*: Mapping technique, used to minimize design latencies without directly considering the logic which accesses the memories, is presented. In the absence of any information about logic partitioning, the mapper tries to optimize various objectives like : (1) the access times of various logical memories, (2) the total number of address and data pins which might poten-

tially be routed across devices, (3) the number of blocks of data that can be processed, (4) the effect of physical location of memory instances on design clock frequency. The mapper assumes that there is only one FPGA and all logic has been mapped to this FPGA. It is also assumed that all physical memory types are local to the only FPGA. A heuristic called *Rectangle Carving* is presented in this thesis. This heuristic is used to map each logical memory to multiple instances of a type of physical memory. Tabu Search is used to guide this heuristic.

- *Partial Information from Logic Partitioner:* The mapping technique can take advantage of some of the information which might be available if logic partitioning has been performed. For example, if information regarding which FPGAs contain the tasks accessing a particular logical memory is known, various routing requirements can be determined. The effect of remote memory accesses (logic accessing physical memory not local to its FPGA) on clock can be estimated.

- *Integrated Logic Partitioning and Memory Mapping:* An integrated approach to perform both logic partitioning as well as memory mapping is presented. This approach promises to deliver high quality porting of the design to the target RC architecture. The search engine is guided by Tabu Search. At each iteration, some compute tasks and logical memories are re-mapped from their current position. Again, the heuristic, Rectangle Carving, is employed to find a mapping of logical memory for a different memory type.

1.6 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 discusses various aspects of the target RC architecture. It presents various components of the input design which is to be mapped. The chapter deals in detail about the Tabu Search technique employed to guide both the memory mapping and logic partitioning algorithms and the adaption of these techniques to specifically suit the problem in hand.

Chapter 3 presents the main contribution of this thesis, the memory synthesis framework. It presents details of how the problem is formulated for Tabu Search and the algorithm used to perform mapping of individual logical memories. We also present various factors used to evaluate a given memory mapping solution and the results of memory mapping.

Chapter 4 presents the integrated logic partitioning and memory mapping technique. It discusses how the integrated approach is different from the stand-alone memory mapper presented in Chapter 3. Results of partitioning on various benchmark examples is presented to show the the effectiveness of the technique.

We present the summary and some directions for the future work in Chapter 5.

Chapter 2

Definitions and Methodologies

In this chapter, we introduce various terminologies related to the *memory mapping* and *logic partitioning* problem. These terms will be used in the rest of the thesis. Here, we will give a broad definition of every term. Different assumptions will be made about these definitions in various parts of the thesis. Depending upon the context of the discussion, we will state those assumptions, the need for them and their justification. We classify the definitions into two categories. *Board Architecture* related definitions are presented in Section 2.1, and *Input Design* related definitions are presented in Section 2.2

Section 2.3 outlines the *Tabu Search (TS)* algorithm, which is used to guide both memory mapping and partitioning algorithms. We present the basic Tabu Search algorithm which is independent of the problem it is used to solve. In Section 2.4, variations and enhancements of the TS used for both memory mapping and spatial partitioning are presented. We summarize the discussion by briefly mentioning about other methodologies which can be used for the problem in Section 2.5.

2.1 Board Architecture

A generic reconfigurable platform will essentially have a set of processing elements (PEs), a set of physical memories, and an interconnection network for communication between various PEs and memories. These are the relevant and significant parts of the RC board and hence discussed here.

1. **Processing Element (PE)**

   A processing element is the FPGA device present on an RC. It provides the computational power to the RC. A PE has limitation in the form of **total area** available for implementation of various logic components. Most of the area on a PE is typically consumed by components that are part of the

input design. Some area is also consumed by the components introduced by the memory mapper to implement memory control and arbitration logic. The aim of the synthesis process is to ensure that the total logic area requirements for a device does not exceed the maximum available area on the device.

2. **Memory Type**

   The various physical memories on a RC can be grouped into different clusters on the basis of similarities in some of their *invariant attributes*. Each such group is said to constitute a *memory type*. A memory type should be viewed as a collection of attributes of physical memories rather than that of physical memories themselves. The attributes which are common among elements of a group of memories are given below.

   local_pe is the processing element which is local to the instances of this memory type. Any logic present in the *local_pe* does not consume any additional routing resources. A fixed connection already exists on the RC between the instances of a memory type and its *local_pe*. It is preferable to have all the accessing logic of a physical memory in its *local_pe*.

   pins_traversed is the number of pins which the logic in the local_pe of this memory type needs to traverse in order to access an instance of this memory type.

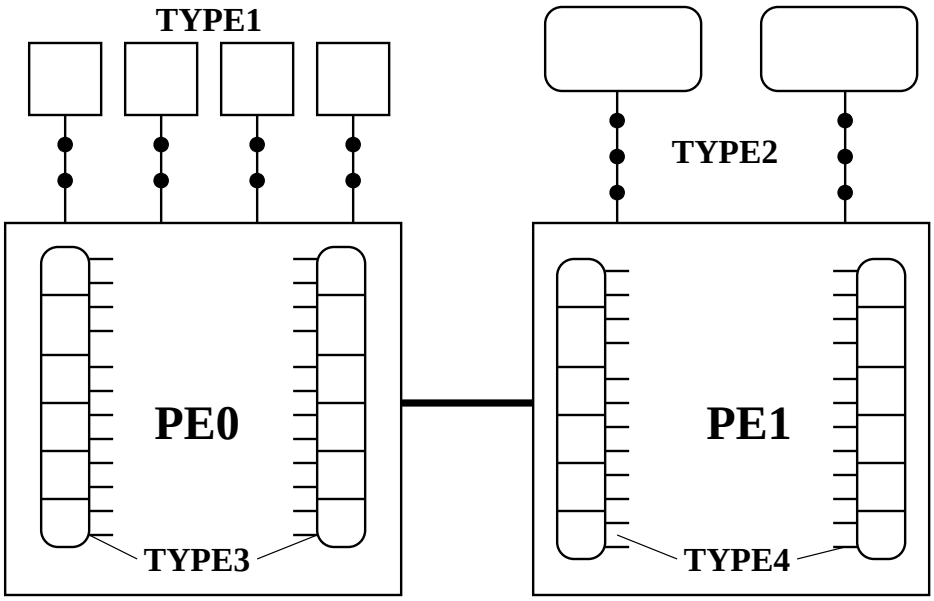
   read_latency is the number of clock cycles for which the logic has to wait before it gets the data requested.

   write_latency is the number of clock cycles for which the data must be available at the input port of the memory so that it can be written reliably. The more clock cycles logic has to wait for accessing (read or write) the memory, more will be the overall design latency. It is desirable to use instances of those memory types which have smaller latencies.

   num_ports is the number of ports available on **each** instance of this memory type. The storage space in a physical memory can be accessed through any of these ports in parallel.

   max_storage_bits is the maximum number of bits of data that can be stored in an instance of this memory type. The mapping tool should ensure that this upper limit on storage space is not exceeded. At the same time, it is desirable to use maximum possible storage space from each instance.

   num_configurations is the number of different ways in which a port on instance of this memory type can be configured. **configuration** of a port is the way in which storage space is accessed through that port. It is specified by the **(width,depth)** pair. Each word of the memory has *#width* bits and there are total *#depth* words available for that configuration.



TYPE1 :
local_pe = 0
pins_trav = 2
rd_lat = 2
wr_lat = 1
num_ports = 1
max_bits = 1 M−bits
num_conf = 1
conf_set = {32K x 32}
num_insts = 4

TYPE2:
local_pe = 1
pins_trav = 3
rd_lat = 4
wr_lat = 2
num_ports = 1
max_bits = 16 M−bits
num_conf = 1
conf_set = {512K x 32}
num_insts = 2

TYPE3:
local_pe = 0
pins_trav = 0
rd_lat = 1
wr_lat = 1
num_ports = 2
num_bits = 4096
num_conf = 5
conf_set = { 4K x 1, 2K x 2, 1K x 4, 512 x 8, 256 x 16}
num insts = 12

TYPE4:
local_pe = 1
pins_trav = 0
rd_lat = 1
wr_lat = 1
num_ports = 2
num_bits = 4096
num_conf = 5
conf_set = { 4K x 1, 2K x 2, 1K x 4, 512 x 8, 256 x 16}
num insts = 12

Figure 2.1: *Various Features of Memories Available on a RC*

configuration_set is the set of various configurations possible for this memory type. The number of elements in this set is equal to *num_configurations*. The storage space can be viewed as a rectangle. There is an upper bound on the area of the rectangle. However, the width and depth of the rectangle can be chosen from the available options.

group_num is used as an identification for the memory_type.

num_instances indicates the total number of elements of this memory type which are available on the board.

Figure 2.1 shows the variations in different features of memories available on the reconfigurable board. Note that memory type 3 and 4 are identical in all respects except the local_pe.

3. **Physical Memories**

   Physical Memories are the data storage units available on the RC for storing inputs, outputs and intermediate data produced as part of the computation. Each physical memory *logically* belongs to a memory type. Attributes of a physical memory are given below:

   mem_type_num is the identification of the memory type to which this instance belongs.

   instance_num is the id of a memory instance over all the physical memories available on the board. Note that instance_num includes instances of all memory types.

   set_of_ports are the ports which are part of this physical memory. The size of this set is equal to num_ports of the memory type of this instance.

4. **Memory Port**

   A physical memory instance can have more than one port. All these ports provide access to the same storage space. Data written by one port can be overwritten by another port if care is not taken. Attributes of port are shown below

   parent_instance_num is the *instance_num* of the physical memory on which this port is present.

   port_num is the identification of the port on the parent memory instance. If a physical memory has k ports, port_num varies from *0 to k-1*.

   selected_config is the way in which this port is configured. If the memory type of the parent instance has c configurations, selected_config can vary from *0 to c-1*.

5. **Interconnection Network**

   If parts of a design, mapped to different PEs, communicate with each other, signals have to be routed between the PEs, through the pins. A direct connection may not be available between two PEs and the signal might need to be routed through another PE or through a interconnection device like a *crossbar*. Anything which facilitates *PE-to-PE* communication can be termed as part of the *interconnection network*. In our work, we assume that a direct connection is available between every pair of PE. If a direct path is not available, some pins from an indirect path can be set aside for communication between the PE-PE pair under consideration. This capacity is not considered for PEs which lie on the path. This assumption can be alleviated by integrating the tool with a smart board-level router. Since routing for generic board architectures is not the focus of this thesis, we simplify our framework without losing any important aspect of the problem under focus. A generic board router is presented in [24].

2.2 Input Design

A Unified Specification Model (USM) is presented in [13]. It identifies four main components of an input design. We use these definitions for input design. They are presented here for convenience.

1. **Compute Tasks**

   As the name suggests, the computational part of the design is done by *compute tasks*. They are synthesized into logic and mapped onto PEs. Each node is assumed to be the smallest unit of logic which can not be broken further. In the rest of this thesis, we will use the words Compute Task and Task interchangeably. Attributes of a compute_task listed below

   area represents the estimated size of the compute task. Note that the *area* is some multiple of the smallest measurable unit of area on the target devices, e.g. for a Xilinx Virtex FPGA, the smallest unit is a *Slice*. Thus, area of a task will change if the PE device in the target architecture changes.

   pe_num is the PE element on which this compute_task will finally be mapped.

2. **Logical Memory**

   A logical memory can be defined as a set of data in the design which can be grouped together and abstracted into one component. The grouping is not on the basis of size but on the basis of the role played by the component in the design, e.g each data structure declared by user in High-Level specification of the design can form a logical memory. For this work, we assume that logical memories have already been formed. A logical memory task has the following attributes

   depth represents the number of words present in the memory task.

   width represents the number of bits per word.

   num_reads indicates how many reads are performed on the logical memory during the run of the design

   num_writes indicates how many writes are performed on the logical memory during the run of the design. A heavily accessed memory has a greater influence on the overall latency of the design. This information can be of great help in producing good quality mapping. If this information is not available, both *num_reads* and *num_writes* are assumed to be equal to the *depth* of the design.

   Mapping for a memory task is a detailed specification of how and to which physical memory or memories it is assigned. We define mapping in more details later in the thesis.

Accessing_compute_task is a list of all the compute tasks which access this logical memory atleast once.

3. **Channels**
   Channels represent direct communication between *compute tasks*. The attributes of a channel include **writer_task, reader_task**, and **width**. If the writer and reader tasks are mapped onto different PEs, the channel will consume routing resources.

4. **Flags**
   Flags can be considered as single bit channels used to synchronize the execution of tasks. The attributes of a flag are **writer_task** and **list_reader_task**.

For a detailed discussion, interested readers are referred to [13].

2.3 Tabu Search

Tabu Search (TS), was introduced by Fred Glover et al. [9]. It is a general purpose *meta-heuristic* for solving combinatorial optimization problems. It guides a local heuristic search procedure to explore the solution space beyond local optimality. One of the main components of TS is its use of adaptive memory, which creates a more flexible search behavior. Memory-based strategies are therefore the hallmark of tabu search approaches.

Figure 2.3 gives the outline of the Tabu Search Algorithm *(as given in "Iterative Computer Algorithms with Applications in Engineering" by Sait and Youssef, 1999, IEEE Computer Society)* [25]. The procedure starts from an initial feasible solution *S* (current solution) in the search space Ω. A neighborhood $\aleph(S)$ is defined for each S. A sample of neighbor solutions $V^* \subset \aleph(S)$ is generated, called *trial solutions* ($|V^*| = \mathbf{n} \ll |\aleph(S)|$). From these trial solutions, the best solution, say $S^* \in V^*$, is chosen for consideration as the next solution. The move to S^* is considered even if S^* is worse than S, that is, $Cost(S^*) > Cost(S)$. It is this feature that enables *escaping* from local optima.

However, it is possible that the search reaches a local minima, ascends (by accepting moves to lower quality solutions), and then return back to the same local optimum, a phenomenon called *cycling*. To prevent this, a list called **tabu list** is maintained. Only selected move attributes are stored in the tabu list, since completely storing previously visited solutions, and comparing them with newly generated ones would be expensive in terms of both computation time and memory requirements. The tabu list can be visualized as a window on accepted moves, as shown in Figure 2.3. Based on the tabu (forbidden) attributes in this list, moves which tend to undo recently made moves are identified and are not accepted.

Algorithm *Tabu_Search();*
Ω : Set of feasible solutions.
S : Current Solution.
S^* : Best admissible solution.
Cost : Objective function.
$\aleph$(S): Neighborhood of $S \in \Omega$.
$\mathbf{V}^*$: Sample of neighborhood solutions.
T : Tabu list.
AL : Aspiration Level.
Begin
Start with an initial feasible solution $S \in \Omega$;
Initialize tabu lists and aspiration level;
For fixed number of iterations **Do**
    Generate neighbor solutions $V^* \subset \aleph(S)$;
    Find best $S^* \in \mathbf{V}^*$;
    If move S to $S^* \notin$ **T Then**
        Accept move and update best solution;
        Update tabu list and aspiration level;
        Increment iteration number;
    Else
        If $Cost(S^*) <$ **AL Then**
            Accept move and update best solution;
            Update tabu list and aspiration level;
            Increment iteration number;
        Endif
    Endif
EndFor
End

Figure 2.2: *Tabu Search Algorithm*

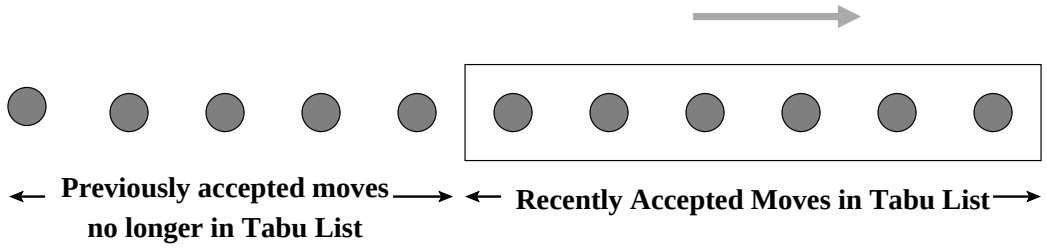


Figure 2.3: *Tabu List*

Since tabu list has only some of the attributes of a move, it may also forbid moves to attractive *unvisited* solutions. The notion of *aspiration criterion* helps to relax the actions of the tabu list and overrule the tabu status of moves in certain situations. It temporarily overrides the tabu status if the move is sufficiently good. It must ensure that overriding the tabu status leads the search to an unvisited solution.

In Section 2.4, we discuss usage of long-term and intermediate-term memories for the memory mapping problem.

2.4 Tabu Search Enhancements for Memory Synthesis

Any search heuristic is very sensitive to the manner in which its features are exploited. In this section, we present more details into the various techniques adapted for *Tabu Search*.

1. **Neighborhood Moves**

   Since neighborhood space is totally random, we make random choices to select neighbors of the current solution. We randomly pick out a small number of logical memories (LM). While retaining the mapping for the unselected LMs, we re-map the picked out LMs using the algorithm described in Section 3.5.

2. **Tabu Attributes of a Move**

   Once a move has been made, we store two attributes for each of the LMs re-mapped. If l is one of the LM that was moved, we store *from_bank* i.e. the memory type to which l was mapped in the previous solution. Second is the *to_bank* i.e. the memory type to which l is mapped after the move. A move which contemplates to re-map l from the *to_bank* or to the *from_bank* (i.e. a move which undoes a recently made move) will be tabued for next few iterations, called the *tabu tenure*. This contemplated move can still be made if it can override the tabu status by satisfying the *aspiration criterion*.

3. **Tabu Tenure**

   Tabu Tenure is nothing but the length of the tabu list. This is the number of iterations for which a move is kept tabu active. Search is highly sensitive to this parameter. Ideally, this parameter should depend on the problem size, and the type of attributes which are tabued. However, to make it independent of the problem size, we use a varying tabu tenure, between 4 and 9. New tabu tenure is randomly picked from this range after every $2 \times curr\_tabu\_tenure$ moves have been made. This has shown to give good results for different sizes of problems.

4. **Aspiration Criterion**

   We use *Global Aspiration by Objective*. If the contemplated move has a value better than the best value seen so far, it indicates that the search has not visited this solution before, thus acceptance of this move is not leading to *cycling*.

5. **Residence Frequency**

   This is a long-term memory of the tabu search. It is a *2-dim* array of size #LM $\times$ #Memory_Types, where each element indicates the total period for which the LM was mapped to that bank type. It is an indication of the suitability of mapping the LM to that bank type. If the search is in a region of high quality solutions, a hight residence frequency represents a desirable LM-bank_type combination. By rewarding a high residence frequency, the search can focus on those solutions which have the good combination of LM-bank_type. This can lead to *intensification* of the search leading to better solutions. Similarly, in the region of poor quality solutions, a high residence frequency represents a poor LM-bank_type combination. By penalizing a high residence frequency, the search can be forced to move away from such solutions leading to *diversification* of search, exploring new regions in solution space.

6. **Transition Frequency**

   This is another form of long-term memory. It is a *1-dim* array of size #LMs. Each element holds value equal to the number of moves which re-mapped that LM from one bank type to another. A higher value shows that those moves contemplating re-mapping this LM were readily accepted. LMs with high transition frequency generally are smaller in size and easily *fit* many bank types. A high transition frequency indicates *fine-tuning* moves, which do not change the solution by a great extent but cause localized search. This can be suitably rewarded or penalized to *intensify* or *diversify* the search.

7. **Weighted Tabu Evaluation**

   The simple version of tabu search picks up the best neighbor based solely on the cost calculated

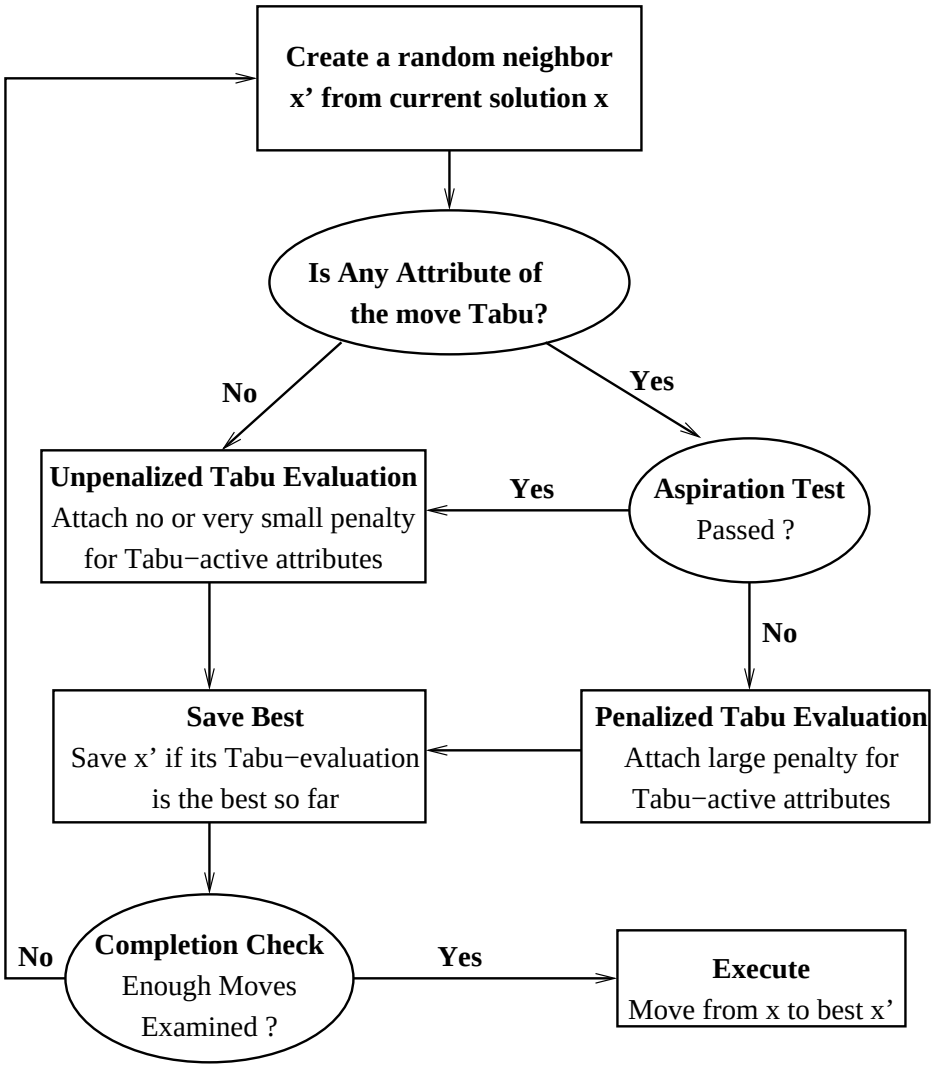


Figure 2.4: *Flow Chart for Weighted Tabu Evaluation*

by solution attributes. However, a modified cost function can also take into account the long term memory attributes of the move. The neighbor having best weight is chosen and tested for tabu status. Figure 2.4 shows the flowchart for weighted tabu evaluation.

$$Weigh(next\_solution) = func(solution\_cost, transfreq, resifreq)$$

8. **Restart**

This is a form of medium term memory. The solution space for memory mapping can be pictured to be divided into regions. Solutions within a region have similar characteristics. Small changes obtained by re-mapping a few LMs will not lead to significant change in the solution quality. After

a fixed number of iterations, an average cost of all the solutions explored in the current region so far is calculated in every iteration and examined. It can be used to judge whether further exploration of the current region will lead to a better solution or not. If this average cost is significantly higher than the cost of best solution found so far, chances of finding a new best solution in this region are very small. Then a new random solution is generated. This solution is compared with representative solutions from other regions to ensure that the search is conducted in a new region. This technique is specially useful in cases where it is difficult to find even a constraint satisfying solution. Every time the search is restarted, all the other short and long term memories are reset.

9. **Probabilistic Tabu Search**
   We explained the idea of performing a weighted tabu search earlier. There, we suggested to pick the solution with best weight. Another technique suggested in literature is to use these weights as relative probabilities of each solution. A solution with higher weight will have higher chances of being chosen. The basic idea is to introduce more randomness into the search as a hedge against risks of being too greedy. However, for memory mapping, we found that the quality of search, in terms of both time to find solution and the quality of solution itself, significantly **deteriorated**. Since introduction of probabilities reduces the localized greedy nature of tabu search, the ability to find solutions quickly is reduced. This technique might become helpful when the search needs to continue over a very long period. Our approach has randomization in choosing candidates from the neighborhood. *Restart* also provides a way of visiting new solution regions.

$$probability\_acceptance \propto Weight(contemplated\_solution)$$

2.5 Discussion and Summary

Meta-heuristic techniques have been classified as evolutionary methods and adaptive memory strategies. Evolutionary techniques manipulate a collection of solutions rather than a single solution at each stage. *Genetic Algorithm (GA)* is an example of such technique. *Simulated Annealing (SA)* is somewhat similar to TS in the way that both manipulate only one solution at a time. Our initial comparison between SA and TS showed that even basic TS outperformed SA in most cases. The basic TS implementation did not include any of the search enhancing techniques which can potentially make TS more powerful. Thus we decided to choose TS over SA.

In this chapter, we introduced the common terminologies used in the thesis. We introduced terms related to the target RC architecture. We introduced the essential components required to specify the input design,

namely *compute_task* and *memory_task*. We also discussed about what constitutes a mapping for each type of task.

We described the basic Tabu Search algorithm, which is the main driving algorithm used in our approach. The details about adapting the TS to the given problem is also presented here.

Chapter 3

Memory Synthesis

This chapter deals with the issues and algorithms for memory synthesis problem targeted towards FPGA based Reconfigurable Computers (RC). In Section 3.1, we present the details of the input specification and its variations accepted by the memory mapper. The details for mapping of a logical memory and formulation of the problem are presented in Section 3.2 and 3.3 respectively. Section 3.4 shows a model of logical memory used by the tool while performing mapping. Section 3.5 presents a heuristic mapping algorithm, called *Rectangle Carving*. This heuristic effectively makes use of the memory model. The tabu search is used to guide this heuristic.

After the memory mapping stage, the second stage in memory synthesis is control logic generation. This is discussed in Section 3.6. Section 3.7 presents the two different categories of constraints faced by the algorithm and their effect on the quality of result produced. Section 3.8 presents various factors which the algorithm tries to optimize to achieve better quality results. Experimental results are discussed in Section 3.9. Finally, in Section 3.10, we present conclusions and summary of observations.

3.1 Design and Architecture Specification

In Section 2.2, we have introduced four components, namely, compute tasks, logical memory, channels, and flags. However, as far as memory synthesis is concerned, we need to consider only logical memories. Later in this section, we will state how memory mapper can indirectly take the other factors into consideration.

The design specification input is a *set* of *logical_memories* which are to be mapped onto the target architecture. Except *accessing_tasks*, all other attributes of a *logical_memory* are considered. The target architecture specification includes a *set* of *memory_types*. Except *local_pe*, all the attributes associated

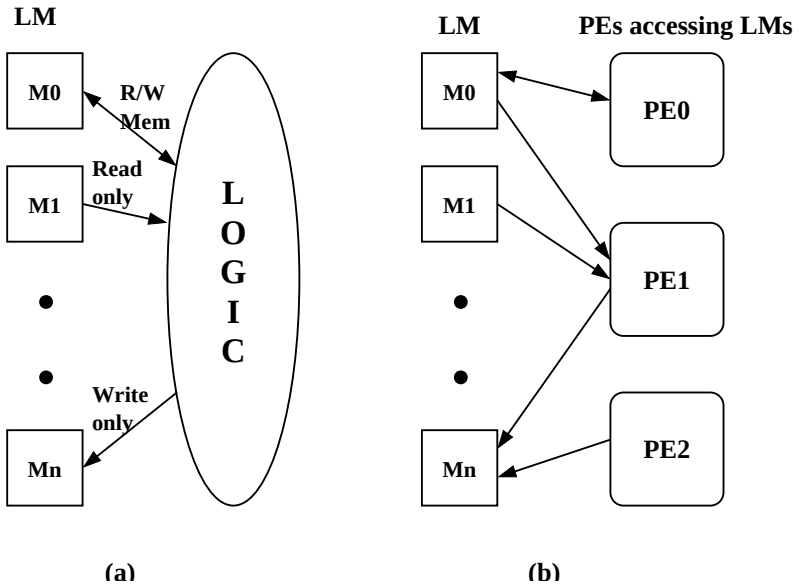


Figure 3.1: *Design Specifications with and without logic partitioning information*

with a *memory_type* are provided as input. In the absence of any information from logic partitioner, information regarding *accessing_tasks* and *local_pe* can not be utilized. We assume that there is only one PE. All *memory_types* are local to it and all *compute_tasks* are present in this *PE*. This is shown in Figure 3.1 (a).

If information from logic partitioning is available in the form of which *compute_task* is assigned to which PE, the memory mapper can also take this input. For each *logical_memory*, there will be information about the PEs in which its accessing *compute_tasks* are present. Memory mapper does not need to know any other details of the *compute_tasks*. Alternatively, it can directly take the output of the logic partitioner as input, in which there are complete details about the compute_tasks. The memory mapper can extract the information about which PEs are accessing a logical_memory depending upon where its accessing compute_tasks are placed. The architecture specification will need to specify the following additional information:

- number of FPGAs on the RC.
- local_pe for each memory_type.
- number of interconnection lines available between each PE-PE pair.

Figure 3.1 (b) shows the second forms of design input accepted by the memory mapper.

3.2 Mapping Definitions

A *Logical Memory* can be divided in an arbitrary manner across multiple instances of physical memories and multiple ports. If the logical memory is *split*, mapping should specify which part of the logical memory is mapped to which instance and port. We view a logical memory as a *rectangle*. We call this the *complete rectangle*. If split, the complete rectangle can be decomposed into *sub-rectangles*. A mapped sub-rectangle is specified by a tuple of 6 values described below.

physical_mem_num specifies the physical instance to which this sub-rectangle has been mapped.

port_num specifies which port of that physical instance will be used to access this sub-rectangle.

depth and **width** are the dimensions of the sub-rectangle.

start_depth and **start_width** define the *top-left corner* of the sub-rectangle with respect to the complete rectangle.

We define a mapping to be a set of sub-rectangles, one corresponding to each part into which the logical memory has been split.

$$\mathbf{mapping} = \{\mathbf{subrect_1}, \mathbf{subrect_2}, \mathbf{subrect_3}, \ldots, \mathbf{subrect_N}\}$$

In the best case scenario, the logical memory will not be split and there will be only one sub-rectangle. The *start_depth* and *start_width* of the only sub-rectangle will both be 0. The *depth* and *width* will be equal to that of the logical memory itself.

Validity : For a mapping to be valid, we require that all sub-rectangles of the mapping be mapped to physical instances of **same memory_type**. We further require that no two sub-rectangles in a mapping share the same port.

Justification : Consider a scenario in which different parts of a logical memory are mapped to two physical memories whose *local_pes* are not same. The address and data buses will need to be routed to both the PEs. This can greatly deteriorate the quality of the solution. Even if the *local_pe* of the physical memories are same, they might differ in read and/or write latencies. If the width of the logical memory is split, a read operation will present different bits of the same word in different clock cycles. In this case, the accessing logic will have to change depending on which part of the logical memory is mapped to which physical instance. Our assumption excludes all such scenario from a valid solution. This leads to reduction in solution space without deterioration in the quality of the solution produced.

3.3 Problem Definition

Given

- **Set** $\mathcal{L} = \{l : logical\_memory\}$, where each $l \in \mathcal{L}$ has the attributes specified in Section 2.2. $\mathcal{L}$ specifies the input design.
- **Set** $\mathcal{T} = \{t : memory\_type\}$, where each $t \in \mathcal{T}$ has attributes specified in Section 2.1. $\mathcal{T}$ is part of the target architecture.
- set $\mathcal{P} = \{pm : physical\_mem\_instances \ | \forall pm : \exists_1 t \in \mathcal{T}\}$. The physical_mem_instance is as specified in Section 2.1
- If logic partition information is available, the target architecture specification will also include num_fpgas and **set** $\mathcal{I} = \{i_{f_1 f_2} | 0 \leq f_1, f_2 < num\_fpgas\}$. Each element of $\mathcal{I}$ specifies the number of interconnect pins available between the corresponding fpga pair.

The objective of the memory mapper is to produce another set

$$\mathcal{M} = \{m : memory\_mapping | \ \forall l \in \mathcal{L}, \ \exists_1 \ l \leftrightarrow m\}$$

such that

$$\{ \ \forall pm \in \mathcal{P}, \ satisfies(Constraints_{pm})\}$$

3.4 Modeling of Logical Memory as a Rectangle

One of the essential features of a logical memory which we try to capture is its size. Size is a dominating factor in deciding the destination of a LM on the RC.

Figure 3.2 shows a model of a LM. The number of words of the logical memory is modeled as depth of a *rectangle* while the number of bits per word is represented by width of the *rectangle*. The weight of a rectangle is the number of times the LM is accessed in the design. If this information is lacking, we assume it to be equal to the number of words in the LM.

The aim is to map all parts of this rectangle to some physical memory on the RC. A good mapper should try to map the complete rectangle onto single physical instance. The splitting of LM over multiple physical

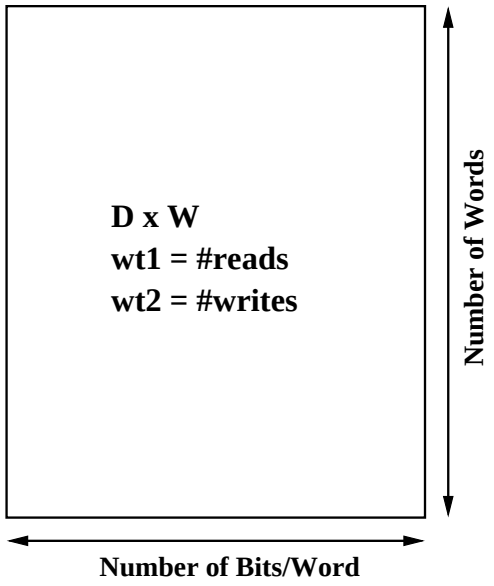


Figure 3.2: *Rectangle Model for A Logical Memory*

instances is indicated by the splitting of the rectangle into various sub-rectangles, each part mapped to a single physical port.

Figure 3.3 shows various ways in which a rectangle can be split. Rectangles might be split depth-wise if the physical instances are not deep enough, or they may be split width-wise if the ports are not wide enough to accommodate the complete width. Larger LMs might be split both ways. If after splitting, a sub-rectangle still does not fit the physical instance, it will be further split until all parts have been completely mapped.

3.5 Heuristic : Rectangle Carving

In Section 3.4, we presented how a logical memory is represented. In this section, we show how this representation is used in the mapping algorithm.

At every iteration, the Tabu Search contemplates a move which re-maps some of the logical memories to new physical banks. While Tabu Search decides which LMs to be re-mapped and to which memory type they should be re-mapped, the new mapping for LM is found by a lower level heuristic. The heuristic, which we call **Rectangle Carving**, efficiently makes use of the rectangle model of a logical memory. It takes as input the logical memory to be mapped and the physical bank to which it has to be mapped and returns a mapping of the LM for that memory type.

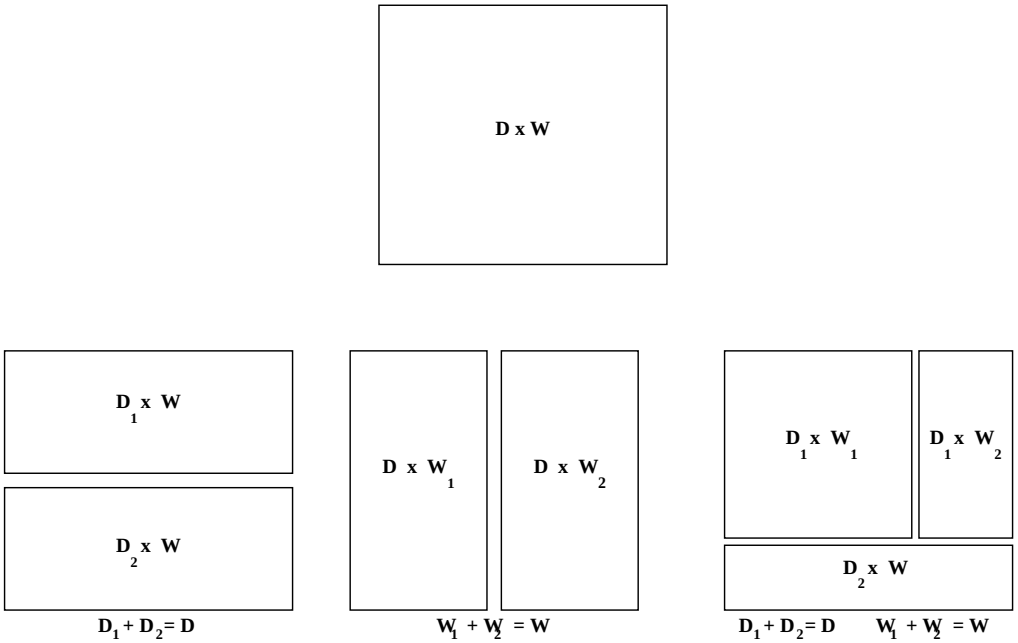


Figure 3.3: *Possible Sub-rectangles for a Rectangle*

The algorithm reads in the logical memory as a rectangle. It maintains a list of rectangles which are to be mapped. This list is initialized with the rectangle corresponding to the LM being mapped. The algorithm maps each rectangle in this list, one at a time. New rectangles are created out of the unmapped part of this rectangle and appended to the list.

In each iteration, the algorithm tries to map the first rectangle in the list. It randomly picks one of the instances in the physical bank and then one of the ports of that instance.

port_usable performs a check to see if this port has already been assigned to some other sub-rectangle, either of same or different logical memory. If port sharing between various sub-rectangles is permissible, *port_usable* returns *true* indicating that this port can be used to map the rectangle under consideration.

Carve_rectangle considers a *subset of configurations* of this port and calculates a fitness value of mapping the rectangle to each configuration. The fitness depends upon how much of the rectangle was mapped, whether the rectangle had to be split width-wise of depth-wise and how much storage space would go to waste (for configurations where width of the rectangle is less than that of the configuration). We put a higher penalty if rectangle has to be split width-wise. The fitness function has following factors.

$$pct\_depth = assigned\_depth/rect\_depth$$

$$pct\_width = assigned\_width/rect\_width$$

Algorithm: Map_LM

Input:

$\mathcal{M}$: Logical Memory

$\mathcal{B}$: Bank Type

Output:

$\mathcal{S}$: Solution, a set of mapping to ports

begin

$\mathcal{C}, \mathcal{R}, \mathcal{N}$: Rectangle

$\mathcal{L}$:List of Rectangles to map

$\mathcal{P}$:Physical Port

$\mathcal{U}, \mathcal{FORCE}$:Bool

$\mathcal{R} \leftarrow Rectangle(\mathcal{M}.depth, \mathcal{M}.width)$;

$\mathcal{S} \leftarrow \emptyset$;

$\mathcal{L} \leftarrow [\mathcal{R}]$;

*/*initialize $\mathcal{L}$ with the rectangle corresponding to $\mathcal{M}$ */*

while ($\mathcal{L} \neq \emptyset$) **loop**

$\mathcal{R} \leftarrow \mathcal{L}$.*first()*;

$\mathcal{P} \leftarrow random\_port(\mathcal{B})$;

$\mathcal{U} \leftarrow \mathbf{port\_usable}(\mathcal{P}, \mathcal{M})$;

$\mathcal{FORCE} \leftarrow$ (*max_fails_reached*);

$\mathcal{C} \leftarrow \mathbf{carve\_rectangle}(\mathcal{R}, \mathcal{P}, \mathcal{FORCE})$;

if ($\mathcal{FORCE}$ **or** (is_valid($\mathcal{C}$) **and** $\mathcal{U}$)) **then**

add_new_subrect ($\mathcal{R}, \mathcal{C}, \mathcal{L}$);

$\mathcal{L} \leftarrow \mathcal{L} - [\mathcal{R}]$;

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{C}\}$;

end if

end while

end

Figure 3.4: *Algorithm for Mapping a Logical Memory*

Algorithm: Carve_Rectangle

Input:

$\mathcal{R}$:Rectangle to map

$\mathcal{P}$:Physical Port

$\mathcal{FORCE}$:Bool

*/*indicates if mapping to be done forcefully*/*

Output:

$\mathcal{C}$:Rectangle

/ Carved from R such that it can be mapped onto $\mathcal{P}$ */*

begin

$Fitness$:Array of fitness, one value for each configuration

$\mathcal{N}$:A subset of Port Configurations

$\mathcal{N} \leftarrow \{n | n \in Configurations of \mathcal{P} \wedge n.width \geq \mathcal{P}.curr\_config.width\}$

for each($n \in \mathcal{N}$) **loop**

$get\_assignable\_rect(\mathcal{R}, \mathcal{P}, n)$;

$Fitness_n \leftarrow \mathbf{func}(assignable\_depth, assignable\_width, \mathcal{R})$;

$save\_best\_assignable\_rectangle()$;

end for

if ($BestFitness > 0$) **then**

$\mathcal{C} \leftarrow Rectangle(best\_assignable\_depth, best\_assignable\_width)$;

$\mathcal{P}.curr\_config \leftarrow best\_config$;

else

/ Failed to carve out a rectangle */*

if ($\mathcal{FORCE}$) **then**

/ Forcefully assign $\mathcal{R}$ to $\mathcal{C}$ */*

$\mathcal{C} \leftarrow \mathcal{R}$;

else

/ Try in future iterations */*

$\mathcal{C} \leftarrow \mathcal{NULL}$;

end if

end if

end

Figure 3.5: *Algorithm for Carving a mappable sub-rectangle from a given rectangle onto the given port*

Algorithm: add_new_subrect

Input:

$\mathcal{C}, \mathcal{R}$: Rectangle

$\mathcal{L}$:List of Rectangles to map

Output:

$\mathcal{L}$:Modified List of Rectangles to map

begin

$\mathcal{N}$: Rectangle, newly created to append to L

if ($\mathcal{C}.width < \mathcal{R}.width$) **then**

$\mathcal{N} \leftarrow$ Rectangle ($\mathcal{C}.depth, \mathcal{R}.width - \mathcal{C}.width$);

$\mathcal{L} \leftarrow \mathcal{L} \cup [\mathcal{N}]$;

end if;

if ($\mathcal{C}.depth < \mathcal{R}.depth$) **then**

$\mathcal{N} \leftarrow$ Rectangle ($\mathcal{R}.depth - \mathcal{C}.depth, \mathcal{R}.width$);

$\mathcal{L} \leftarrow \mathcal{L} \cup [\mathcal{N}]$;

end if;

end

Figure 3.6: *Algorithm for Generating Rectangles from unmapped part of another rectangle after carving has been done*

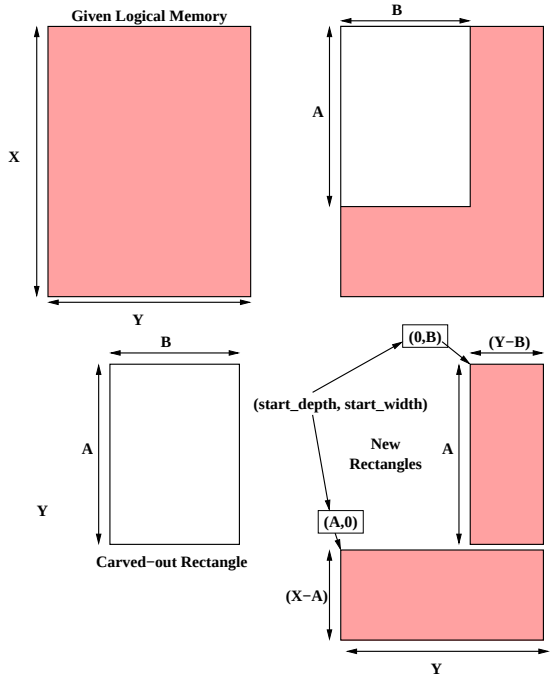


Figure 3.7: *Rectangle Carving Process*

$$bits\_waste = (config\_width - assigned\_width) * assigned\_depth$$

$$pct\_waste = (bits\_waste + prev\_bits\_waste)/total\_bits$$

pct_depth and *pct_width* represents percentage of depth of the rectangle under consideration which could be mapped for a given configuration of the port. pct_waste is the percentage of storage space which will go to waste if a sub-rectangle is mapped with the current configuration. Note that prev_bits_waste represents the storage space wasted due to already mapped sub-rectangles to this port. Let $\mathcal{I}$ be the factor which we want to maximize and $\mathcal{D}$ be the factor we want to minimize. Then we formulate as follows

$$\mathcal{I} = pct\_depth + floor(pct\_depth) + 2 \times pct\_width + 2 \times floor(pct\_width)$$

$$\mathcal{D} = pct\_waste$$

$$Fitness = \mathcal{I}/(6 + \mathcal{D})$$

Function *floor(pct_depth)* will be 1 only when the complete depth has been assigned to this port, thus rewarding such solutions. Same is applicable for *pct_width*. Note that a factor of 2 has been used to give higher weight to solutions mapping more width than solutions mapping more depth. The numerator varies from 0 to 6, while denomenator ranges from 6 to 7. The fitness varies in the range 0.0 to 1.0. Thus, the fitness value is more sensitive to the factors in $\mathcal{I}$ than $\mathcal{D}$ thus prefering solutions with better mapping over solutions with lesser wastage. In case of a tie between two configurations in terms of the quality of subrectangle carved out (e.g. $\mathcal{I}$ will be 1.0 if the rectangle is small enough to fit completely for more 2 or more configurations), we prefer a configuration with lesser wastage. We also tried another fitness function given below :

$$Cost = 1 + \mathcal{I} - \mathcal{D}$$

$$Fitness = 1/(1 + Cost)$$

However when run for same number of iterations, the tool gave a much deteriorated solution with the second cost function than with the previous one. The logical memory segments were much more fragmented as splitting into smaller parts leads to solutions with lesser wastage of storage space. Therefore, the second fitness function is not used.

It is possible that another subrectangle has already been mapped to the port under consideration. In such cases, we consider only those configurations which have width larger than that required by the already

mapped subrectangle. This is in tune with the idea of keeping the heuristic constructive. If no rectangle is assigned previously to this port, all configurations can be considered. *carve_rectangle* determines the configuration for which the rectangle is mapped with best fitness. If the carved-out rectangle is smaller that the given rectangle, at most two new rectangles are created out of the unmapped parts and added to the list of rectangles yet to be mapped. Figure 3.7 shows two new rectangles being created. However, if either depth or width of the carved out rectangle is equal to that of the original rectangle, only one new rectangle will be created. While creating the new rectangles, we intuitively try to keep all bits of a word in the same rectangle, i.e. depth-wise splitting is preferred over width-wise splitting. The port's configuration is updated to accommodate the new as well as any old rectangles mapped to this port.

An important characteristic of the algorithm is that it does a constraint satisfying mapping as far as possible. The other option was to ignore constraints violation during mapping stage and then penalize any violation heavily during evaluation. Although it is unavoidable to come across constraint violating solutions, our *constructive approach* reduces their occurances.

Complexity Analysis: *port_usable* looks at all the sub-rects mapped to a port to compare if a sub-rect of the current LM has already been mapped to it. In worst case, all LMs would have at-least one sub-rect mapped to this port and *#LogMem* checks will be performed. *Carve_Rectangle* calculates the fitness of mapping for each configuration. In the worst case, it will calculate fitness for each configuration of the port. The *while* loop in *Map_LM* will iterate for (#ports + k), where *#ports* is the total number of ports present over all instances of mem_type. The bounding is due to upper limit on *max_fails*. All other operations are constant time. Thus, the overall complexity of mapping a LM to mem_type *t* is $\mathbf{O}(\#\mathbf{Ports_t} \times (\#\mathbf{LM} + \#\mathbf{config_t}))$.

3.6 Control Logic

The process of memory synthesis can be divided into two stages : *Memory Mapping* and *Control Logic Generation*. Mapping involves assigning all logical memory segments onto some physical memory instance on the board. We have defined mapping in Section 3.2 and the process of mapping in Section 3.5. However, any memory mapping requires some additional control mechanism to ensure proper functioning of the design. The complexity of control logic can pose significant limitations on the way mapping can be performed. There is an inverse relationship between the restrictions applied to mapping process and the complexity of control logic. The gains in mapping process can be easily undone by the associated control logic it might warrant. This requires a careful tradeoff analysis. We have identified two constituents of the control logic. *First*, the logic required to handle conflicts due to parallel memory accesses by more than

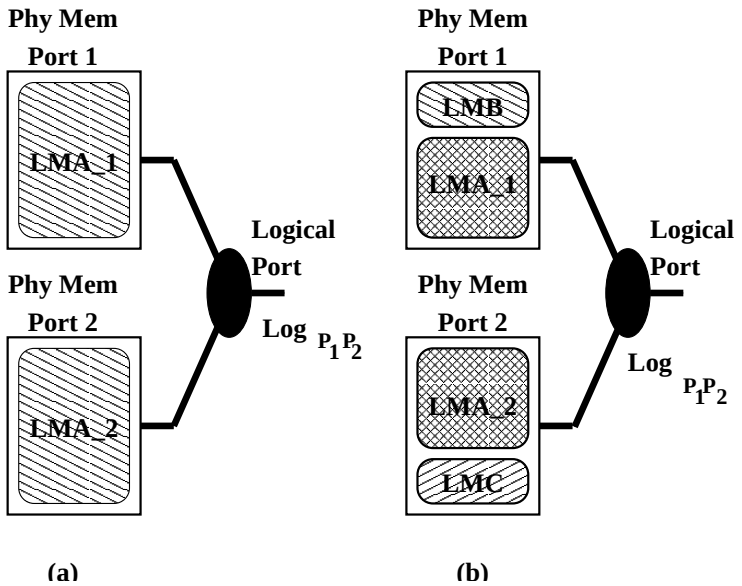


Figure 3.8: *Logical Port*

one tasks. This is tackled by introduction of *arbiters*. *Second*, making the details of mapping transparent to the accessing logic. A *address translation and enable logic* unit is introduced for this purposed.

3.6.1 Arbitration

If a logical memory has more than one accessing task, an arbitration logic is required to serialize any potential parallel accesses to that logical memory. Since we do not consider the life-times of the logical memory inside various tasks, it is assumed that all accessing tasks can access at the same time. Thus all the accessing tasks have to be arbited. If more than one logical memories are sharing the same port, tasks accessing either of them will have to be arbited.

Partitioning a large LMs across multiple ports is permitted. It is also possible that a LM might share different ports with different LMs. To handle this situation, we introduce the concept of *logical ports*. Each logical port is a group of physical ports, access to which need to be arbited by the same arbiter. Let $share(port_i, LM\_list)$ represent a group of LMs, in *LM_list*, which share port $port_i$. Consider the scenario given in Figure 3.8 (b).

$$share\ (port_1, [A, B])\ and\ share\ (port_2, [A, C])$$

Here, parts of LM *A* have been mapped to $port_1$ and $port_2$. To take a simple case, we assume that LM *B* and LM *C* are not split and are completely mapped to $port_1$ and $port_2$ respectively. Ideally, parallel

access to *B* and *C* should be allowed. However, *LM A* shares a port with each of them. Therefore, access to $port_1$ has to be arbited between accessors of LM A and B. Similarly, access to $port_2$ has to be arbited between accessors of LM A and C. We form a **logical port** log_{p1p2} by combining $port_1$ and $port_2$. This forces accessors of all A, B and C to be arbited together. Consequently, we have a logical port, such that, *share*(log_{p1p2}, [A,B,C]). Note that if even if the same compute task is accessing more than one logical memory at a logical port, for arbitration purpose, it is still counted separately for each logical memory it is accessing. Thus, the number of tasks arbited at a logical port is equal of sum of tasks accessing the constituent LMs at that port. Recall that a LM is split only across ports of same memory type. Since LM A is spilt across $port_1$ and $port_2$, they should belong to same memory type. The arbiter is synthesized in the *local_pe* of this memory type.

For formation of logical ports, we view the design a *hypergraph*[1] [30]. Each logical memory is a node of this hypergraph. A physical port which has more than one logical memory mapped to it is a hyperedge. Logical memories which are mapped to different types of physical memories will never share a port. Thus the hypergraph will have multiple disjoint parts. If no port is shared, there will be disjoint parts for logical memories mapped to same physical memory type. We perform a depth-first traversal of this hypergraph and form *equivalence classes*. All nodes (logical memories) which are directly or indirectly connected to each other through a hyperedge are part of the same equivalence class. Finally, each equivalence class forms logical port.

A scalable arbiter logic for reconfigurable architectures is presented in [22]. The arbiter grants access to tasks in a round-robin fashion. For each logical memory accessed by the task, a pair of *request-acknowledge* signals is introduced between the task and the arbiter. When the task wants to access the memory, it asserts the *request* signal and then wait for the arbiter to grant the access. *First*, the *arbiter* determines if the current accesses to the memory is complete. Then, if either no other task has requested for the memory access or it is turn of this task to access, it asserts the *acknowledge* signal for that task. Now, the task performs the memory access operations. Once it is done doing so, it de-asserts the *request* signal indicating the *arbiter* that access can be granted to some other task. When not accessing or while waiting for *acknowledge* signal from the arbiter, the task *Z*-out all *address* and *data-out* signals.

3.6.2 Address Translation and Enable Logic

An address translation mechanism is required to handle the *mismatch* between the *logical address* of the word being accessed and the *physical address* location where that word is placed. In this discussion, we assume that every physical memory instance has only one port. It implies that at every physical port,

[1] A hypergraph is graph in which a hyperedge connects arbitrary number of nodes, rather than just two nodes as in regular graph

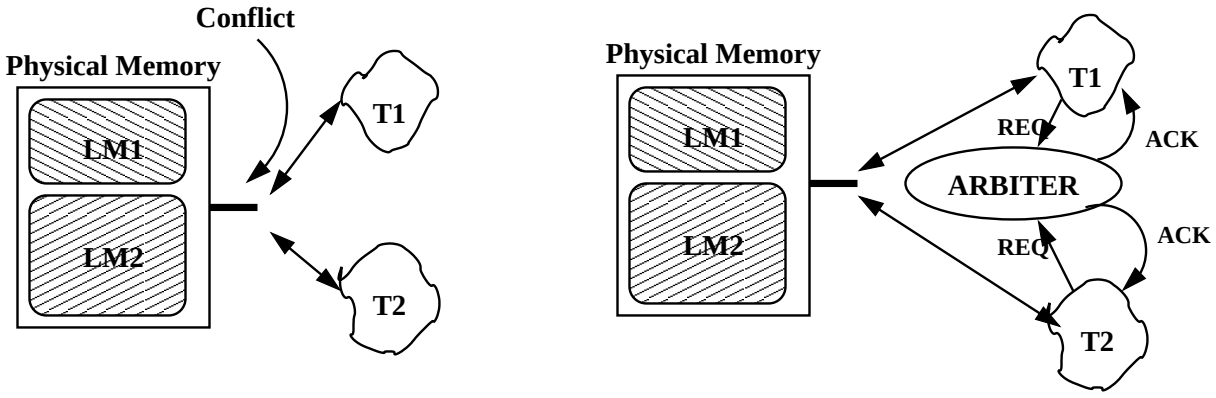


Figure 3.9: *Memory Access Conflict resolution using Arbiters*

starting address location has a physical address of 0. This assumption is later relaxed.

A mismatch can arise under two cases. *First*, if a LM is split across multiple ports, then at all logical addresses where the LM is split, a non-zero logical address is be mapped to physical address location 0 of the new port causing a mismatch. If a logical memory has been split across *p* ports, there will be *p* address translation and enable logic units required, one for every (LM, P) pair. *Secondly*, if multiple LMs are sharing a port, then only one LM can be placed starting at physical address location 0 of the port. All other LMs will be placed at an *offset*. This can also lead to mismatch. Thus one address translation unit is required for each logical memory mapped to a shared port. Consider the example in Figure 3.10. Logical depth 128 to 191 of LM A has been mapped to physical location 448 to 511. We call 128 the *logical_starting_address* and 448 the *physical_starting_address* of LM A on port P_1. Address for LM A going to port P_1 has to be adjusted with a **constant offset** to match the logical address produced by the accessing task with the physical address location where the word being accessed is present. This offset is given by

$$\mathbf{lm\_port\_offset} = \mathbf{physical\_starting\_addr} - \mathbf{logical\_starting\_addr}$$

$$\mathbf{phy\_addr} = \mathbf{addr\_from\_task} + \mathbf{lm\_port\_offset}$$

Note that the *offset* value will be negative when *physical_starting_address* is smaller than the *logical_starting_address*.

The address after translation is given to the input of a tri-state buffer, output of which goes directly to the address bits of the port. The enable of this tri-state buffer is controlled by the lm_port_enable signal. We now describe the generation of enable signal.

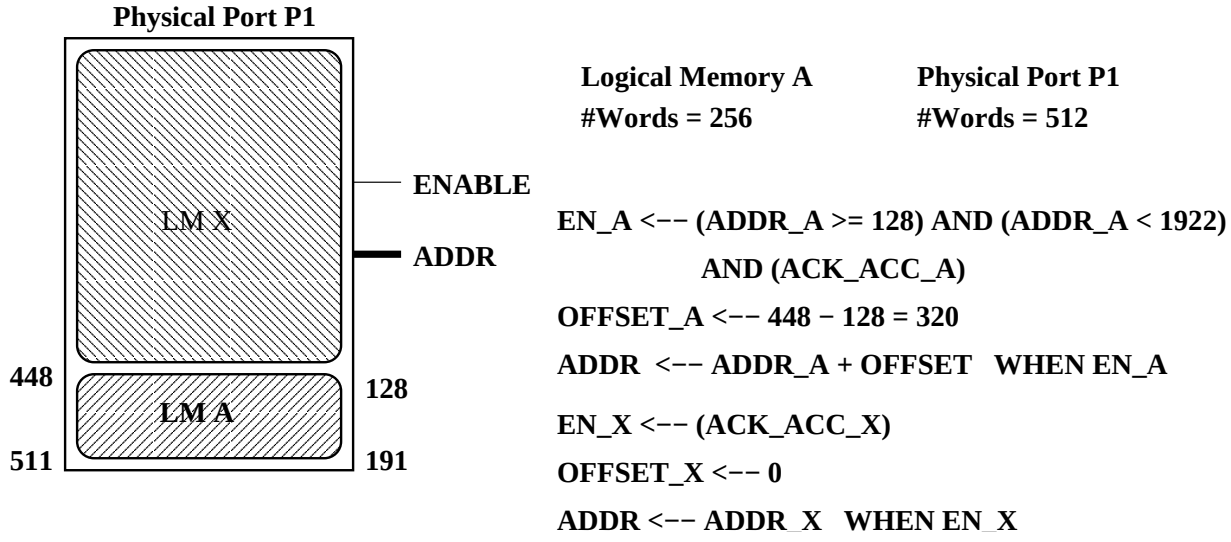


Figure 3.10: *Address Translation and Enable Logic*

Again, consider again the example given in Figure 3.10. LM A is 256 words deep. However, only 64 words, from 128 to 191 are mapped to a particular port P_1. The total words available at this port is 512. Remaining 448 words are occupied by some other LM, say X. The tristate buffer at the output of address translation unit for LM A at port P_1 should be enabled only when an access is being performed on LM A and the word being accessed is mapped to port P_1 i.e. address is in range 128 to 191. During a clock cycle, say if word *001* of A is being accessed, this port should be disabled. If any task is accessing LM A, the arbiter controlling this port would have granted *acknowledge* signal to that task. Thus by **OR***ing* acknowledge signals of all the tasks accessing LM A, it can be determined if the LM is being accessed during a particular clock cycle or not. The address value for LM A is compared with the valid lower and upper limits of logical address on P_1 to ensure that the address is indeed of a word mapped to this part. Note that if these limiting address values are equal to some power of 2, the size of comparators can be very small. The scheme shown in Figure 3.11 generates a valid enable signal for the address tristate buffer.

If the LM is not partitioned, *no enable logic* is required. If any logic is accessing the unpartitioned LM, the only port to which it is mapped will always be enabled every time a task accesses the LM. There is an implicit assumption that the accessing tasks will always generate valid addresses. For the subrectangle representing the mapping of part of LM to a port, if *subrect_start_depth* is 0, then lower limit comparison is not required. Similarly, if (*subrect_start_depth*+*subrect_depth*) is equal to total depth of the LM, higher limit comparison is not required.

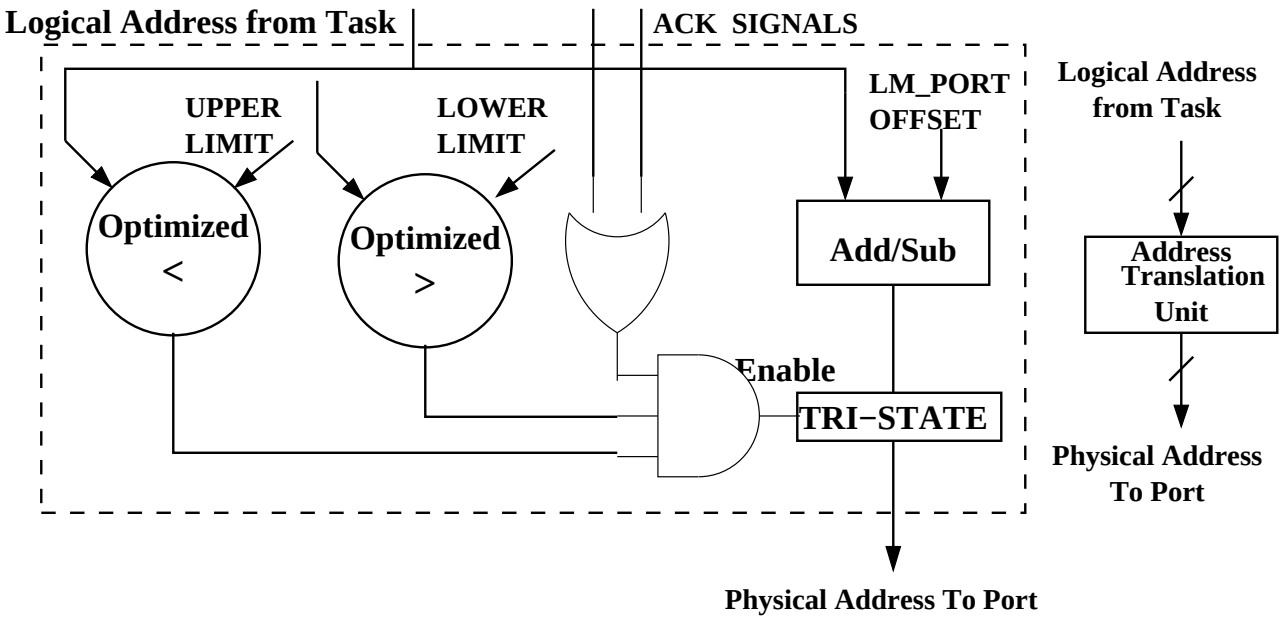


Figure 3.11: *An Implementation for the Address Translation and Enable Logic Unit*

3.6.3 Handling Multiple Ports

The discussion so far assumed that all physical memories are single-ported. This was reflected in the assumption that the first physical location at every port has address 0. However, when a physical instance has multiple ports which share the same storage space, this is no longer true. In our work, we do now allow access to the same *bits* through multiple ports. Thus, the control logic should also ensure mutually exclusive access of storage space by multiple ports present on same physical instance.

We observe that the address translation unit ensures that the addresses reaching any port will lie in certain range by doing a check on the upper and lower limits of the address values. Thus, the maximum address value which can appear at a port can be predetermined. If we add the maximum address value for $port_0$ of an instance as an offset to address at $port_1$ of that instance, we can ensure that the two ports will not access the same storage space. Care has to be taken about different configurations for each port. If $port_0$ has maximum possible address of 63 for a configuration with width 4, it will translate into a maximum address of 127 for configuration with width 2. Figure 3.12 shows the algorithm for generating port offset. The addition of port offset can be merged with the addition of lm_offset logic inside the address translation unit. Thus, no additional stage is required and hence no additional delay is incurred.

Algorithm: Assign_Port_Offest

Input:

$\mathcal{M} : Arrayofmaximumaddressesatport$

$\mathcal{W} : Arrayofwidthofassignedconfigurationstoports$

$\mathcal{N} : Numberofportsatgivenphysicalinstance$

Output:

$\mathcal{O} : ArrayofOffsetaddressesforports$

begin

$BitsConsumedSoFar : int := 0;$

$CurrPort : int := 0;$

while $(CurrPort < \mathcal{N})$ **loop**

$O[CurrPort] \leftarrow (BitsConsumedSoFar/W[CurrPort]);$

$BitsConsumedSoFar \leftarrow BitsConsumedSoFar + (M[CurrPort] * W[CurrPort]);$

$CurrPort \leftarrow CurrPort + 1;$

end while

end

Figure 3.12: *Algorithm for assigning offset to ports of a multi-port physical memory*

3.6.4 Putting It Together

Figure 3.13 shows how all the control logic functions. A compute task has address/data ports for each logical memory it accesses. If the access to logical memory is arbited, the task also has a pair of *request/acknowledge* signals for that logical memory. We assumed that when the task is not accessing a particular logical memory, or is waiting for acknowledge signal from the arbiter, the address ports of the task are driven to **Z** (tri-state), a *default-Z* situation. All compute tasks accessing a LM drive inputs of all the address translation units for that LM. Thus, multiple tasks can be driving the input of the address translation unit. However, the arbiter ensures that only one of them drives the input while all others drive their ports to **Z**. This logical address for the LM is translated into the physical address location where the desired word is present on that port. If the address after translation is valid, the tri-state logic (shown in Figure 3.11) is enabled and the address is sent to the port. Again, output of address translation units of all the LMs mapped to this physical port drive the address bits of the port. However, the only one of them will drive a valid address while others will drive **Z**. The *data-in* and *data-out* ports of the accessing tasks are directly connected to that of the physical ports.

The case where ports of the task have *default-0* or *default-1*, appropriate *multiplexer* logic will have to be inserted at the input of address translation unit and the acknowledge signals will have to combined to generate select signal for this multiplexer.

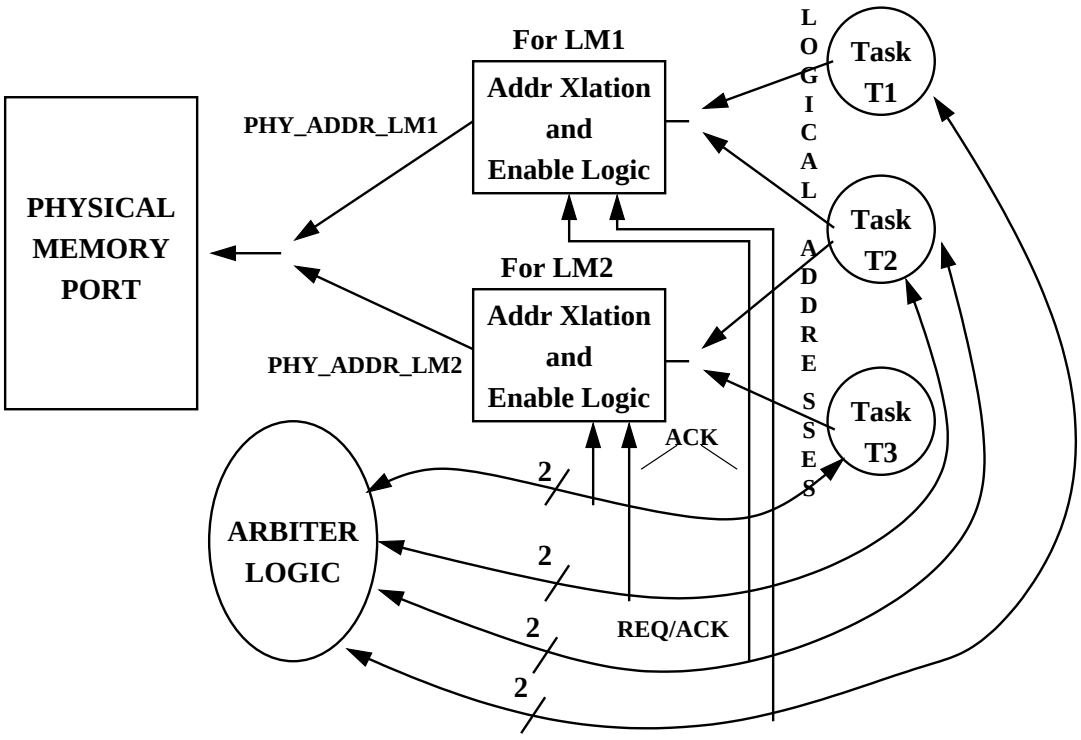


Figure 3.13: *The Overall Scheme*

3.7 Constraints

Constraints for the memory mapping algorithm can be classified into two categories : *Architectural* and *Design* Constraints. Architectural constraints are dictated by the RC onto which the design has to be mapped. On the other hand, design constraints are imposed by the user to suit the application.

3.7.1 Architectural Constraints

Architectural constraint is a set of *hard* constraints. In other words, if these constraints are not met, the design can not be executed on the given RC board.

- **Physical Memory Sizes**

  Each physical memory instance has a capacity, specified in the physical bank type to which it belongs. The mapper should ensure that the sum of bits consumed by all logical memories assigned to a physical instance does not exceed the maximum capacity of that instance.

$$MemoryPenalty = \sum_{i=0}^{num\_phy\_inst-1} \frac{\lceil occupied\_bits - available\_bits \rceil}{available\_bits}$$

  The operator $\lceil x \rceil$ represents the non-negative ceiling, i.e. there is no penalty if occupied bits are less than the available bits.

- **Interconnect Constraints**

  The mapper should ensure that there are enough pins available between each FPGA pair for routing all the address, data and arbitration signals. If there is only one FPGA on the board, this constraint does not exist.

$$InterconnectPenalty = \sum_{\forall f_1, f_2}^{0<f_1,f_2<num\_fpgas} \frac{\lceil occupied\_width_{f_1 f_2} - available\_width_{f_1 f_2} \rceil}{available\_width_{f_1 f_2}}$$

  Again, the operator $\lceil x \rceil$ represents the non-negative ceiling, i.e. there is no penalty if occupied width is less than the available width.

- **Mapping Constraints**

  Mapping Constraints are a reflection of the limitations in the ways in which a logical memory can be mapped. We require that two sub-rectangles of the same logical memory should not be mapped to the same port. If the memory had been split *width-wise*, different bits of the same word will be

present in two sub-rectangles. If these two sub-rectangles are mapped to the same port, they can not be accessed during the same clock cycle. The specified constraint avoids such situations.

$$MappingPenalty = \sum_{\forall p \in ports} \frac{\sum_{l \in LM\_mappedto\_p}(\#subrects\_of\_l\_mapped\_to\_p - 1)}{\#total\_subrects\_mapped\_to\_p}$$

3.7.2 Design Constraints

The user has the option of imposing certain additional constraints to ensure mapping being done in certain ways he or she might desire. This can play an important role in the quality of final design. It can also be looked upon as broad guidelines being provided by the user based on design requirements.

- **Sharing of ports between Logical Memories**

  The user can specify whether the mapper should assign more than one logical memory to the same port. Sharing of ports between logical memories can lead to better utilization of storage capacity available in big physical instances. It also leads to address and data bus sharing between various logical memories. However, the number of compute tasks which can parallely access this physical port is now the union of compute tasks accessing all the logical memories accessed through this port. Thus, the number of compute tasks which need to be arbited can be more. This will introduce additional delay in memory access due to increased arbiter size [22]. Moreover, extra arbitration signals need to be routed. The bigger arbiter will also consume more area on the FPGA devices.

- **Latency Constraint**

  The user can specify an upper limit of the overall read/write latency of the design. This can force the mapper to put more logical memories together overriding increased arbitration cost. The memory mapper should ensure that this constraint is satisfied, otherwise return with a failure.

- **Splitting LM across different memory types**

  A large LM can be split across different memory banks which have same read and write latencies. If these memory banks are not connected to the same FPGA, routing and arbitration can be highly complicated and the quality of the mapped design will be very poor. We **do not** allow this option in our technique.

For each constraint violated, we can quantize it and add an equivalent penalty. The mapper continues until a solution with no penalty is found or returns with failure.

#Tasks arbited	Area (#Slices)	#Tasks arbited	Area (#Slices)
2	5	9	59
3	8	10	78
4	15	11	88
5	22	12	99
6	26	13	126
7	40	14	142
8	50	15	175

Table 3.1: *Area occupied by arbiters of different sizes*

3.8 Cost Function and Estimation

At every iteration, the Tabu Search evaluates the solution by calculating a cost for the current solution. We present the factors which constitute the cost and the process of estimating the cost.

- **Read/Write Latency**

  The read and write latency of the design can be estimated based on the memory bank to which various logical memories are mapped. For an accurate calculation, we need to know the number of times each logical memory is read from and written to during execution. In the absence of this information, we consider the number of words in the LM to be a rough estimate of number of accesses. Total latency is given by the following formula :

$$Latency = \sum_{i=1}^{N} ((NumReads_i \times RdLat_{Bank_i}) + (NumWrites_i \times WrLat_{Bank_i}))$$

- **Arbitration Cost**

  As explained in Section 3.6.1, the number of tasks which need to be arbited at any *logical port* is equal to the sum of number of tasks accessing the logical memories. Table 3.8 shows the areas of arbiters of different sizes synthesized for Xilinx Virtex architecture. The values are obtained by taking the arbiter design through *Synplify 5.2.2* [29] and *Xilinx M1.5.25 PAR (Place and Route)* [36]. Xilinx Virtex XVC1000 devices has more than 12000 Slices. An arbiter of size 7, having area of 40 Slices, is less than 0.33 % of the total devices area !

- **Clock Frequency**

  More and more dedicated resources are being provided for *on-chip* routing of signals, e.g. *versaring* on *Virtex* architectures. Any signal which needs to be routed across chips becomes the bottleneck

in operating at high frequency. We take that maximum number of pins traversed by a signal in the design to be an indication of maximum operating clock frequency of the design. The two values are inversely proportional. If logic partitioning has already been performed on the design, the mapper tries to place logical memory into physical instances which are local to the accessing FPGAs. The number of pins traversed to reach from a physical memory port to its local_pe and the number of pins traversed to reach from one PE to another are supplied as input to the algorithm. For logical memory *m* and compute task *t*

$$\#pins\_trav_{mt} = \#pins\_trav\_from\_localpe_m + \#pins\_trav\_from\_localpe_m\_to\_fpga_t$$

$$Max\_Clk\_Freq = \frac{k}{\mathbf{Max}_{\forall m,t}(pins\_trav_{mt})}$$

- **Blocks Processed**

  Block processing means processing on multiple sets of data, one after the other. The concept, and the methodology for automated generation is explained in [10] in detail. A counter is synthesized on each FPGA. Once, the design has finished processing one set of data, the counter is incremented. The bits of the counter are simply prepended to the bits of physical address to form larger size address. The counter serves to move the offset address for each block of data. From memory mapper's view point, this translates into doing mapping in such a fashion that the maximum percentage of occupied depth across all physical instances is minimized. This would maximize the number of blocks of data which can be processed. For each physical instance, if any logical memory is assigned to it, we can calculate the number of blocks of data that can be placed in that physical instance by the following formula

  $$PhyMem\_Blks\_Processed = \frac{Total\_Bits\_Available}{\lceil Bits\_consumed \rceil}$$

  where operator $\lceil x \rceil$ stands for rounding to the next power of 2. Taking a ceiling allows us to simply prepend the counter bits. We also take advantage of all the unused instances of the same type. If there are many physical instances of same type which are not used at all, they can be combined with used instances to *form* bigger physical memories. Note that we can only consider unused physical instances of same type for combining together.

  $$MemType\_Blks\_Processed = \frac{\#Total\_Instances\_Available}{\#Instances\_Occupied}$$

  Thus the number of blocks that can be stored in a particular type of physical memory is the product of the minimum blocks possible in a physical instance of that type and the MemType_Blks_Processed

for that type. The overall number of blocks processed is the minimum number of blocks that can be processed for a bank type.

$$\mathbf{Min}_{\forall t \in MemType}(\ MemType\_Blks\_Processed_t \times \mathbf{Min}_{\forall i \in t}(PhyMem\_Blks\_Processed_i) \ )$$

- **Address Translation and Enable Logic**

  For each port to which part of a LM is mapped, an address translation unit is required. The size of the logical address bus as well as that of the offset is known. Thus the total area required can be estimated to be the sum of area requirements of each of the constituent components, namely, 2 comparators, an adder/subtracter and tri-state buffer.

  Since comparison is always done with an already known constant, it can be easily optimized. If the constant is such that its last *k-bits* are equal to 0, and the logical address is *n-bits* wide, the comparator only needs to compare *n-k* most significant bits of the address. For example, if the total logical address range is 0-31 or 5 bits and it is to be compared again a constant value, say 8. Since last 3 bits of value 8 are 0, we need to compare only the 2 MSBs. Thus if $Addr(5:4) > \text{“}00\text{”}$, then the address is greater than 8, otherwise not.

- **Address and Data Bus Routing**

  Access to logical memories is governed by the way their corresponding physical memories have been grouped into *logical ports*. Since only one LM can be accessed through a logical port at a time, this is exploited to share address and data buses between them. For each FPGA accessing at least one LM of a logical port, address and data bus need to be routed to that FPGA. The number of bits will depend on which LM that FPGA is accessing. If memory mapping is done in the absence of logic partitioning information, mapper does not have the information as to which physical instance's signals will need to be routed to other FPGAs. In such cases, we assume that all address and data buses might need to be routed to all FPGAs and therefore try to minimize the total number of signals required for routing address and data bus.

The overall cost function is a weighted sum of all the above factors.

3.9 Experimental Results

Table 3.2 shows results of memory mapping for some benchmark examples. The target architecture is shown in Figure 3.14 is assumed to have one FPGA. There are 3 instances of on-chip memories of size 4096 bits each, with 5 configurations varying in width from 1 to 16 and a read/write latency of 1 clock cycle

Design Name	#Log. Mem.	Total Size	R/W Latency	Split Cost	Share Cost	#Ports Used	Num Addr Data Pins	Exec. Time(s)
DCT1	15	112	256	0	13	2	22	17.9
FFT	12	80	160	0	10	3	41	15.0
DCT2	10	48	96	0	8	5	50	15.4
Laplace	14	509	1018	0	12	4	54	17.0
MeanValue	13	1200	2400	2	12	4	28	17.4
LUD	13	1343	2686	3	15	4	15	17.4
Rand100	100	3513	11038	5	100	7	28	142

Table 3.2: *Results of Memory Mapping for benchmark examples*

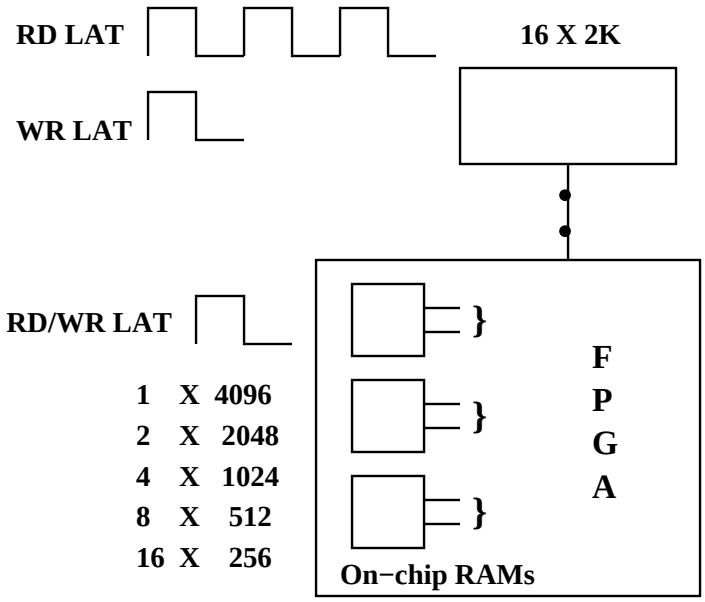


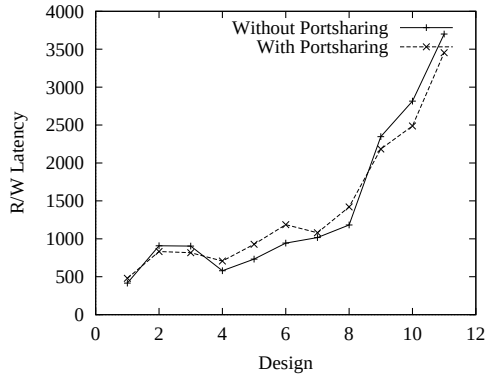
Figure 3.14: *Target Architecture for Examples*

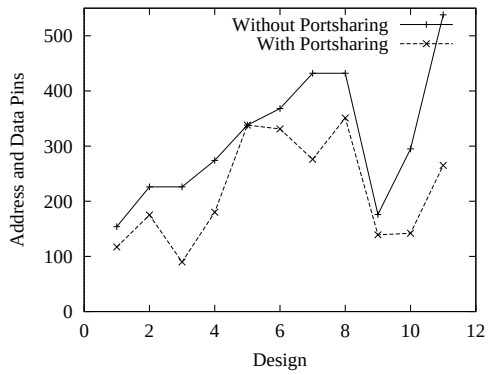
each. There is one instance of off-chip memory having 2K words, 16-bit wide, single-ported with read latency of 3 and write latency of 1. The total number of words in all logical memories is given in Column 3. If no logical memory is split, *Splitting Cost* is 0. *Sharing Cost* indicates whether multiple logical memories were mapped to same port or not. As more logical memories share physical ports, the size of logical_ports formed increases. This leads to sharing of address and data buses, thus lower Addr/Data pins requirements. Similarly, splitting causes formation of bigger logical_ports leading to sharing of Addr/Data pins.

We now present graphs showing variations in the results for four different runs of the tool. There are variations in design constraints and in the cost function. The results are for some synthetic examples. The design and the architecture remains the same for each run. All the results present here assume a single FPGA board architecture. Thus, the results shown present only the memory mapper without direct effect of other factors like FPGA area or interconnect constraints. The case when port sharing is permitted, bigger logical ports are formed, leading to sharing of address and data pins between various logical memories. This enables it to find a constraint satisfying solution very quickly. In fact, for design 9, 10 and 11, no constraint satisfying solution was found if port sharing is not allowed. The total number of **address and data pins is always lower for port sharing** case.

However, allowing port sharing greatly increases the solution space to be explored as many more solutions now constitute a valid mapping. The objective function has an additional constituent in the form of cost of port sharing. More the port sharing, more will be the serialization of memory accesses and bigger will be the size of arbiter required. The advantage is a decrease in latency and address/data pins used. Thus we see two balancing factors in the form of latency and port-sharing cost **pulling** the search process in **opposite directions**. Ideally, we would expect the algorithm to do portsharing to map more logical memories to faster physical memory types, thus lowering the overall read/write latency of the design. However, due to two main factors, namely, increase in the solution space and opposing factors in the objective function, we observe the search process returning latencies which are higher than those in the case of no-portsharing. The same algorithm run with another objective function, which gives much higher weight to design latency, shadowing the effect of port-sharing cost, guides the search in one direction. This leads to overall better solutions. Thus with the second objective function, the search process always returns designs with lower latencies for port-sharing as compared to noport-sharing. This illustrates the sensitivity of the search process to the objective function.

The graphs in Figure 3.15 compare port-sharing and noport-sharing solutions obtained with a balanced objective function. The graphs in Figure 3.16 show the comparison with the objective function which gives very high weight to latency.





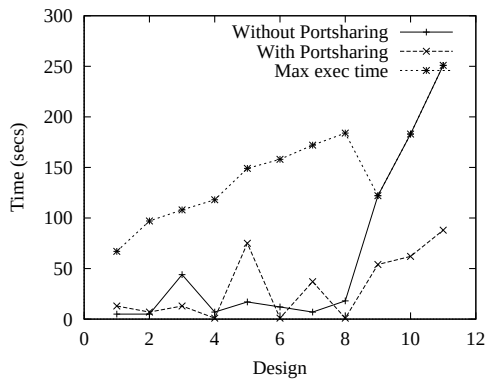
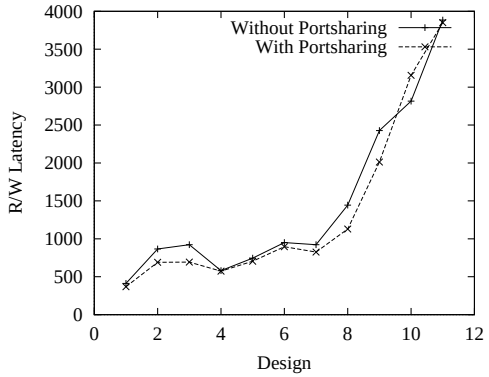
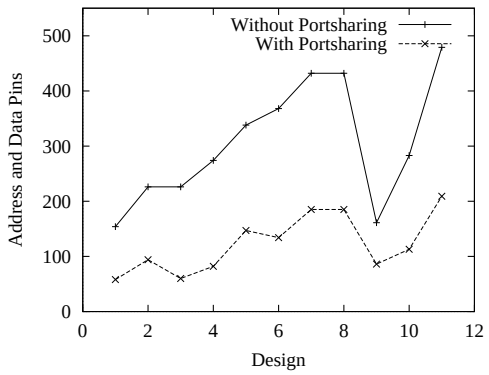


Figure 3.15: *For balanced cost function*





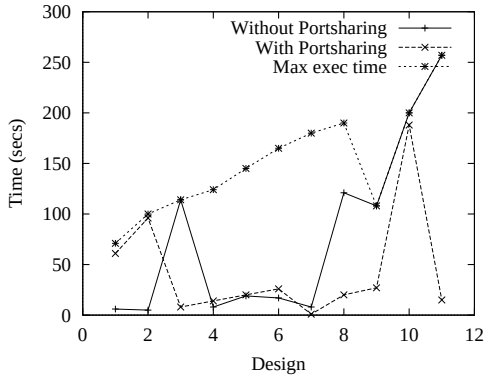


Figure 3.16: *For unbalanced cost function*

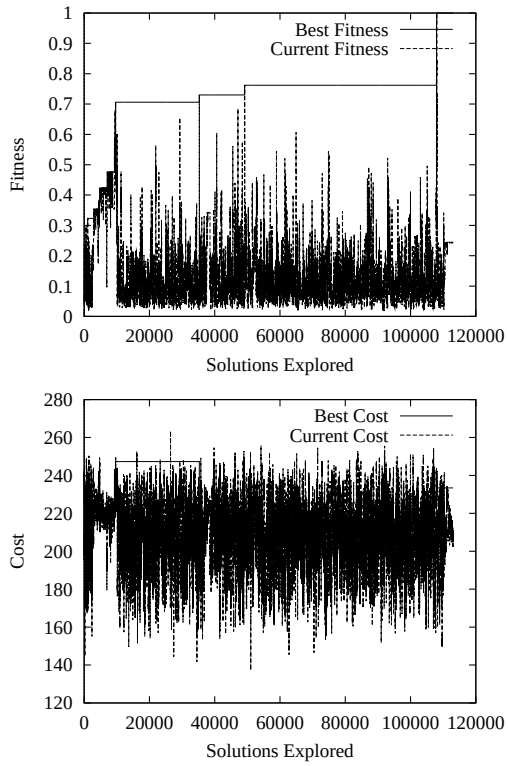


Figure 3.17: Trace for no portsharing

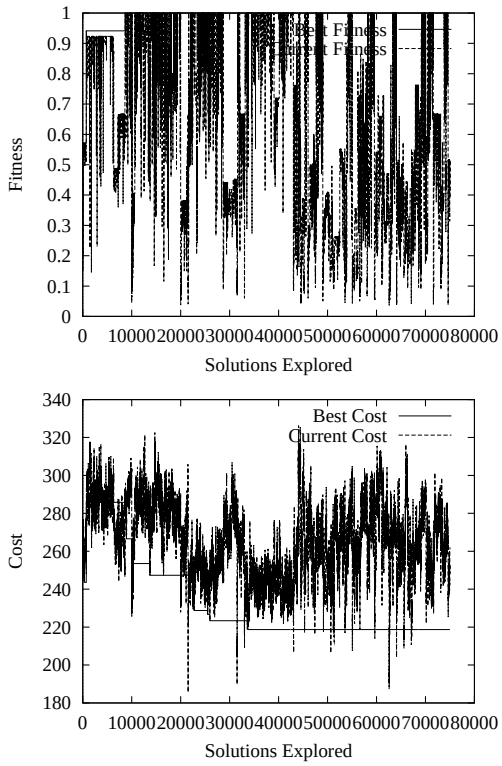


Figure 3.18: Trace for portsharing

The graphs in Figure also plot the time at which the search process found its last best solution. Note that we continued to run the search, but after this time, no better solution was found. The *Max Exec Time* is total the execution time for the portsharing case. Both techniques, were run for same number of iterations and take roughly the same time. Thus, *Max Exec Time* is a very good indication of the maximum execution time for both cases. However, in cases where the search could not find a valid solution, it continues for 3 times the number of iterations. This happened for design 9, 10 and 11 in noportsharing case. We observe that in all cases considered, if the search is not able to find a solution in the original stipulated time, it does find solution even in extended time.

We also trace of some values during the progress of Tabu Search. Figure 3.17 shows trace for case when no portsharing is permitted between various logical memories, while Figure 3.18 shows trace when port sharing is permitted. One value is the *Fitness* of the solution while the other is the *Cost* of the solution. A *Fitness* of 1 indicates that the solution meets all constraints. *Cost* is the value of the objective function. We trace the *Current Fitness* and the *Best Fitness*. As the search progresses, Best Fitness increases and finally becomes 1 when a constraint satisfying solution is found. We also show a trace of the *Current Cost* and the *Best Cost* for all visited solutions. The search heuristic always prefers a better fitness solution to a better cost solution. Thus, as long as Best Fitness is less than 1, the Best Cost is not the lowest cost found

so far, but the cost of the solution having Best Fitness. Once a constraint satisfying solution is found, Best Fitness can not be improved any further. Now, the algorithm proceeds to optimize the objective function, leading to a reduction in the cost.

For noportsharing case, since the solution space is restricted, the search takes a longer time to find a constraint satisfying solution. In the shown example, it was found only towards the end of the search. The trace of current cost shows that most of the solutions visited had very low cost as compared to the highest cost solution ever encountered during the search. This is because of the intermediate term memory feature of the Tabu Search. After exploring the region for a minimum number of iterations, the TS memory starts *monitoring* the average cost and fitness of all the solutions encountered in that region. If the average fitness is significantly lower than the best fitness found so far, or if the average cost is very high as compared to the best (lowest) cost found so far, the search terminates exploration in this region and *restarts* with another region. This is because solutions in the *neighborhood* have similar characteristics thus enabling to view the solution space in terms of regions. Medium term memory helps in determining the chances of finding a better solution in this region.

In contrast, for portsharing case, the search finds a constraint satisfying solution very early. Thus, we can see a constant decline in the best cost value after best fitness becomes 1. Unlike noportsharing case, here we see distinct fluctuations in the current fitness and current cost as the search moves from one region to another. Even after finding the best fitness of 1, search is continued to be performed in regions of low fitness. However, the algorithm quickly terminates its exploration in such regions. This enables it to spend more time exploring regions of higher fitness and lower costs, where chances of finding a better solution are higher.

To establish the quality of results produced by the heuristic approach, we present comparison of our approach with that of the Integer Linear Programming (ILP) approach presented in [21]. The examples and the target architecture were randomly chosen so as to get a good variation of problem instances. The design is characterized by the number of logical memories. The target architecture is characterized by the total number of ports available over all instances of physical memories. The target architecture contains a mixture of on-chip and off-chip memories with multiple types of each of them. The same cost function was used in both approaches to enable a direct comparison.

As expected, the ILP approach gives better results in terms of lower cost function. However, in most of the examples, the ILP approach did not converge and had to be terminated forcefully. Thus the value of the cost function presented for ILP approach is not necessarily optimal. The commercial *CPLEX* [12] ILP-solver was used to obtain results. From Figure 3.19, we observe that in worst case, the heuristic approach was able to reach within 13 % of the best result obtained by ILP approach and within 3.5 % on an average.

Serial Num	#Logical Memories	#Ports	ILP Cost	Heuristic Cost	Heuristic Exec. Time(sec)	%Difference
1	8	18	422	422	1	0.0
2	18	39	770	777	5	0.9
3	32	63	1292	1319	39	2.0
4	27	29	1302	1381	29	5.7
5	27	37	1327	1411	27	5.9
6	39	95	1584	1622	13	2.3
7	42	77	1841	1872	52	1.6
8	49	99	1997	2280	1	12.4
9	18	52	3139	3143	5	0.1
10	32	60	4702	4931	35	4.6

Table 3.3: *Comparison of results obtained from ILP and Heuristic Approach*

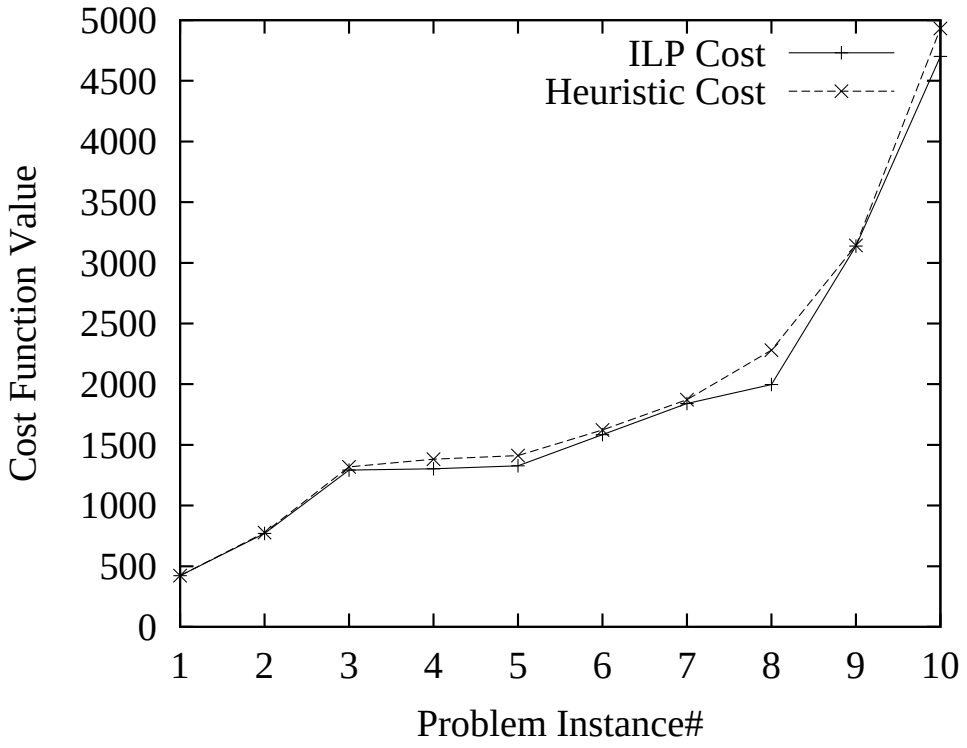


Figure 3.19: *ILP and Heuristic Cost Comparison*

For Problem 1, ILP converged to find the optimal solution and we find that even the heuristic approach was able to find the optimal solution. These results indicate the effectiveness of the heuristic approach. As the mapping complexity increases, the ILP approach becomes highly expensive in terms of execution time. The closeness of the heuristic solution to the ILP solution depends more on the complexity and constraints for that problem than the size of the problem itself. In addition, since the meta-heuristic Tabu Search *guides* the search through the solution space, its very easy to estimate the control logic and interconnect requirement at every stage of the search process without increasing the complexity of the search process itself. A more complex cost function containing equally weighted but contradicting factors could lead to an adverse effect on the execution time of ILP approach, which is already very large.

3.10 Observations and Summary

In this chapter, we presented the memory synthesis methodology. We observed that memory synthesis can broadly be divided into two stages : (1) *Memory Mapping*, which decides the physical location of various logical memories of the design, and (2) *Control Logic generation*, which produces the additional logic required to synchronize memory operations performed by the logic.

We presented a heuristic, called *Rectangle Carving* which does a *local mapping* of a single logical memory to a given physical memory type. The heuristic can easily handle splitting of logical memories across physical instances and multiple ports. The highlight of the heuristic is that it attempts to do a constraint satisfying mapping thus helping the search meta-heuristic avoid visiting non-fruitful solutions without hampering its ability to escape out of local minima.

The various factors used to evaluate a given memory mapping solution were presented. These factors provide a clearer understanding of issues involved in memory mapping.

Chapter 4

Integrated Logic Partitioning and Memory Mapping

The Integrated Logic Partitioner and Memory Mapper presented in this chapter is an extension of memory mapper presented in Chapter 3. We begin by presenting the various possible ways of achieving the same objective and the motivation for integrating the two to do the same. We present the problem formulation in Section 4.4. In Section 4.5, we discuss the new constraints which arise because of integrating the two problems and how old constraints get modified. We present the cost function factors and evaluation in Section 3.8. Experimental results are presented in Section 4.7 and conclusion in Section 4.8.

4.1 Alternatives and Motivation

During Synthesis for multi-FPGA RC Architectures, the task of logic partitioning and memory mapping are closely related. The quality of each depends on the other. An good memory mapping performed for a particular logic partitioning can turn out to be infeasible if the compute tasks are displaced to some other FPGAs.

We briefly present different ways in which the task can be performed without integrating the two tools and also present the demerits of doing so.

4.1.1 Iterative Approach

A lot of research has been done to present good spatial partitioning algorithms. An intuitive approach is to use the already present tools in conjunction with each other to perform the complete task. However,

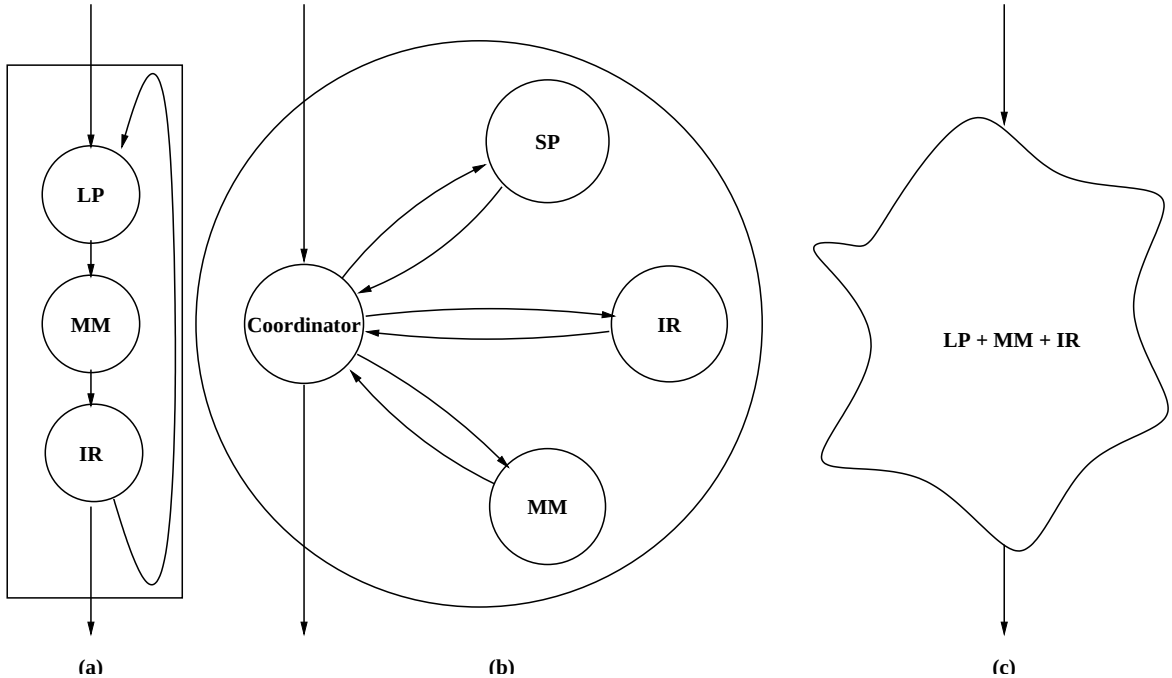


Figure 4.1: *Different Ways of performing Logic Partitioning, Memory Mapping and Interconnect Routing*

currently existing tools perform trivial form of memory mapping. There are three essential stages involved: Logic Partitioning (LP), Memory Mapping (MM) and Interconnect Routing. In order to perform any kind of routing, we need to perform both LP and MM to know where the tasks have been placed. Thus, we view routing to be part of evaluation of a given logic partition and memory mapping. The degree of success/failure of routing is one of the measures of how well the other two tasks were performed. The main two operations are LP and MM.

The current LP and MM do not have any *'understanding'* of the other's job. The LP might put big compute tasks into bigger FPGA without worrying about where their corresponding LMs will be mapped to. The MM might be forced to map LMs onto memories of FPGA other than where the accessing tasks are present. This can result in a design which is unroutable. Thus, both these LP and MM need to have the ability to take *feedback inputs* from the other. Only then can they perform their task differently in each iteration such that the overall objective is satisfied. This options corresponds to Figure 4.1 (a).

The situation discussed above defeats the initial purpose, that of reusing the existing tools. Another option is to create a **Coordinator** which makes use of the existing tools in the form they are. This Coordinator can decide which tool to call and at what time. But in order to perform the task efficiently, we will need a complex controller which will *'understand'* various aspects of all the three tools. This option requires to create a new tool. It still suffers from the disadvantage of going through iterations of LP and MM.

4.1.2 Integrated Approach

We propose to take this interaction between LP and MM to the extreme: combine the two into a single tool. This helps evaluate immediately the effect of making any change in one part on the other. In iterative approaches discussed earlier, a better cost function for both LP and MM does not necessarily mean a better overall design. However, a single cost function in integrated approach guarantees progress towards a better overall design.

We have adapted the memory mapping part for this work from the work presented in Chapter 3 with some small modifications.

4.2 Input Specifications

The Unified Specification Model proposed in [27] is highly suitable for specifying concurrency and coordination among various design segments. We are using the USM model as input.

A USM graph $\mathcal{U} = (\mathcal{T}, \mathcal{M}, \mathcal{C}, \mathcal{F})$ represents a directed graph, where $\mathcal{T}$ is the set of compute tasks, $\mathcal{M}$ is the set of logical memories, $\mathcal{C}$ is the set of logical channels and $\mathcal{F}$ is the set of flags. The definitions for compute tasks, logical memories, channels and flags has been provided in Section 2.2

For the spatial partitioning tool, all $c \in \mathcal{C}$ and all $f \in \mathcal{F}$, translate into communication requirement between the tasks. If a compute task is communicating with another compute task present in a different FPGA, or a compute task is communicating with a logical memory, which is mapped to physical memory instance not local to its FPGA, then required pins must be assigned from the interconnect architecture. All $t \in \mathcal{T}$ translate into area requirements which have to be satisfied from the area of given FPGA devices. All $m \in \mathcal{M}$ translate into storage requirements, to be mapped to various physical memory instances. Figure 4.2 shows an example of the input design specified in USM.

A target RC architecture is represented by the following:

- set of X FPGAs, $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_x\}$
- set of Y Physical Memories, $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_y\}$
- interconnection matrix $\mathcal{I}$, which specifies the available bits between each *PE-PE* pair which can be used for routing.
- set $\mathcal{T} = \{t : memory\_type\}$ such that $\{\forall m \in \mathcal{M}, \exists_1 t \in \mathcal{T} | m \rightarrow t\}$

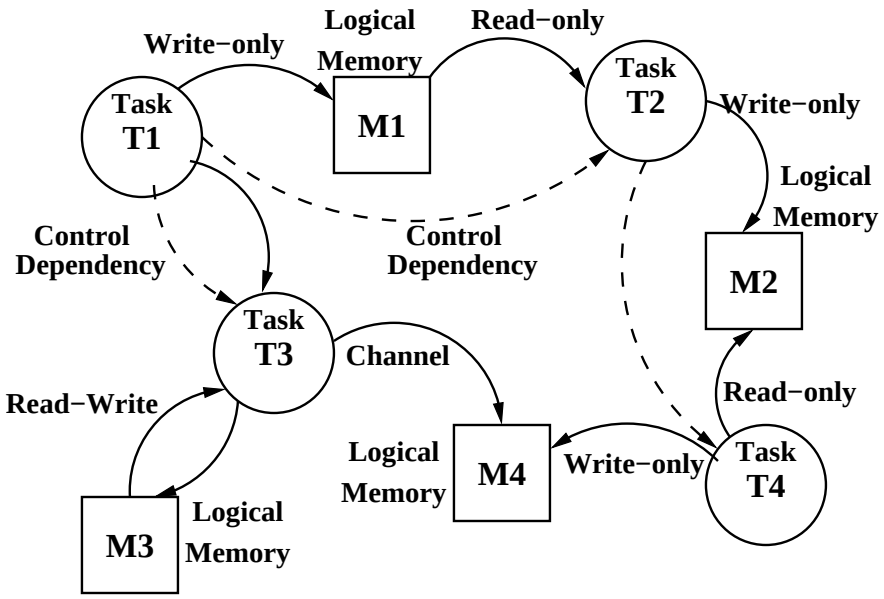


Figure 4.2: *An Example design specified in USM*

4.3 Problem Definition

Given:

- A USM graph $\mathcal{U} = (\mathcal{T}, \mathcal{M}, \mathcal{C}, \mathcal{F})$, where $\mathcal{T}, \mathcal{M}, \mathcal{C}, \mathcal{F}$ are compute tasks, logical memories, channels and flags respectively. $\mathcal{L}$ specifies the input design.
- **Set** $\mathcal{B} = \{b : memory\_type\}$, where each $b \in \mathcal{B}$ has attributes specified in Section 2.1. $\mathcal{B}$ specifies the target architecture.
- set $P = \{pm : physical\_mem\_instances \mid \forall\, pm\, \exists_1 t \in \mathcal{T}\}$
- num_fpgas
- **set** $\mathcal{I} = \{i_{f_1 f_2} \mid 0 \leq f_1, f_2 < num\_fpgas\}$. Each element of $\mathcal{I}$ specifies the number of interconnect pins available between the corresponding fpga pair.

The objective is to produce sets of mapping

$$\mathcal{M}_t = \{m : compute\_task\_mapping \mid \forall t \in \mathcal{T}, \exists_1\, t \leftrightarrow m \in \mathcal{M}_t\}$$

$$\mathcal{M}_m = \{m : memory\_mapping \mid \forall l \in \mathcal{M}, \exists_1 l \leftrightarrow m\}$$

such that

$$\forall f \in F, satisfies(Constraints_f)$$

$$\forall pm \in P, satisfies(Constraints_{pm})$$

4.4 Mapping : Definitions and Formulation

The task of a spatial partitioner is to assign physical location to various components of the input USM graph. Channels and Flags consume various interconnect resources in the form of pins. A mapping for interconnect component is the problem of *pin assignment*. In our work, we ensure the routability of the design for an abstract interconnection representation of the board architecture. Since our approach is board architecture independent, pin assignment can not be performed for a generic board model.

4.4.1 Mapping Definitions

We now present definition of mapping for the other two components of the USM.

- **Mapping for Compute Task**

  We assume that a compute task can not be divided across PEs. In addition, we also assume that a compute task is mapped to only one PE, i.e. duplication of compute tasks is not permitted. Therefore, mapping for a compute task simply means assigning it to a particular PE instance. This is reflected by assigning a value to *pe_num*. The value can be anything between *0 and num_pe-1*, where num_pe is the number of PEs on RC.

- **Mapping for Logical Memory**

  Mapping for a logical memory implies detailed information about how many parts it has been splitted and where the various parts are assigned to. A detailed definition of mapping for logical memory is provided in Section 3.2.

4.4.2 Problem Formulation

In this section, we present the formulation of the problem suited for tabu search. Figure 4.3 pictorially shows the formulation. At an abstract level, both compute and logical memory are treated alike. A

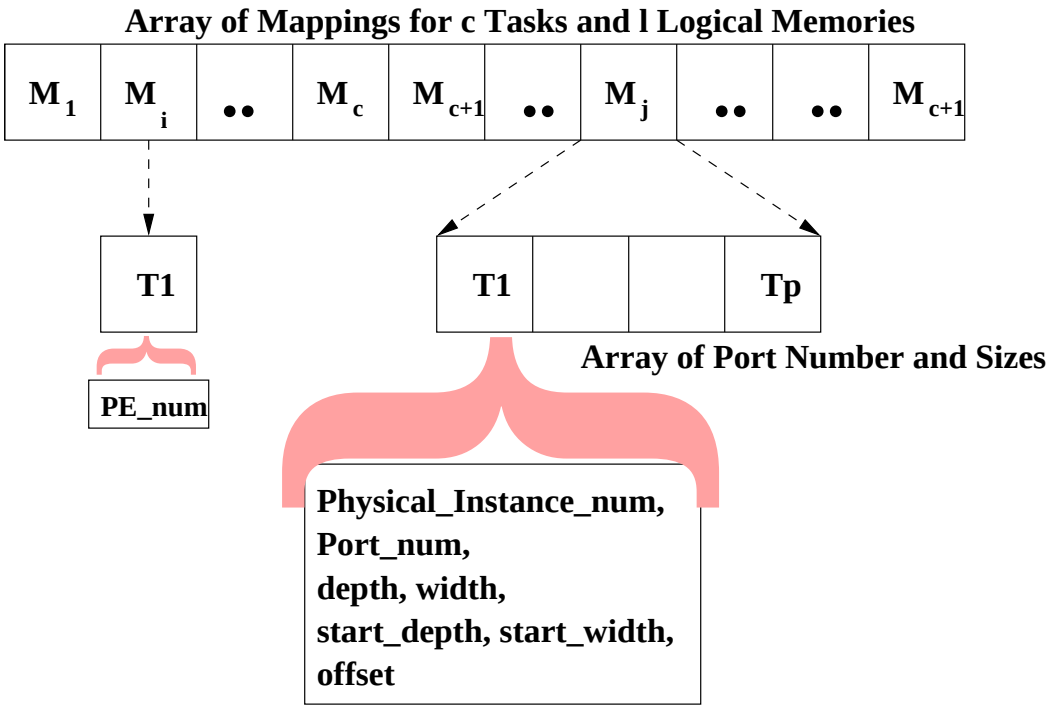


Figure 4.3: *Heterogeneous array of mappings for compute tasks and logical memories*

single solution array of size equal to sum of number of compute and logical memory is created. This is a heterogeneous array of mappings. The kind of mapping will depend upon the type of the task.

4.5 Constraints

As in the case of memory mapping, there are certain architectural constraints dictated by the target RC architecture. Besides the three constraints specified in Section 3.7, an additional constraint in the form of FPGA Area of individual devices is present for logic partitioning. The mapping should ensure that the total logic area requirements for a FPGA device does not exceed the maximum area available on that device. A non-zero area penalty, calculated as shown below, indicates that area constraint on at-least one of the FPGA devices has been violated.

$$AreaPenalty = \sum_{i=0}^{num\_fpgas-1} \frac{total\_occupied\_area - maximum\_area}{maximum\_area}$$

For design constraints, the user can specify if he desires the partitions to have a balanced area. If the partitions are balanced in terms of area requirements, the design will get through the place and route

tools faster. However, a distributed logic can lead to more signals being routed across FPGAs, potentially slowing down the design. This is an optional input which the user can provide. All the other design constraints specified for memory mapper in Section 3.7.2 are also applicable.

4.6 Cost Function and Estimation

The total area required on an FPGA is equal to the sum total of areas of all the tasks mapped to that device and the control logic required for all the physical memory instances local to that FPGA. The area of compute tasks is obtained as input from the user. Alternatively, the tool can also do a design space exploration to determine the area of all compute tasks such that the overall area and latency is optimized. A design space exploration methodology is presented in [10] and can be easily integrated with this tool. The control logic required is inserted in the design by this tool itself. The estimates for area of each component like arbiter, comparator, tri-state buffer etc. are fed into the tool based on stand-alone pre-synthesized library estimates.

All other factors of the cost function given in Section 3.8 for memory mapping are also applicable for the logic partitioning and memory mapping tool, and hence are not repeated here.

4.7 Experimental Results

We present the results of our Integrated Logic Partitioning and Memory Mapping tool in this section. Table 4.1 shows the details of the various benchmarks used. All communication between tasks is done through memories. Hence memory address and data buses are the only channels present in the taskgraph.

For each benchmark, we present results for three different board architecture, with 2, 3 and 4 FPGAs respectively. The physical memories present on the board have been chosen to represent boards close to the real architectures available. Each FPGA has two types of memories associated with it. One type of memory has 4096 bits, which can be configured in 5 different ways as $\{4096\times1$, 2048×2, 1024×4, 512×8, $256\times16\}$. This memory type has 2 ports, read and write latency of 1 clock cycle and is 0 pins away from its local_pe i.e. it is an on-chip RAM. The other type of memory has 32K bits and has one configuration, 2048×16. There are 10 instances of the first type of memory and 2 instances of the second type associated with each FPGA. We assume that the interconnection network provides 108 wires between every FPGA pair. This is in tune with most of the boards available now a days. However, for the *Rand100* example, we assume a capacity of 250 wires between every FPGA. Such high interconnect requirements are a result of logical memories being mapped to different physical memories available, which prevents

Example	#Compute Tasks Total Area	#LogMems {depths}	#Flags	#Channels
DCT1	9 2598	15, {4,4,4,4,4,4,4,4, 16,16,16,16,16}	9	36
LUD	9 1337	13, {105,110,120,115,75, 96,87,90,100,110,110,115,110}	11	24
Laplace	9 607	14, {45,50,40,55,25,36,26,17 10,70,10,50,35,40}	14	26
MeanValue	9 1337	13, {15,90,190,15,55,175, 47,100,120,110,118,115,50}	12	24
FFT	12 3375	12, {9,9,9,9,8,8, 8,8,8,8,8,8}	11	48
DCT2	32 4875	10, {4,4,4,4,4,4, 4,4,4,4,4,4 }	31	80
Rand100	100 {30-60}	100 {20-50}	100	149

Table 4.1: *Design Data for Benchmark Examples*

sharing of address and data buses between them.

We also observe that the execution times are fast. For examples in which it was able to find a solution, the maximum execution time was less than 10 minutes. The results were taken on a machine with four 336 MHz sparcv9 processors with 1600 MB of RAM. However, the implementation is single-threaded [28] and the maximum memory requirements of the algorithm were found to be less than 4 MB.

4.8 Observations and Summary

In this chapter, we presented the technique to do memory mapping along with logic partitioning. We briefly discussed other options and motivated the need to perform the two tasks in an integrated manner. We showed that the two are closely inter-related and doing without the other can lead to much poorer design quality or much higher time requirements on part of the tool. We presented the results of logic partitioning and memory mapping on a number of benchmark examples. However, the tool takes an abstract specification of the interconnect on the target architecture. It assumes that routing is done only through direct wires available between any PE-PE pair.

Example	#FPGAs, $\mathbf{Area_F}$	Estimated Occupied Areas	Interconnect Requirements	Exec Time (sec)
DCT1	2, 1400	{1369,1389}	{74}	113
	3, 950	{903,933,910}	{38,62,62}	128
	4, 700	{684,683,686,693}	{61,30,29,37,14,49}	434
FFT2	2, 1800	{1850,1801}	{106}	321
	3, 1300	{1126,1279,1284}	{35,45,65}	122
	4, 1000	{972,940,892,847}	{35,67,34,61,17,49}	137
DCT2	2, 2750	{2719,2719}	{99}	114
	3, 1830	{1784,1796,1802}	{58,67,67}	129
	4, 1380	{1365,1357,1334,1361}	{55,33,34,66,67,45}	145
MeanValue	2,730	{720,720}	{99}	112
	3,470	{476,484,469}	{97,51,55}	381
	4,400	{375,337,382,344}	{21,50,21,37,78,34}	144
Laplace	2,350	{345,362}	{107}	352
	3,230	{247,227,233}	{76,61,106}	398
	4,175	{169,183,182,173}	{78,32,45,79,49,32}	449
LUD	2,750	{710,754}	{100}	112
	3,550	{438,547,457}	{69,50,55}	129
	4,400	{374,374,363,342}	{35,64,52,34,22,51}	144
Rand100	2,3100	{3092,3335}	{236}	2082
	3,1900	{1937,2135,1894}	{213}	2188
	4,1600	{1560,1553,1552,1523}	{126,118,111,133,139,123}	765

Table 4.2: *Results of Logic Partitioning and Memory Mapping*

Chapter 5

Conclusions and Future Work

In this thesis, we presented an automated memory mapping methodology during high level synthesis flow targeted towards FPGA based Reconfigurable Platforms. In Chapter 2, we presented an overview of various components of input design and the target architecture. We also presented a detailed description of how Tabu Search meta-heuristic was adapted for memory mapping. Various techniques designed specifically for speeding up the search process for memory mapping were also presented. In Chapter 3, we present the details of the memory mapping methodology. This is the main contribution of this thesis. We present various important factors which affect memory mapping and how they were incorporated in the search heuristic to obtain better design. In Chapter 4, we present an integrated logic partitioning and memory mapping methodology.

In any data intensive application, the design spends most of the time in reading from and writing to the memories. Any methodical mapping which helps design exploit the fast physical memory instances available on-chip can lead to significant reduction in design execution time. Any contemporary FPGA device comes with large number of such on-chip RAMs which provide very fast access to storage space. Considering the large number of physical instances available, it is not feasible to come up with a good mapping by hand. Sometimes, for big design, it might be impossible to even find a constraint satisfying mapping by hand, let alone a optimized one. Thus we clearly see the need for a framework to perform the memory mapping task automatically.

Since memory mapping is a combinatorial problem, a meta-heuristic search algorithm does a good job of finding a near optimal solution in very little time. Other techniques like Integer Linear Programming (ILP) formulation for memory mapping, find the optimum solution. However they take an unreasonably long time. We present the use of Tabu Search to perform memory mapping. It has been found to be highly effective both in terms of the quality of solution and the time required to find the solution.

We present a heuristic algorithm to perform mapping at every iteration of the Tabu Search. The algorithm is designed to handle splitting of logical memories across physical instances and multiple ports. It is a constructive algorithm in the sense that it tries to build a valid mapping for a logical memory with all constraints satisfied as far as possible.

We also established the need for tackling both logic partitioning and memory mapping problems in totality. We presented an integrated approach based on tabu search.

5.1 Contributions of the Thesis

This work presented an automated memory mapping methodology to improve overall latency of a design targeted towards reconfigurable platforms. The important contributions of this work are :

- *Logical Memory and Mapping Model:* A detailed model of the logical memory is presented which captures many essential attributes which are crucial in determining its mapping. We also showed how some of the attributes of the model can be easily simplified in the absence of information. We also defined what constitutes a complete mapping. The mapping definition includes information required for generation of control logic necessary for the mapping to be meaningful.
- *Mapping Heuristic:* We presented a heuristic algorithm called *Rectangle Carving*, designed to perform mapping of logical memories, by splitting them over multiple instances or combining multiple logical memories to single instance. The algorithm takes as input a logical memory to be mapped and the physical memory type over which it has to be mapped. The algorithm takes into consideration previously mapped logical memories to this physical memory type. It tries to do a constraint satisfying mapping thus helping speed up the tabu search process by avoiding non-fruitful solutions.
- *Tabu Search Formulation and Enhancement:* We presented a formulation of the memory mapping problem which can be easily exploited by the tabu search algorithm. We defined moves from one solution to another and identified some critical attributes of the moves. These attributes were used to build the short-term, intermediate-term and long-term memories of the tabu search which enhanced the effectiveness of the search algorithm.
- *Integration with Logic Partitioner:* We identified the need to integrate the memory mapper with logic partitioner in order to get good results in reasonable time. Consequently, we present the methodology to perform the two tasks together using tabu search formulation.

5.2 Directions for Future Work

We presented a model of the logical memory which we make use of to perform the mapping. This model can be further enhanced. The model presented does not consider the life times of logical memories. If all accesses to a logical memories are done before any access to another logical memory is even started, they both can be mapped to same storage space. The host has to take care whether the memory earlier in life time needs to be read-back or not. It also has to consider whether the new logical memory needs initialization. This will require the mapper to prevent any access to that storage space during transition from one logical memory to another.

The model also does not consider the time of access of a logical memory by various compute tasks. Instead of arbitration the tasks, the mapper can introduce control flags between the accessing tasks to ensure that parallel access does not happen. However, in case of sharing of port by multiple logical memories, which compute tasks might need serialization will depend upon which logical memories share the same port. Thus high level synthesis tool needs to be run again after memory mapping has been performed to introduce these flags.

The arbiter in our approach arbitrates all compute tasks accessing any logical memory at a logical port. It allows access to only one logical memory at a time. However, it is possible that two logical memories at a logical port do not directly share the same port. They might have been grouped together because they separately share port with a third logical memory. A new arbiter logic needs to be developed which can handle such parallel accesses.

If a physical instance has multiple ports, our approach ensures that access to same storage space does not happen through multiple ports. This is ensured by specifying a region of storage space for each port, outside which it is not allowed to access. However, this leads to loss of parallelism. If two tasks are performing read operations on the same logical memory, they should be allowed to access it simultaneously through the multiple ports available on the physical instance. This will require change in control logic generation as well as arbiter logic.

Bibliography

[1] AHMAD, I., AND CHEN, C. Y. Post-process for data path synthesis. In *Proceedings of International Conference on Computer Aided Design* (1991), ACM Press, pp. 276–279.

[2] ALTERA CORPORATION. *APEX 20K Programmable Logic Device Family Data Sheet*, March 2000.

[3] ALTERA CORPORATION. *FLEX 10K Embedded Programmable Logic Family Data Sheet*, May 2000.

[4] ANNAPOLIS MICRO SYSTEMS, INC. *WILDFORCE Reference Manual*. http://www.annapmicro.com, 1998.

[5] ANNAPOLIS MICRO SYSTEMS, INC. *WILDSTAR Reference Manual, Rev. 4.0*. http://www.annapmicro.com, 2000.

[6] BALAKRISHNAN, M. Allocation of multiport memories in data path synthesis. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, volume 7* (April 1998), pp. 536–540.

[7] CONG, J., AND YAN, K. Synthesis for fpgas with embedded memory blocks. In *Proceedings of International Symposium on Field Programmable Gate Arrays* (February 2000), ACM Press, pp. 75–81.

[8] FIDUCCIA, C. M., AND MATTHEYSES, R. M. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference* (1982), ACM Press, pp. 241–247.

[9] GLOVER, F., AND LAGUNA, M. *Tabu Search*. Kluwer Academic Publishers, 1997.

[10] GOVINDRAJAN, S. *High-Level Synthesis and Exploration Techniques for Multi-Device Architectures*. PhD thesis, University of Cincinnati, 2000.

[11] HO, W., AND WILTON, S. Logical-to-physical memory mapping for fpgas with dual-port embedded arrays. In *International Workshop on Field Programmable Logic and Applications* (September 1999), pp. 111–123.

[12] ILOG INCORPORATION. *Using the CPLEX Callable Library*. http://www.cplex.com.

[13] IYAD OUAISS, SRIRAM GOVINDARAJAN, VINOO SRINIVASAN, MEENAKSHI KAUL, AND RANGA VEMURI. "A Unified Specification Model of Concurrency and Coordination for Synthesis from VHDL". In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis* (July 1998).

[14] JHA, P., AND DUTT, N. High-level library mapping for memories. In *ACM Transactions on Design Automation of Electronic Systems* (July 2000), ACM Press, pp. 566–603.

[15] KARCHMER, D., AND ROSE, J. Definition and solution of the memory packing problem for field-programmable systems. In *Proceedings of International Conference on Computer Aided Design* (November 1994), ACM Press, pp. 20–26.

[16] KAUL, M. *Optimal and Near-optimal Temporal Partitioning techniques for Dynamically Reconfigurable Computers*. PhD thesis, University of Cincinnati, 2000.

[17] KERNIGHAN, B. W., AND LIN, S. An efficient heuristic procedure for partitioning. In *Bell System Technical Journal* (Feb. 1970), pp. 291–307.

[18] KIM, T., AND LIU, C. L. Utilization of multiport memories in data path synthesis. In *Proceedings of 30th Design Automation Conference* (June 1993), ACM Press, pp. 298–302.

[19] N. NARASIMHAN, V. SRINIVASAN, M. V. J. W. S. G., AND VEMURI, R. Rapid prototyping for reconfigurable co-processors. In *International Conference on Application-Specific Systems, Architecture and Processors* (August 1996), pp. 303–312.

[20] OUAISS, I., AND VEMURI, R. Global memory mapping during synthesis in fpga-based reconfigurable computers. In *Reconfigurable Architectures Workshop* (San Francisco, September 2000), Springer-Verlag, pp. 284–293.

[21] OUAISS, I., AND VEMURI, R. Hierarchical memory mapping during synthesis in fpga-based reconfigurable computers. In *Design Automation and Testing Conference of Europe* (Berlin, Germany, September 2000), Springer-Verlag, pp. 284–293.

[22] OUAISS, I., AND VEMURI, R. Resource arbitration in reconfigurable computing environments. In *Proceedings of Design Automation and Test in Europe* (April 2000), IEEE Computer Society Press, pp. 560–566.

[23] R. DUTTA, J. R., AND VEMURI, R. Distributed design space exploration for high-level synthesis systems. In *29th Design Automation Conference* (June 1992), ACM Press.

[24] RADHAKRISHNAN, S. Generic interconnect synthesis system for routing in reconfigurable boards. Master's thesis, University of Cincinnati, 1998.

[25] SAIT, S. M., AND YOUSSEF, H. *Iterative Computer Algorithms with Applications in Engineering*. IEEE Computer Society, 1999.

[26] SMITH, D. Race: A reconfigurable and adaptive computing environment. Master's thesis, University of Cincinnati, 1997.

[27] SRINIVASAN, V. *Partitioning for FPGA Based Reconfigurable Computers*. PhD thesis, University of Cincinnati, 1999.

[28] STROUSTROUP, B. *Programming in C++*. 1995.

[29] SYNPLICITY INC. *Synplicity Synplify*. http://www.synplicity.com, 2000.

[30] T. CORMEN, C. L., AND RIVEST, R. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[31] TSENG, C. J., AND SIEWIOREK, D. Automated synthesis of data paths in digital systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (July 1986), pp. 379–395.

[32] UNIVERSITY OF SOUTHERN CALIFORNIA. *SLAAC-1V Boards*. http://www.east.isi.edu/projects/SLAAC.

[33] WILTON, S. Architecture and algorithms for field programmable gate arrays with embedded memory.

[34] WILTON, S. Heterogeneous technology mapping for fpgas with dual-port embedded memory arrays. In *Proceedings of International Symposium on Field Programmable Gate Arrays* (February 2000), ACM Press, pp. 67–74.

[35] XILINX CORPORATION. *Using Virtex BlockRAMs*, 1999.

[36] XILINX INC. *PAR: Xilinx Place and Route M1.5*, 1998.