DATE: 21st June

I, Ankur Pandey, hereby submit this as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

A Multithreaded Runtime Support Environment
For Dynamic Reconfigurable Computing.

Approved by:

Dr. Harold W. Carter
Dr. Fred Beyette
Dr. Raj Bhatnagar
A Multithreaded Runtime Support Environment for Dynamic Reconfigurable Computing

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

June 2002

By

Ankur Pandey

B.Tech in Electronics and Communication Engineering
Regional Engineering College (REC), Warangal, India.
May 1999

Thesis Advisor and Committee Chair: Dr. Harold W. Carter
Abstract

Reconfigurable computing system containing a number of reconfigurable devices is designed to reconfigure some or all of the devices during execution. An important aspect of reconfigurable computing is to develop tools that aid designer in mapping algorithms and computational tasks to the adaptive system. In this thesis we propose a runtime multithreaded support environment for managing the loading, execution and swapping of application modules on a FPGA. A virtual hardware manager is created that schedules execution of application modules, allocates and de-allocates FPGA resources. A fast runtime placement algorithm is proposed and evaluated for the system. Issues involving data storage are investigated. The impact of multithreading and pre-fetching is summarized. We have also suggested a component-based approach to application design. The performance of proposed a runtime system incorporating these algorithms was evaluated in a simulator based environment and is described here.
Acknowledgements

I would like to take this opportunity to thank my advisor Dr. Harold W. Carter for his guidance and support. Without his constant motivation and constructive criticism this research work would not have been possible. I would also like to thank Prof. Karen Tomko for giving advice and support when needed.

Thanks also to all other members of the Distributed Processing Laboratory who helped in whatever way possible through the course of this thesis. Kunal Mehra deserves special mention for his helpful tips for developing this system. He was always there when times were tough. Thank you Kunal.

And whatever I am today is because of my terrific family. Their love and good wishes were a constant source of inspiration. I owe them so much that saying thank you does not express how much I appreciate all that they have done for me.
Contents

List of Figures

1 Introduction

1.1 Objectives .................................................................3

1.2 Thesis Outline ............................................................4

2 Background and Related Work

2.1 FPGA’s (Field Programmable Gate Arrays) .....................5

2.2 Reconfigurable Computing Design Styles ......................5

2.3 JBits -- A Java Interface for Reconfigurable Computing ....7

2.4 Limitations of JBits ......................................................10

2.5 Related Work .............................................................10

3 Concepts and System Model

3.1 Goals and Concept.......................................................13

3.2 System model............................................................14

3.3 Intermediate Format Representation...............................17

3.4 Virtual Hardware manager..............................................18

3.4.1 Placement..............................................................20

3.4.2 Resource Monitor....................................................20

3.4.3 Interrupt mechanism................................................22
3.4.4 Core Eviction and Loading.................................23

3.4.5 Handling High Level Language Constructs..............23

4 Optimizations

4.1 BRAM Vs Registers.................................................25

4.2 Multithreaded Environment.....................................29
   4.2.1 Implementation Multithreaded Environment..............30
   4.2.2 Advantages and Disadvantages of Thread Based Approach..32

5 Placement and Routing

5.1 Routing.................................................................35

5.2 Placement..............................................................36

6 Simulation and Results

6.1 Simulation Environment...........................................45

6.2 Performance Evaluation Methodology.........................46

6.3 Placement Results...................................................47

6.4 Evaluation of Data Storage Techniques.........................55

6.4 Evaluation of Multithreaded Environment......................57

7 Conclusions and Future Work

7.1 Conclusions..........................................................60

7.2 Future Work..........................................................61
List of Figures

Figure 1 Traditional tradeoff between performance and Flexibility..................2

Figure 2 Compile Time Reconfiguration....................................................6

Figure 3 Run Time Reconfiguration.......................................................6

Figure 4 JBits Application Flow.............................................................8

Figure 5 Top Level View of Reconfigurable System.................................13

Figure 6 Conditional Execution.............................................................16

Figure 7 Overview of Our System..........................................................17

Figure 8 Tasks Performed by VHM.........................................................19

Figure 9 VHM Execution Flowchart.......................................................21

Figure 10 A Finite State Machine..........................................................22

Figure 11 LCU Structure For Four Nested Loop Construct.......................24

Figure 12 Interaction of BRAM with Data Path.......................................26

Figure 13 BRAM with A Demultiplexer.................................................27

Figure 14 Building Block of a Demultiplexer.........................................28

Figure 15 Threads Vs Hardware Objects...............................................29

Figure 16 Life Cycle Of A Child Thread...............................................31

Figure 17 Working With Threads.........................................................32
Figure 18 Working of Threads with Semaphore

Figure 19 Flowchart for First Fit

Figure 20 Flowchart for Best Fit

Figure 21 Placement of 2 X 1 module

Figure 22 Choice of Placement Coordinates

Figure 23 Working of Our Algorithm –1

Figure 24 Working of Our Algorithm –2

Figure 25 Checking Successive Radius for Optimum Placement Coordinates

Figure 26 Pin Location for CLB’s in Our Algorithm

Figure 27 Pseudocode for Our Algorithm

Figure 28 Placement Times For Various Algorithms

Figure 29 Number of Unused CLB’s

Figure 30 Average Distance per Placement call Vs Size of FPGA

Figure 31 First Fit Vs Our Algorithm with Core Eviction

Figure 32 First Fit Vs Our Algorithm

Figure 33 Our Algorithm With Core Eviction –1

Figure 34 Our Algorithm With Core Eviction –2

Figure 35 First Fit With Core Eviction –1
Figure 36 First Fit With Core Eviction –2........................................55

Figure 37 Time required by BRAM Vs Register based approach for 2 inputs.....56

Figure 38 Time required by BRAM Vs Register based approach for 3 inputs ....57

Figure 39 Execution Time with BRAM, Register & Demultiplexer..............57

Figure 40 Execution Time with different number of Threads....................59

Figure 41 Execution Time in Different Size of Applications....................59

Figure 42 Virtex Architecture Overview ..............................................69

Figure 43 CLB Overview .................................................................70
Chapter 1

Introduction

By definition, Reconfigurable Computing means using a flexible, structurally changeable platform to achieve computational tasks. There are traditionally two primary methods of executing a computational problem. The first is use of hardwired, customized Application Specific Integrated Circuits (ASIC). ASIC’s derive their primary advantage from speed. Tailored to execute a particular application, the hardware resources are optimally tuned for that specific algorithm. However a major drawback that ASIC’s suffer from is lack of re-programmability after chip fabrication. If a change is desired in the functionality of the circuit a potentially costly re-design and re-fabrication is required.

Another approach is to use a general purpose microprocessor with a finite instruction set capable of executing many applications – requiring only a change in the software instructions to change its programmability. This advantage of increased flexibility comes at a cost of relatively degraded speed performance. The controller-datapath cycle of execution results in much slower speed over similar hardwired functions in ASIC.

Reconfigurable Computer (RC) systems provide two main advantages over traditional “fixed structure” computers:

• More flexibility means less restricted system design.
• Improved performance with limited resources by utilizing dedicated custom solutions.
Traditionally, there exists a design tradeoff between flexibility and performance for computational systems. With a fixed structure, to obtain a higher degree of flexibility the system must be able to implement more algorithms with limited hardware resources. On the other, by using special purpose custom logic, the system can achieve maximum performance on a single, or very limited set of tasks, but without the flexibility to solve other problems. Figure 1 shows this tradeoff in a general diagram.

![Diagram of Performance vs. Flexibility](chart)

**Figure 1. Traditional tradeoff between performance and Flexibility [14].**

The primary difficulty in a reconfigurable approach lies in system design. The designer must manage the behavior of the system, determine the operational modes of the system, define the rules for transitioning between those modes and the functional
properties between those modes. An application programmer would rather spend his / her
time and energy developing algorithms instead of the issues involved in managing the
execution involving tasks such as loading, executing and swapping hardware objects in
and out of the FPGA. This means an intelligent abstraction layer (at runtime) is needed
that hides the details of the particular devices from the application programmer.

1.1 Objectives

It is the objective of this thesis to investigate dynamic runtime support
environment supporting multiple threads of execution. The purpose of this system is to
relieve designer from the management issues like loading, execution and swapping of
hardware objects from the FPGA. This system acts as an abstraction layer and allows the
designer to use high level requests to access the system. We have proposed a
multithreaded environment that can keep track of multiple hardware objects being
executed on a FPGA.

A resource manager is developed which provides capabilities including

- Interfacing with FPGA hardware/Simulator.
- Placement of a sub partition of an application on the FPGA.
- Creating and destroying child threads as and when required. Each thread keeps
  track of one hardware object on board.
- Handling input/output of data to/from cores (pre placed and pre routed
  functionality).
- Dynamic demand based loading of program modules / hardware objects / cores.
In my thesis emphasis has been given on study of placement algorithms for this runtime support system. Various data storage techniques have been evaluated. A study has been done on performance improvement with multiple threads.

1.2 Thesis outline

Background information on FPGA’s, types of reconfiguration and JBits has been provided in chapter 2. Chapter 3 deals with the basic concept and system design including placement, resource usage, parameterizable core creation etc. We deal with optimization issues including multithreading in chapter 4. In chapter 5 we discuss various place and route algorithms that can be used and the tradeoffs involved. Chapter 6 provides the simulation results. The thesis concludes with summary of conclusions and future work in chapter 7.
Chapter 2

Background and Related work

2.1 FPGA (Field Programmable Gate Arrays)

FPGA’s are completely prefabricated logic devices optimized for implementing logic circuits. The programmable points on an FPGA can be antifuses, EPROM, EEPROM or SRAM cells. SRAM based cells are easily programmable more than once but they are volatile i.e. they retain their configuration only while they are powered. These type of FPGA’s are most suited for our system as we have to execute multiple hardware modules.

2.2 Reconfigurable Computing Design Styles

Reconfigurable computing design methodologies can broadly be classified into two styles

- Compile Time Reconfiguration (CTR)
- Run Time Reconfiguration (RTR)

In CTR, configuration of the FPGA board remains static for the life of the application. It is downloaded once to the board before initiating the application and is destroyed only after termination of the program. Figure 2 illustrates this. The primary benefit we extract from an FPGA by using this style is the ability to re-use the hardware logic as opposed to the single time usage of an ASIC’s logic cells. Because hardware resources remain static for life of the application conventional design tools provide adequate support for application development [20].
Figure 2. Compile Time Reconfiguration.

We can further extract more performance benefits from an FPGA by following a less restrictive scheme in terms of allowing the configuration on board to change many times during the course of an application’s lifetime. RTR does this, where multiple configurations are downloaded to the board and may even be concurrently active on chip. One of the main advantages of RTR is that it provides more throughput per silicon area. Unused hardware resources are minimized. This also gives the designer flexibility of partitioning the application into multiple time exclusive cores to more effectively reuse hardware resources. Figure 3 shows RTR.

Figure 3. Run Time Reconfiguration.

It can be observed that there exists the possibility of splitting the above design style into multiple time exclusive replica’s – Local RTR and Global RTR. Specifically, we can either allow multiple active data cores on FPGA at any given time and keep on reconfiguring the board for every set of cores (Local RTR) or we can follow the single core, configure—execute policy for each of the sub-cores of the application (Global RTR). Global RTR involves several issues like inter configuration communication,
passing of intermediate results between successive configurations, and time partitioning of application into sub-cores. Local RTR is one step more flexible than Global by providing the freedom of having multiple active data cores on FPGA. However it also increases the complexity of the overall system. Besides currently available CAD tools are a poor match for local RTR implementations.

2.3 JBits -- A Java Interface for Reconfigurable Computing

We give here a brief overview of JBits, which we have used in our research. The JBits API is a set of Java classes that provide an Application Program Interface (API) into the Xilinx Virtex FPGA family bitstream. The real power of JBits is its use in the development of Java Run Time Reconfiguration (RTR) applications. In this flow, the circuits can be configured on the fly by executing a Java application that communicates with the circuit board containing the Virtex device. This is made possible by using the JBits to specify the design and use the XHWIF (Xilinx Hardware Interface) API to download the design within the same Java application. Again, this bitstream input to the Java Application can be a null bitstream or a bit-stream for an existing design. Figure 4 gives JBits application flow. We have provided more information on JBits and XHWIF in Appendix. To sum up JBits program creates logic, converts it to a bitstream, interfaces with the FPGA through a universal XHWIF, reads back the current state, modifies the bitstream and repeats the process for every module in the application, thereby modeling the functionality and performance of an application.
Figure 4. JBits Application Flow.
JBits also has a library of Run time Parametrizable (RTP) core primitives [1], allowing designers to build applications on top of these RTP cores. In addition to providing support for runtime reconfiguration computing, RTP cores permit runtime parameterization of cores. Parameterizable cores allows the user to enter information about the desired core, and a customized circuit conforming to the information supplied by the user is constructed. One example of parameterizable core would be an adder circuit. When requesting an adder core a user would be asked to specify bit width of an adder core. We have used these RTP cores provided by JBits library and developed some of our cores based on these primitive cores.

We reserve the acronym FPGA to refer to the actual hardware chip that the virtual FPGA (vFPGA) represents. The configuration routines are a part of the XHWIF. The application uses these routines to interface with the vFPGA. JBits takes as its input, a program bitstream (from the FPGA) and uses it to dynamically modify circuits at runtime.

A limitation of our research was the unavailability of an FPGA board. Thus we used the VirtexDS [9] to verify our approach. VirtexDS has a stimulus/probe interface that provides the ability to probe/stimulate LUTs (Look up tables) at the individual pin level – providing a virtual testbench capability. VirtexDS supports RTR in that partial configurations can be inserted into the bit-stream at run time and executed.
2.4 Limitation of JBits

Perhaps the largest drawback of JBits is its manual nature. Everything must be specified explicitly in the source code, including the routing. JBits favors more structured circuits. Users must be familiar with the underlying device architecture. In addition JBits works at the bitstream configuration level, it exists at the most downstream end of tool chain. This eliminates the possibility of using analysis tools available to the circuit designer further up the tool chain.

The ability to do any sort of timing analysis is absent in JBits. Small changes in the circuit configuration may have a dramatic impact on functionality as well as timing. During our research we have noticed that JBits does not perform that well with RTR. It sometimes leave some pins undefined after reconfiguration.

2.5 Related work

Currently, there are a few run-time support systems that have been studies and reported in the literature. One of the early studies in this area is the “First Real OS for Reconfigurable Computers” [3] by Wigley et al that attempts to develop a complete operating system for RC system management. It is based on the concept of virtual memory management for commercial operating systems and performs common analogous OS tasks like dynamic partitioning of applications, FPGA placement and multi-user execution support. The system uses a simple and fast constructive placement algorithm to reduce the algorithm run-time. Placement and routing are performed at run-time. It is however still not completely out of the implementation phase with issues like
inter process communication and pre-emptive user circuitry of cores remaining to be resolved.

Another approach is taken by Bubb et al who propose an FPGA support system [4] to support placement, routing and core management. It provides a support mechanism for communication between concurrent user processes. It was implemented on the Xilinx 6264 FPGA connected to an ARM7 host processor. It however does not include any optimizations to reduce reconfiguration times.

Wirthlin and Hutchings propose the concept of a dynamic instruction set computer, DISC [5]. DISC supports demand driven modification of its instruction set using a partially reconfigurable FPGA. Instructions are treated as removable modules paged in and out, as demanded by the executing program. It adopts a demand driven approach to load cores onto FPGA, much similar to what we have used in this thesis. Instructions occupy FPGA resources only when needed and resources can be reused to implement an arbitrary number of performance enhancing applications. It executes by filling up the FPGA with custom instruction modules till there is no further space remaining, upon which it invokes a cleaning daemon to evict least recently used modules. DISC maintains a library of circuits and allows them to be placed at any vertical location on the FPGA.

Work similar to ours has been done in development [7] of Resource Manager (RM) to allocate and de allocate FPGA resources at runtime and to preload FPGA configuration files. For each individual application, different tasks that require FPGA
resources are represented as a flow graph which is made available to RM as to manage efficient resource management and preloading.

Donlin et al at the reconfigurable architecture group (RAGE), present the design of a run-time system for managing dynamic reconfiguration of FPGAs. The RAGE system [6] again draws upon parallels from operating system principles and FPGA resource management to develop a virtual hardware manager. It proposes the concept of virtual hardware where circuits are swapped on and off FPGAs. The system uses a library of pre routed, pre-placed cores to be invoked on demand. It supports global RTR and uses partial evaluation to determine static circuitry at compile time, thereby freeing up some routing resources. The virtual hardware manager deals with communication between application and FPGA to pass circuits, results, check for interrupts and core execution.
Chapter 3

Concepts and System Model

We describe in this chapter the concept, system model, and hardware & software issues involved in its implementation. We start with our basic model in this chapter and later in the next chapter we discuss the optimizations done on this model. Our final system is a multi threaded run time environment, which can support execution of multiple cores on FPGA. The basic model is proposed by the author and K. Mehra [2].

3.1 Goals and Concept

Fast runtime reconfigurable hardware enables system designers to swap hardware into and out of an FPGA much as the pages of virtual memory are swapped in and out of virtual memory. We have tried to exploit productivity of software design with the performance of hardware design. The goal of such a system is to isolate an application programmer from issues involved in managing hardware objects such as loading, swapping and executing hardware objects.

Figure 5. Top Level View of Reconfigurable System.
We have developed runtime environment supporting demand based reconfiguration of FPGA. Figure 5 shows the top level view of our system. As shown in Figure 5 our RC (Reconfigurable Computing) system takes application written in software language (Java in our case) and its outputs are the applications executed on the FPGA.

3.2 System Model

The input program is represented in an intermediate format described in later sections. The program can have two types of modules: data modules and control modules. A data module consists of one or more cores where each core is a set of assignment statement in the input program. A typical program consists of three basic structures 1) a sequential set of assignment statements 2) condition statements and 3) loop statements. Our system has the capability to handle all these constructs in hardware. Sequential statements are of the form

<variable> = <expression>

where an expression is of the standard form incorporating any number of variables, operations and matching parentheses.

Condition statements are of the form

If <condition> then
    <true-part>
else
    <false-part>
Loop statements take the form of

\[
\text{<for loop> ::= for( <expression1> ; <expression2> ;<expression3>) <statements>}
\]

and

\[
\text{<while loop> ::= while( <condition> ) <statement>}
\]

A detailed discussion on how to use these constructs has been provided by K. Mehra [2].

Modules are configured in the VHM (Virtual Hardware Manager), downloaded to, and executed in the FPGA in the execution sequence of the input program i.e. the schedule. A detailed description of the VHM is provided in section 3.4. Each module is a single thread of control, that is, each module executes sequentially on the board. Thus the application is partitioned \textit{temporally} for execution. The task of temporally partitioning an application into a collection of modules has been widely investigated [12,13] and thus we take the view that a partitioning is already made and, for the purpose of our study we start from a defined schedule of modules. Figure 6 shows a set of modules with the control modules shaded gray. The decision of which module to execute is made dynamically at run-time based upon the current state of execution.

As the system is demand driven once a core is executed on the FPGA it raises an interrupt and the VHM tracks it and loads the next core depending on the execution sequence. Figure 7 shows the overview of our system. The cores used in our system are pre placed and pre routed functionality that an application programmer has to develop in house or buy that from third party. The key feature to these cores is that they are relocatable. Theses cores make up the components that a designer can use to implement a
given design. We have developed some of these cores and utilized some of the cores already present in JBits library.

**Figure 6. Conditional Execution [2].**

Another key feature of these cores is that they are runtime parameterizable [1]. These are an extension to traditional static core model. Written in Java programming language, RTP (Run time programmable) cores are created at runtime and may be used to dynamically modify existing circuitry. This adds flexibility and portability to design. One example of such a core could be adder circuit. When requesting an adder core, a user would be asked to specify the bit width of the adder. With a library with fixed cores we would need sizes, typically 4 bit, 8 bit and 16 bit. This can be avoided by using parametrizable cores which can be customized at runtime and we need to maintain only one generic adder circuit in our library.
Figure 7. Overview of Our System [2].

3.3 Intermediate Format Representation

By providing this intermediate form we have tried to provide a standard method for interfacing to a reconfigurable coprocessor. Our assumption is that hardware compiler will convert the high level language to this intermediate form, which can be understood by our system. The application is represented as a sequence of modules executing one after another. To accomplish this we need to provide a flexible data structure to represent
modules in the run-time support environment. For each module attributes are stored in a
data structure. Some of the main attributes are

1) Physical dimensions: length and width of module.
2) Module ID: identifies the module in the schedule.
3) Inputs and outputs.
4) Conditional data for control modules

A detailed description is provided by K. Mehra [2] in his master’s thesis.

3.4 Virtual Hardware Manager

We have developed a virtual hardware manager (VHM), which takes care
of following tasks.

a) Place data / control modules on FPGA board.

b) Abstract the routing of the cores and control structures from the designer.

c) Detect module completion events.

d) Interface with actual board to download bitstreams, store intermediate value.

e) Keep track of resources on FPGA.

f) Initiate partial reconfiguration of board as and when required.

As described earlier, the VHM is the main component of the system entrusted with the
critical tasks aforementioned. Calls made to the VHM place a data/control module on
FPGA, route it to the control flow structures, execute it and free up the resources after its
usage. A close analogy can be of a kernel in an operating system. We view our run time
environment as an operating system for the FPGA, which executes on the host computer
and the VHM performs the task of the kernel. Figure 8 gives an overview of tasks
performed by the VHM.
Figure 8. Tasks Performed by VHM.
3.4.1 Placement

One of the main tasks of the VHM is to guarantee successful and efficient placement of a requested control/data modules. The dynamic nature of the system has a strong effect on the nature of placement algorithm we choose to implement. We will deal with placement and routing algorithms in subsequent chapters in detail. In case of no space being there on the FPGA to be allocated, a stall occurs. Main objectives of placement routine are

a) Compactness of placed modules with minimum number of unused CLB’s (Configuration Logic Block). We would not like to have cores scattered around the FPGA as it would finally lead to fragmentation and low utilization of resources.

b) This is a run time environment where all the placement coordinates are decided at runtime. We would like placement algorithm to be as fast as possible. As a result we have tried to use deterministic algorithms as adaptive algorithms, though efficient take more time to converge.

3.4.2 Resource monitor

The VHM has a resource monitor that keeps track of free resources on board. To facilitate this we store the CLB information of the device in a matrix, the allocation matrix, indicating whether it is occupied or free. On the basis of free resources placement coordinates are found. Once a resource has been allocated, resources monitor marks those resources as used. Once a core has finished execution, resources are freed and corresponding entry is toggled by resource monitor.
Figure 9. VHM Execution Flowchart [2].
3.4.3 Interrupt Mechanism

Each data path is associated with a control path. When data path has finished execution, control path circuit (state machine) gives a one at its output. The VHM tracks this interrupt. Usually a counter can be used as a state machine. We have modified the circuit, so it takes less number of CLB’s. This is also a run time parameterizable core whose size varies on number of clocks it has to count. Figure 10 shows the circuit.

![Figure 10. A Finite State Machine.](image-url)

For sake of consistency we will call it a Loop clocking unit (LCU) as other member of my team has used this name [2]. We poll the output pin of comparator after every few clock cycles. We have changed LCU circuit such that once we get a ‘1’ at output it will always show a ‘1’ indicating corresponding data path has finished it’s execution. By not polling every clock cycle we reduce the overhead and also makes it easier for multi threaded applications to run. We will talk about multithreaded environment in chapter 5.
3.4.4 Core Eviction and Loading

Once a core has finished execution, the VHM evicts the core, freeing up space. There is no replacement policy used. K. Mehra [2] in his master’s thesis has discussed replacement policies. The VHM loads up new cores and corresponding state machines for the next state in execution sequence. After loading it, initiates partial reconfiguration of FPGA so that changes are reflected on FPGA.

3.5 Handling High Level Language Constructs

The other team member, K. Mehra handled this part of project [2]. I will present a brief overview of these techniques.

Implementation of Loops

We have implemented the loop construct by first designing a basic loop control block called the Loop Clocking Unit (LCU). It takes as input the number of times the loop has to be stepped and outputs a one, once its main body gets executed. We have shown a LCU above in Figure 10. After execution of loop once, LCU is incremented. For nested loops, each successive loop structure has its own corresponding LCU, the output of which is connected to its parent loop’s LCU and so on upwards till the outermost loop. Once the innermost loop gets done executing, it sends a one up to its parent LCU. Figure 11 shows the nested loops.

```
For(i=0 to m)
  For(j = 0 to n)
    For(k = 0 to l)
      For(p = 0 to a)
        { main body }
```
Implementation of condition constructs

The condition constructs i.e. ‘if then’ are implemented using a simple comparator RTP core from the JBits library. Each of these conditions are evaluated individually in hardware and we perform a logical computation on the results based upon the operators specified in the module.
Chapter 4

Optimizations

Our basic model as discussed in previous chapter is slow and does not exploit parallelism inherent in hardware design. We will be discussing various optimization strategies in storing data and use of threads to execute multiple cores at runtime.

4.1 BRAM Vs Registers

We have used on chip BRAM to store data as described in previous chapter. One or more data items are simultaneously needed at input of data path. All input items are stored in BRAM which is singly addressed, thus must read all items from memory before data path execution. Addresses are applied sequentially and data is stored in input registers.

Variables in the module are memory mapped, where each datum is stored at a predefined offset in BRAM. The data-out width of BRAM is related to the address width. Data out width can be 1, 2, 4, 8, or 16 bits. The Block RAM itself has bit 0 being the most significant bit. Archiving data in BRAM turns out to be a bottleneck for our applications. Figure 12 illustrates our initial architecture of reading datum in BRAM. Since this is a brute force approach multiple partial reconfigurations are required for reading input data from BRAM. This can be attributed to change in routing from BRAM output to input registers. The problem becomes more acute as number of inputs increase.
There are various strategies to improve this situation.

a) Use of multiple BRAM’s. It can be seen as interleaved memory. We can supply multiple addresses and thus get all the inputs at same time. We don’t have to do multiple reconfigurations with this approach.

b) Use of Demultiplexors. We only have to change the control signals of demultiplexer. No need for repeated reconfiguration arise. Figure 13 shows the new design for cores with two input buses.

Figure 12. Architecture for reading data from BRAM [2].
c) Use of registers only. A data structure is maintained by the VHM that keeps a mirror image of values of registers on FPGA only when transitioning between modules. For each new core being executed on the FPGA, input registers are instantiated with value stored in the data structure by the VHM.

![Diagram of BRAM with Demultiplexer](image)

**Figure 13. BRAM with a Demultiplexer.**

d) We can maintain a set of input registers on FPGA at a fixed location. Each register stores value of a particular data. Each data variable is stored at a particular offset (address) on FPGA.

We analyzed all the four approaches and fully investigated the demultiplexer and register based approach. Multiple BRAM approach becomes infeasible if you have many variables. We have a limited number of BRAM’s on FPGA. It is also difficult to keep
track of variables on each BRAM. With multiple threads, with each thread updating the data sets it becomes difficult to keep track of latest value of variable on BRAM’s. Global register set approach is difficult to implement in applications that use lot of variables. Moreover, a lot of resources will be locked and not available to placement routine. Further research is required to fully evaluate this approach. Experimental results suggest that demultiplexor based is attractive when input size is small. We have developed a core for demultiplexer that can handle different widths for two input bus. We have limited our approach to two input buses. As first address is supplied the output from BRAM goes to first input of the core. With next address, control logic is changed and the output from BRAM goes to the next input of core. The core is generic core. Figure 14 shows the building block of the demultiplexer core.

**Figure 14. Building Block of a Demultiplexer.**
As the input bus becomes larger (bus width increases) more of these blocks can be added in parallel. But it can be seen for an input bus whose width is eight bits eight such blocks will be required. This becomes bulky and occupies lot of resources on FPGA.

Approach ‘c’ is the best approach as shown by results given in chapter 6. Registers are instantiated on FPGA depending upon the input bus width. A data structure is maintained by the VHM that updates value of data variables once a core has finished execution. The value from output registers is used to update the data structure maintained by the VHM. All the input and output registers for a core are removed, once the core has finished execution thus freeing up resources. Figure 16 illustrates the idea. The approach ‘c’ (Register based approach) has shown to take up less number of CLB’s on FPGA compared to approach ‘d’ (Global Register approach) and does not require multiple reconfigurations of FPGA while supplying data to cores. This approach when compared to any BRAM (approach ‘a’ or ‘b’) based approach occupies more resources on FPGA. It is a space / time conflict. As we have a runtime system we have given priority to faster execution time on our system.

![Diagram of I/O in Register Based Approach](image)

**Figure 15. I/O in Register Based Approach.**
4.2 Multithreaded Environment

The loading and execution of single hardware object or core can lead to wastage of resources. From the software perspective, the closest analogy to a hardware object is a thread in a multitasking operating system. Figure 16 summarizes these similarities.

4.2.1 Implementation of Multithreaded Environment

We have provided a mechanism in our system where multiple threads can be spawned from the main execution thread. The main thread or the VHM is created on program execution and is the parent thread for all threads that will be initialized later.

<table>
<thead>
<tr>
<th>THREAD</th>
<th>HARDWARE OBJECT (CORE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begins execution at a predefined location.</td>
<td>Begins location in a predefined state.</td>
</tr>
<tr>
<td>Single minded, executes linearly from start to end.</td>
<td>Single minded, executes continuously until complete.</td>
</tr>
<tr>
<td>Independent of other threads.</td>
<td>Independent of other hardware objects.</td>
</tr>
<tr>
<td>Executed in pseudo parallel.</td>
<td>Executed in parallel.</td>
</tr>
</tbody>
</table>

Figure 16. Threads Vs Hardware Objects.

Each child thread in turn is responsible for following tasks

a) Loading of core and corresponding control structure (FSM) on the board.

b) Providing inputs if there are no dependencies between this core input’s and other cores already running on FPGA. This is necessary to prevent data hazards.

c) Polling for completion event of core.
d) Sending a signal to main thread (VHM) that it has finished execution.

Figure 17 shows life cycle of a thread.

The main thread is the VHM. Now depending upon the flow graph (intermediate form in our case) the VHM can spawn as many threads as required. It is duty of the VHM to initialize threads. Some of tasks that the VHM used to perform related to loading, execution and swapping of hardware objects are now passed to child threads. The VHM still keeps track of all variables and updates data structures containing data variables. The VHM plays role of a manager keeping track of all the spawned threads. If there are multiple requests for same resource, the VHM has to ensure that multiple threads do not acquire the same resource, child threads have to wait for their turn before acquiring a resource.

Performance is improved when actual configuration of the hardware is overlapped with computations performed by the host processor, because programming the reconfigurable hardware requires milliseconds to seconds to accomplish. Threads and Cores can execute in parallel. For example a core can be loading while one or more cores are executing. Each core is associated with a separate thread. This hides the configuration time from program execution time
Figure 17. Life Cycle Of a Child Thread.
4.2.2 Advantages and Disadvantages of a Thread-Based Approach

With a multithreaded environment the performance of the dynamically reconfigurable system improves dramatically. Even if the cores are not being executed they can be combined with other cores on FPGA thus potentially saving on partial reconfiguration time.

A major problem with threads is synchronizing them. Suppose two threads want to do an I/O it leads to error in our system. Similarly two threads cannot simultaneously do a partial reconfiguration of FPGA on our system.

When two or more threads need to access a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Java provides language level support for it. Key to synchronization is the concept of monitor (also called as a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. Other threads wait for the monitor. We have used semaphores to control hardware access by threads. Figure 18 illustrates this idea.

Multiple threads reduce the time of execution of application. The relevant results are shown in chapter 6. Multi threaded environment also gives us the flexibility of running concurrent applications, even though we have not tried that in our system. It might be good idea to divide basic blocks in a data flow graph (execution sequence). A basic block is defined as a sequence of instructions with a single entry and a single exit. There are no branch targets or labels except perhaps at the first instruction. There are no jumps outside the basic block except at last statement. We can try to divide a basic block
in such a way that instructions taking same number of clock cycles be treated as one basic sub block in flow graph. Each thread will take care of each such basic sub block. This means that we can remove the sub blocks that have finished execution from the FPGA. This will free up resources for other cores yet to be placed. We leave these issues on the person developing Hardware Compiler and the Intermediate form needed by our system.

![Diagram](image.png)

**Figure 18. Controlling Resource Access with Semaphores.**

As will be shown in Chapter 6 having too many threads might deteriorate the system performance. It is difficult to predict the number of threads that will provide best system performance.
Chapter 5

Placement and Routing

Placement and Routing has a major impact on the overall execution time of dynamically reconfigurable system. The time spent in placing and routing for execution is above and beyond that required for actual problem execution. They must therefore be performed as quickly as possible even if results are sub optimal. The logic circuit that is to be placed and routed is specified in terms of reconfigurable logic blocks, which are the basic logic elements that make up a FPGA. Placement is assigning a unique position inside the FPGA to each of the circuit’s configurable logic blocks. Routing a signal is essentially assigning routing resources such that all sinks are connected to the source. Since the number of cores on FPGA is changing constantly in a dynamically reconfigurable runtime system a time efficient algorithm to place and route is needed.

5.1 Routing

A common goal of most routers is to accomplish complete automatic routing using as short a wirelength as possible. In a dynamic runtime environment, an added constraint of small execution time for routing is imposed. Further minimal geometries must be maintained. These include minimum width and spacing of routing paths, which are dictated by technological processes [19].

Unfortunately, the details of FPGA interconnect are hidden in JBits from the user. Therefore it would have been very difficult to implement a router (if not impossible) that satisfies the constraints. JBits provides an auto router, JRoute that performs the auto routing. The auto router provided by JBits uses A* search algorithm. Unfortunately goals
of the JBits auto router are not known. We would have preferred *minimum time* for routing as a goal for auto routing.

JRoute is a tool that provides an abstraction layer for performing routing on a Virtex FPGA. Two kinds of routing, Template routing and Auto routing are provided to the user. Template routing is an abstraction above manual connection but below auto-routing. It provides the user the ability to specify the general route to follow, including the types of resources to use and the direction those resources go, without having to specify the specific resources to use.

### 5.2 Placement

Ideal characteristics for placement algorithms suited for dynamically reconfigurable system are to execute as fast as possible and to place as many functions as possible in the given area [24].

Placing a core on a FPGA is a dynamic overhead activity (in contrast to static placement which occurs before any execution takes place). The placement is NP-complete and therefore takes inordinate amount of time. Placement algorithms can be broadly classified into two major categories: constructive and iterative [19]. Constructive placement algorithm construct a solution by placing one cell at a time and thus are greedy in nature. Iterative algorithms such as simulated annealing follow a more sophisticated strategy (non greedy approach) by choosing suitable perturb and accept functions [19]. Obviously iterative algorithms take much more time than deterministic algorithms.

In the proposed system constructive algorithm is used rather than iterative as iterative algorithms take time to converge. Two well known algorithms for runtime placement are First Fit and Best Fit [24]. Another fast algorithm is the Bottom-left
heuristic [25] implemented in total quadratic time, but the algorithm cannot be applied to reconfigurable systems. We have evaluated our algorithm against First Fit and Best Fit.

The Semi-Perimeter method is used as a cost function in the proposed algorithm. While considering routing distances only distances between two successive cores are considered. This helps in reducing search space (makes it faster) and ensures that data path, control path and input & output shift registers are close together. If all the blocks are near each other routing distances are considerably less. Moreover for nested loops, one state machine interacts with another [2]. Placing them close to each other results in smaller routing distances.
Figure 19. Flowchart for First Fit.
First Fit starts searching for placement coordinates on the FPGA from top left corner. The module is placed on the first set of coordinates found. Figure 19 shows the flowchart for First Fit. First Fit tends to be very fast in a system where cores are moving in and out of the system. Relevant results are shown in chapter 6.

Best Fit looks at all possible coordinates on the FPGA where core can be placed. From all those coordinates one that has minimum distance with respect to some reference coordinates (previous core coordinates are used) is chosen. Figure 20 gives a flowchart for this algorithm. Best Fit strategy is evaluated and results shown in chapter 6. The advantage of using First Fit and Best Fit is that they are easy to implement. First Fit is extremely fast but results in larger routing distances. It does not take into account the placement coordinates of the previous core before placing the new core. This leads to larger routing distances between successive placed cores.

The algorithm used in our dynamically reconfigurable system is a deterministic algorithm. It’s search mechanism is analogous to dropping a pebble in a still pond and causing waves to ripple outwards. The algorithm takes as input a set of reference coordinates indicating the placement of the last placed core and the length and width of the core to be placed. The search starts around the reference coordinates with radius equal to one. If desired coordinates are not found the radius is increased by one. The Figure 21 below illustrates the search mechanism. The process continues till either radius is greater than width of chip or desired coordinates are found. Figure 21 shows radius of one with dotted lines and radius of two with bold lines. When placement is successful, the allocation matrix is updated for the length and width of the module.
Figure 20. Flowchart for Best Fit.
Figure 21. Placement of 2 X 1 module.

In Figure 21 grey shaded blocks indicate the placement coordinates and black shaded blocks indicate that resources are already occupied. Our algorithm exploits the fact that a good packing density is needed on any FPGA based system. As a result it starts the search around the last placed core. A cluster growth is attempted in our algorithm. As orientation (input and output location of pins) of CLB’s is fixed in a Xilinx Virtex FPGA, orientation of CLB’s is also considered in choosing the ideal placement coordinates. Multiple placement locations at the same radius are possible to fit in a new core. It is possible for different coordinates at the same radius to occupy different amount of routing resources. Figure 22 illustrates one such scenario. Suppose a new core can be placed either at 5 or 3. As output of 5 compared to output of 3 is nearer to input of 1 we choose location 5 for placement of core. We give preference to CLB 5 than 3 in our algorithm. Figure 23 and Figure 24 depicts working of our algorithm. CLB marked ‘R’ shows the reference CLB. In all iterations in our algorithm two successive wavefronts or
radius are considered before allocating the final placement coordinate. There can be cases where CLB’s located at a radius of \( r + 1 \) take less routing resources than CLB’s located at radius of \( r \). Figure 25 shows one such scenario. Here the assumption is channel width is considerably less than width of the CLB. Another assumption is all pins on CLB are at center of CLB. Figure 26 illustrates this point. Assuming all CLB’s of size 1 X 1 in Figure 25, we get a manhattan distance of 3 from CLB 1 and manhattan distance of 2 from CLB 2.

![Diagram](image)

**Figure 22. Choice of Placement Coordinates.**

![Diagram](image)

**Figure 23. Working of Our Algorithm –1.**
Figure 24. Working of Our Algorithm –2.

Figure 25. Checking Successive Radius for Optimum Placement Coordinates.

Figure 27 shows the pseudocode for our algorithm. Evaluation results comparing our algorithm with other algorithms considered are shown in Chapter 6.
Figure 26. Pin Location for CLB’s in Our Algorithm.

1) Get Reference coordinates and set initial radius R to 1
2) Do for( each CLB in a square of side –R to R)
   {
     2.1) Search for a contiguous space of m X n in the given radius at each CLB. Store obtained coordinates in a width array. If space not found, break and do R++.
     2.2) For each pair of coordinates in the array find minimum distance with respect to the reference coordinates.
   }
3) Return coordinate pair and update the allocation matrix.

Figure 27. Pseudocode for Our Algorithm.

As shown in Chapter 6 results obtained from our algorithm are better than other constructive algorithms, First Fit and Best Fit. Results indicate performance of our algorithm is optimum both in space (Unused CLB’s) and time.

All the algorithms described above are trying to optimize the packing density that might lead to congestion in some cases. A study of these issues has been left out for future research. Similarly no effort is made to deal with fragmentation issues. Eviction of cores from FPGA may leave small holes (unused CLB’s) that lead to wastage of CLB’s.
However by moving cores such that all the unused CLB’s are placed together the CLB utilization on the FPGA can be improved. Our algorithm does not take these factors into account. We leave this for future research.
Chapter 6

Simulation and Results

We present in this chapter results of simulations performed on our system. A description of simulation environment is provided before presenting the results.

6.1 Simulation Environment

JBits program flow is used to evaluate our dynamically reconfigurable system. We were not able to use the actual FPGA due to lack of resources so we have used VirtexDS, a Virtex device simulator. The various steps in the JBits program flow include, creating a JBits object, reading the bitstream into the JBits object, modifying the bitstream with design data and writing out the design bitstream. These steps are discussed below [25].

1) The Constructor: First Step is creation of JBits object. The constructor builds the device model for the selected part and performs various initializations.

2) Read Bitstream: This method takes a single parameter, a string containing the name of the bitstream file to be read. This method loads the bitstream into the constructed JBits object and maps the bitstream data into the device model. Once a bitstream has been loaded, configuration data in the form of bits may be read and written.

3) Set bits: This method writes the configuration data of a given resource in the configurable element. The resource is identified by a CLB row and a CLB column. The resource in the selected CLB is then identified by a constant.

4) Write bitstream: It is similar to the read bitstream method, the write bitstream method takes a single parameter, a string containing the name of the bitstream
file to be written. This method will write the bitstream from the constructed JBits object into a file.

5) Get bits: The get() method is used to read the configuration of a given resource in a configurable element. The resource is identified using the same convention mentioned in the set() method for example configuration of the resource **SLICE0 F1** input is accomplished by using the **S0F1**.

Xilinx Hardware Interface (XHWIF): XHWIF provides various methods to send/receive data to/from the FPGA board/simulator. To read more about XHWIF refer Appendix.

### 6.2 Performance Evaluation Methodology

JBits has limitations as given in Chapter 2, that it does not support any type of timing analysis. VirtexDS is an event driven simulator but it does not incorporate actual physical delays into its model. Unit delays are used for modeling, so this gives us only a relative estimate of the reconfiguration and execution time. While measuring execution time we have measured total time required to simulate the application. This incorporates other factors like time required by JBits to assemble and modify the packets. Moreover other factors like PCI transfer speed, host processor speed, amount of system memory etc cannot be modeled as we have a simulator not a actual FPGA. Therefore a factor ‘K’ is introduced, where all the times are scaled by ‘K’ factor. The scaled execution times will represent the actual time. In our present setup it is not possible to accurately predict the value of ‘K’. The simulation environment used is:

Hardware: AMD 1.0GHz Athlon processor running Red Hat Linux

Software: JBits 2.8 with the VirtexDS FPGA simulator.
The workload in our experiments consists of various size cores chosen at random.

6.3 Placement Results

Different sizes of cores were placed on the FPGA. We started with empty FPGA and went on adding cores till the FPGA was full. The ‘K’ factor reflects factors like load on machine, system memory, cache etc. It has been observed in our simulations that depending upon ‘K’ value, actual times change but the ratio between them remains constant.

![Placement Time vs Size Graph](image)

**Figure 28. Placement Times For Various Algorithms.**

As can be seen from Figure 28 Best Fit is the slowest algorithm in our simulations for large sizes of FPGA. First Fit and Our Algorithm give comparable performance but as
size increases our algorithm is slightly faster than First Fit. This is because First Fit starts from top left of FPGA and goes down considering all the empty spaces (unused CLB’s).

Figure 29. Number of Unused CLB’s.

Figure 29 shows First Fit uses space on FPGA most efficiently. This means that number of CLB’s that are unused after all the placement calls is minimum in First Fit, that is, the level of fragmentation in First Fit will be least compared to other algorithms considered. Performance of Best Fit and Our Algorithm is similar (95% confidence level at FPGA size of 20 X 20 and above).

The total Manhattan distance obtained by each of the algorithms divided by the number of placement calls required to fill the FPGA completely gives the Average routing distance per placement call. This ensures that we are plotting the routing introduced per placement call that gives better picture of how routing density changes
with various algorithms. As seen from Figure 30 given below First Fit performs badly compared to other algorithms. Performance of our algorithm is similar to Best Fit (95 % confidence level), which generally provides the best performance in terms of average routing distances.

![Figure 30. Average Distance per Placement call Vs Size of FPGA.](image)

Based on above observations we discarded Best Fit algorithm, as it was too slow. Our system is a run time system and one of the major objectives is to have a very fast placement algorithm, especially when size of FPGA is large. In an actual system cores are evicted from FPGA when their execution is over. We did some simulations based on random eviction of cores from FPGA. Each core was assigned some random time, for that particular time it is active on FPGA. After that time is over resources occupied by that core are freed from FPGA and subsequent cores can be placed there. We have not
assigned any replacement policy, so as soon as core execution is over they are evicted from FPGA.

![Time vs Size Graph](image)

**Figure 31. First Fit Vs Our Algorithm with Core Eviction.**

Total time required to fill the FPGA is still more in First Fit as shown in Figure 31. But in the above graph number of placement calls is not considered. From the experimental data it can be seen that number of placement calls are more in First Fit. Taking this into account we plot the average total time defined as total execution time divided by number of placement calls made for one particular FPGA size. Figure 32 shows the related graph. From the graph it can be seen that the average time required per placement call is still more for First Fit for all the sizes of FPGA. This leads to the conclusion that First Fit will be slower than algorithm we have used in our system.
Figure 32. First Fit Vs Our Algorithm.

In an actual runtime system cores are continuously being loaded on FPGA or being evicted once execution is over (assuming our replacement policy is to evict as soon as execution is over). In Figure 33 and Figure 34 we have compared performance of our algorithm with and without core eviction. It has been observed, behavior (space and time requirements) of our algorithm does not change appreciably when we have cores actually moving in and out of the FPGA. Figure 34 validates this fact. Total average time per placement call using our algorithm remains almost the same when compared to our system with no runtime eviction of cores. Figure 33 shows that total average distance per placement call (routing introduced per placement call) increases but as shown in later graphs it is lesser than First Fit’s increase with core eviction.
Figure 33. Our Algorithm With Core Eviction -1.

Figure 34. Our Algorithm with Core Eviction -2.
Figure 35 and 36 show the change in average time and distance per placement call when First Fit is run with and without the policy of core eviction at runtime. As can be seen from Figure 35 First Fit runs faster when cores are being evicted (95% confidence level). This means in some cases First Fit will find a space to allocate near top left where it starts it’s search. This reduces the average time per placement call. On the flip side we see in Figure 36 that the average distance per placement call increases (95% confidence level). One of the drawbacks as seen in Figure 30 was that First Fit takes lot more routing resources. This behavior becomes worse with a runtime eviction policy. This leads to conclusion that even though the execution times are comparable with our algorithm, routing distances are not.
Figure 35. First Fit With Core Eviction -1.

Figure 36. First Fit With Core Eviction -2.
6.4 Evaluation of Data Storage Techniques

An efficient way of storing data has been proposed in Chapter 4. The relevant results of our simulations are presented in this section. Random sized cores containing two input buses (2-4 bits wide) were placed and executed on FPGA. The results are given in Figure 37. The origin of factor ‘K’ has been explained in section 6.2. As the number of input buses is increased to three, total execution time for BRAM based approach increases, as shown in Figure 38. This is because it increases the number of partial reconfigurations required. A small increase in time is also seen for Register based approach. This may be attributed to a larger configuration file as the number of registers increase. But compared to increase in BRAM based approach this increase in time is significantly less.

![Figure 37. Time required by BRAM Vs Register based approach for two inputs.](image-url)
Figure 38. Time required by BRAM Vs Register based approach for three inputs.

Figure 39. Execution Time with BRAM, Register and Demultiplexer Based Approach.
Demultiplexer based approach give results similar to Register based approach. Figure 39 shows results for two input (2-4 bit wide) buses. In this approach control logic becomes complicated as number of inputs are increased. Larger bus width results in a much larger circuit. The basic block shown in Figure 15 is repeated multiple times to get the desired effect i.e. if we have 2 bits in each of the two buses, two basic blocks (as shown in Figure 15) are placed in parallel. This decreases the CLB’s available to the placement routine. Moreover registers at input end are still required to store the value of inputs being supplied to data path (cores). This leads to wastage of resources and offers no particular advantage over Register based approach.

6.5 Evaluation of Multithreaded Environment

Our final system is a multithreaded system that can keep track of multiple cores being executed on FPGA. Figure 40 shows how increasing the number of threads can increase the execution speed. Six random chosen cores are executed in our dynamically reconfigurable system with different number of threads each time. As can be seen in Figure 40 execution time decreases as the number of threads increase but after reaching minima it starts increasing again. This suggests that increasing the number of threads arbitrarily will not provide better performance as there are certain overheads associated with using threads. Inter thread communication and resource contention leads to increase in execution time. Once a thread locks a resource, other threads sleep for some predetermined time. If there are more threads, synchronization and resource contention eats up time. It requires knowledge of application to decide upon the number of threads required in a particular application.
Figure 40. Execution Time with Different Number of Threads.

Figure 41. Execution Time in Different Size of Applications.
Figure 41 clearly shows that minima is achieved at different points with different number of applications. It is not possible to predict how many threads will result in minimum time. A careful analysis of application is required.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we have proposed a runtime multithreaded support environment for execution of FPGA based applications. We investigated various placement algorithms for this runtime system. Various data and state storage techniques were studied and evaluated.

A dynamic demand based runtime support environment was developed and simulated using JBits API. VirtexDS was used as a FPGA simulator. A virtual hardware manager capable of performing tasks like placement, execution and swapping of hardware objects in and out of FPGA was designed. Proof-of-concept was evaluated by running various applications [2].

Various placement algorithms were studied and evaluated. The algorithm developed by us and subsequently used in our system performed better than other algorithms in terms of time and space complexity. First Fit was found to be a good algorithm in terms of time complexity but performed poorly when space considerations (resources used) come into picture. Best Fit was found to be too slow for our runtime system, especially when FPGA size is large.

Different data storage techniques were evaluated. The Register based approach was found to be suitable for our system as it was fast and occupied less number of resources on FPGA. BRAM based memory mapped variable approach was found to be slow.
Multiple threads supporting concurrent execution of cores on FPGA provided better execution time in our system. It is difficult to estimate the number of threads that would lead to minimum execution time.

7.2 Future Work

As of now we have a runtime multithreaded support system capable of supporting multiple core execution using JBits and VirtexDS. We have not used real hardware for testing our system. The system we have developed can be ported to real FPGA board as XHWIF interface used in our system can be used to interface to any Virtex based FPGA board. Some external hardware interface routines specific to the board need to be added, when this system is ported.

Some of the future work and optimizations could include:

- A hardware compiler that will generate the intermediate form we are using in this system. Our System performance will depend on the optimizations the hardware compiler can provide. It needs to extract maximum parallelism for multithreaded environment to be effective.

- A better scheme is needed to avoid excessive data transfer between our system and FPGA. A better polling mechanism by threads is needed. Study can also be done on finding some theoretical way of predicting number of threads in runtime system that will lead to minimum execution time.

- A replacement policy is needed. We have not studied the effects of replacement of modules in our system. Effective replacement policy might reduce the total execution time in our system. Some way of caching the configurations can be studied.
• A detailed analysis of speculative loading can be done. This can promise benefits in terms of reduced execution times. We have used a simple strategy where we try to avoid RAW (Read after Write) hazards. A thorough study of hazards is required before prefetching can be done completely.

• Fragmentation leads to low utilization of resources. A study can be done on compacting the unused resources, on the lines of operating system.
Bibliography


Appendix

Most of the material in this chapter is taken from JBits documentation [25]. It is intended to give reader a background on JBits, Virtex DS and XHWIF.

Virtex Architecture Overview

Virtex FPGAs are composed of an array of Configurable Logic Blocks (CLBs) surrounded by a ring of Input/Outputs Blocks (IOBs). On the east and west edges are Block RAMs (BRAMs). The CLBs are the primary building blocks that contain elements for implementing customizable gates, flip flops, and wiring for connectivity. The IOBs provide circuitry for communicating signals with external devices. The BRAMs allow for synchronous or asynchronous storage of kilobits of data, though each CLB can also implement synchronous/asynchronous 32-bit RAMs. Figure 42 shows the architecture overview.

CLB Overview and Slice Internals

Each CLB contains two slices. Each slice implements 2, 4-input Look-Up-Tables (LUTs), 2 D-Type flip-flops, and some carry logic. A Virtex slice is similar in functionality to a Xilinx XC4000 CLB, which means that each Virtex CLB has roughly twice the logic capacity of a XC4000 CLB. The general routing allows data to be passed to or received from other CLBs. The input mux's allow wires in the general routing to pass data to the slices, while the output mux's allow the slices to pass data to wires in the general routing. The primary elements in the slices are the F and G LUTs, and the X and Y flip flops. The LUTs can be used to implement gates or to implement small memories. The flip flops can be used to create state machines. The slices also have internal mux's to
control the connectivity of internal resources. Finally, there is logic inside each slice to implement fast carries for arithmetic type logic. Figure 43 shows the CLB overview.

![Figure 43: Virtex Architecture Overview](image)

**Figure 42. Virtex Architecture Overview.**

JBits API may be used to construct digital designs and parameterizable cores that can be executed on Xilinx Virtex FPGA devices. The API provides the lowest level interface to the Virtex architecture and thus it can also serve as a base to construct traditional circuit placement and routing, as well as application-specific tools to perform more narrowly-defined tasks. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. This provides the capability of designing, modifying and dynamically modifying circuits for Xilinx Virtex-series FPGA devices. This capability is acheived by providing access to all the resources of a Virtex device. That is, the API provided gives access to the Look Up Tables (LUT)
inside a Configurable Logic Block (CLB) of the Virtex device and to the routing resources of the Virtex device. The device architecture is represented as a two-dimensional array of Configurable Logic Blocks in JBits (CLBs). Each CLB is referenced by a row and column, and all configurable resources in the selected CLB may be set or probed. Control of all routing resources adjacent to the selected CLB are made available in the API.

Figure 43. CLB Overview.

*JBits* design flow is different from traditional CAD tools. Unlike the conventional design flow, that uses HDL or schematic design entry, a design is specified in a Java program using the *JBits* API and/or *JBits* cores. The application takes a bitstream file as
an input and extracts the device configuration data. This data is modified according to the design specified using the JBits API and is output to a bitstream file that will be used to configure the Virtex hardware. Once downloaded to the Virtex hardware, the design can be debugged using the hardware debugger, Boardscope. The input bitstream can be a null bitstream or one for an existing design. When a null bitstream is used the JBits API is used to build the entire design. JBits can also be used to modify portions of a bitstream for an existing design. This enables the designers to start with the bitstream generated by the Xilinx design tools.

Another key feature of JBits suite is Virtex Device simulator or VirtexDS. VirtexDS simulates at the device/bitstream level, providing an interface which operates much like actual hardware. This approach not only support simulation for run-time reconfiguration, but also interfaces easily to JBits applications, like BoardScope. The Virtex Device Simulator provides users with the ability to test Xilinx Virtex bitstream files without the need for actual hardware. VirtexDS provides this versatility by implementing the XHWIF portable interface allowing for immediate use with the BoardScope GUI debug tool and any other application utilizing XHWIF. The bitstream files to be simulated can be created by the standard Xilinx tool flow or with JBits. VirtexDS has another interface besides the XWHIF interface, known as the Stimulus/Probe interface. This allows you to stimulate and probe wires and pins in the running simulator. This interface provides the basic functionality for writing test benches.

XHWIF is the Xilinx HardWare InterFace API provided with the JBits SDK. XHWIF API provides various methods to describe a FPGA-based board and to send data on and off the board. It includes methods for reading and writing bitstreams to FPGAs,
and methods for describing the kinds and number of FPGAs on the board. In the absence of a real hardware, XHWIF interface can be used to connect to the simulator, VirtexDS. The interface standardizes the way that applications communicate with hardware, so that using the same interface, applications like BoardScope for example, can communicate with a variety of boards. All of the hardware specific information is hidden inside of a class that implements the XHWIF interface. Using the Java programming language's Native Methods, XHWIF can communicate with hardware directly, through libraries or through a device driver.

We have used BoardScope during the course of our research to test functionality of cores. BoardScope 2.7 is a graphical, interactive tool for debugging designs on Xilinx Virtex FPGAs or VirtexDS device simulators. This tool provides hardware control such as downloading configuration bitstreams with circuit designs to devices, reading back configuration data, single- or multi-stepping the device clock and resetting the device.