Design of a Resource Manager and a Simulator for a Massively Parallel Dynamically Reconfigurable Accelerator

Thesis

Submitted in partial fulfilment of the requirements of

BITS F421T Thesis

By

B. Vignesh

ID No: 2011A8TS368P

Under the Supervision of

Dr. S.K. Nandy

Professor, CADL, Supercomputer Education and Research Centre, Indian Institute of Science

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMUS

15th May 2015
## Contents

1 INTRODUCTION

2 REDEFINE Resource Manager (RRM)  
  2.1 Modules of RRM: Resource Tracker (RT) and Memory Address Translator (MAT)  
  2.1.1 RT: Resource Tracker  
  2.1.2 Memory Address Translator  
  2.2 Interfaces  
    2.2.1 Signature of the Call Function  
    2.2.2 Sequence of Execution  
    2.2.3 Format of Requests  

3 Resource Tracker - A Detailed Description  
  3.1 Introduction  
  3.2 Legend for this section  
  3.3 State transition diagram  
  3.4 CASE I: One kernel at a time  
  3.5 CASE II: Multiple Kernels at the same time  

4 Implementation Details  
  4.1 Anatomy of Signals  
  4.2 Journaling activities of the Resource Tracker  
  4.3 REDEFINE Emulation on FPGA  

5 RESULTS  

6 CONCLUSIONS  

7 Appendix
ACKNOWLEDGEMENT

I would like to acknowledge my sincere thanks and gratitude to the people who helped me in completion of this thesis. To begin with I would like to express my sincere thanks to BITS PILANI for allowing me perform this thesis course. I would also like to thank CADLAB,IISc for providing me the opportunity of this internship. I would like to express my heartfelt gratitude to Prof. S. K. Nandy for his valuable guidance and continued support during the thesis without which this would not have been completed. I would also like to thank Mr. Abhijit R. Asati, my co-supervisor who has guided me at various stages of the thesis and has been a constant source of motivation. I would also like to thank my friends and project mates S. Balasubramanian and Divyaansh Anuj for the fruitful discussions we had which have given me a deeper understanding and insight into various topics. Thank You all for the immense learning experience through the making of this thesis.
1 INTRODUCTION

Mobile handsets are emerging as the unified embedded platform which supports a plethora of applications such as communications, multimedia, image processing etc. Such a vast range of applications require flexible computing platforms for different needs of each application and its derivatives. General Purpose processing platforms are good candidates for the flexibility they offer; but they cannot meet the critical performance, throughput and power criteria. Unlike traditional desktop devices, embedded platforms have stringent requirements on power and performance. On the other hand, customized processors and ASICs do not scale to accommodate new application features or application derivatives. The emergence of embedded devices have created a need for new generation of vertical processors which are necessarily “domain specific” to meet the performance and power requirements and sufficiently “general purpose” to handle such a large number of applications.

The embedded industry has successfully used the paradigm of custom co-processor to meet the power-performance goals of these devices. These co-processors are designed to accelerate certain specific instruction sequences which occur frequently in the application under consideration. But such solutions are useful for accelerating individual applications. In order to provide accelerators for an entire application domain, we necessarily need to support the following:

1. Domain specific Compute Elements (CEs) and Communication Structures: Applications that contain same kernels within them, exhibit similar computation and communication patterns. The similar nature of communication and computation in kernels allows us to determine suitable CEs and interconnect characteristics of an application domain well in advance.

2. Dynamic reconfigurability: The accelerators need to configure themselves at runtime to cater to the performance requirements of the particular application or its derivatives (in the domain).

This is where REDEFINE HPC comes into the picture. REDEFINE in a unique coprocessor that offers the two above stated benefits. REDEFINE can be thought of as the perfect hybrid between ASICs (which offer high performance at high cost of production)
and General purpose computers (which are relatively inexpensive because they are not optimized for any one application in general). In this thesis project, we strive to showcase the performance benefits that come along with using REDEFINE HPC.

The benefits of REDEFINE cannot be realized until we come up with a mechanism to efficiently send in code and data to the coprocessor. This is where we introduce the concept of a REDEFINE Resource Manager (RRM). One of the major tasks of the RRM is address translation. The concept of address translation, from that of the host to REDEFINE, comes into the picture because we have a requirement of sending instructions and data which are present in the address space of the host processor to the appropriate addresses on REDEFINE. The scope of this thesis project is to develop this resource manager. My role in the project was to design the Resource Tracker sub module of the RRM.

We designed experiments to show a proof of concept for our resource manager. We aim to perform a Hardware emulation of the entire system on a Virtex-7 FPGA board. In order to check the resource manager for functional correctness we designed a software simulator comprising of a Network-On-Chip (NOC) simulator and individual modules designed by us.

The report is organized in the following manner. In the first section, we introduce the basic functionalities of the RRM. In the next section, a detailed description of the Resource Tracker module is provided. In section 3, we provide implementation details for parts of the project. Section 4 provides the results obtained from the software simulation.
Host and REDEFINE HPC operate in a client-server model. Applications are compiled to run on the host. Calls to kernels (that need to be accelerated by REDEFINE HPC) are replaced by stubs in the compiled code of the application - stubs are calls to REDEFINE HPC device driver, running on the host. Kernels are cross-compiled (using the REDEFINE HPC compiler) to generate HyperOps. Compiled objects so produced, target the virtual address space of REDEFINE HPC. Note that the size of the virtual address space of REDEFINE HPC is $\leq$ the size of its physical address space. Note that the kernels maybe cross-compiled on any general purpose host (and not necessarily on the host to which REDEFINE HPC serves as an accelerator). Binaries (HyperOp and pHyperOp meta data) so produced need to be loaded to REDEFINE HPC (after the host redirects calls to the kernels to REDEFINE HPC). Calls to kernels (in the application running on the host) are offloaded to REDEFINE HPC by the device driver running on the host. Refer figure 1 which gives a block diagram of REDEFINE connected to a Host through RRM.

The CRs required by a kernel to be executed to completion (when launched to be executed on REDEFINE) is specified by the compiler. Note that at any instant of time of execution of a kernel, a CR is exclusively allocated to the kernel, both in terms of compute entities and the portion of DGM that it houses. Further during the compilation process, the compiler identifies a static mapping of HyperOps to CRs. Once a CR completes execution of all HyperOps allocated to it, it signals the RT that the kernel has finished execution. REDEFINE executes HyperOps which are statically mapped to CRs. In the first version, this is the execution model for which REDEFINE Resource Manager (RRM) is designed.

Note: In the case of dynamic mapping, compiler indicates only the number of CRs (and type of CR, in case of heterogenous fabric) needed to execute a kernel. It is the responsibility of RRM to allocate the required number (and type) of CRs to a kernel. In this model, CRs are allowed to ”steal” work from their neighbors.

Handling exceptions:

In case of certain exceptions, the host sends a “Flush” command to REDEFINE. A ”Flush” command can arrive from the host prior to, during or after execution of a
kernel. Note that a "Flush command takes the highest priority and cannot be ignored by REDEFINE under any circumstance. Upon receiving this command, REDEFINE stops execution of kernel(s), if any. In order to handle certain exceptions during execution of a kernel, REDEFINE executes a “Flush” command. The effect of this is same when this command is received from the host.

**Single-phase vs. 3-phase execution:**

We consider two cases of launching and executing kernels. In the first case, the host loads meta data of a kernel to REDEFINE, only after all the inputs needed for the kernel are ready, i.e., kernel and its inputs are loaded into REDEFINE in a single-phase. While in the second case, a kernel is launched by the host even before it receives some/all of its inputs. Inputs to the kernel, as and when they are available (at the host) are loaded to REDEFINE by the host. This way of launching a kernel (in a 3-phase manner) has the advantage that a kernel can start executing (on REDEFINE) as soon as inputs are made available to it. In this exposition we design and implement the former. Further we restrict the first version of implementation to non-streaming kernels. (ie, once a kernel
completes execution on a set of input data, the CRs allocated to it are free to be loaded with a different kernel.)

2.1 Modules of RRM: Resource Tracker (RT) and Memory Address Translator (MAT)

The REDEFINE Resource Manager (RRM) comprises of a Resource Tracker (RT) module and a Memory Address Translator (MAT) module. These two modules perform all activities necessary to load and run kernels on REDEFINE and receive outputs produced due to execution of kernels. This section provides a brief description of RT and MAT.

2.1.1 RT: Resource Tracker

As mentioned before, a kernel is launched for execution on REDEFINE only when the required resources are available for execution. RT receives request (from device driver running on the Host) for launching a kernel, along with the CRs that are necessary for the kernel to execute. Based on the availability (non-availability) of requested CRs, RT responds with a positive (negative) acknowledgment to the device driver. It is the responsibility of RT to maintain information regarding status of resources of REDEFINE prior to, during and after execution of kernels.

Note: The following text needs to be moved to implementation section: In order to maintain information about the availability of CR’s on REDEFINE, the RT maintains two structures - a CR list and a Kernel Table. The CR list contains information of the state of each CR (either “free” or “busy”). Kernel Table is used to keep track of the number of CR’s assigned to each kernel. This table contains information regarding which CR’s are assigned to a particular kernel, the state of a kernel (will be discussed in subsequent sections) and other information (like whether the kernel is streaming or non-streaming). A more detailed description of the structures will be provided in the implementation section.

For ease of implementation, it is the RT module that communicates with the host (and vice-versa).
2.1.2 Memory Address Translator

This module is responsible for sending code and data to the appropriate memory locations of REDEFINE. It is the MAT unit that handles the programming phase of a kernel. Host processor (upon receiving positive acknowledgment for a request for launching a kernel) sends instruction and data pages to RT. RT forwards the same to MAT. Essentially, this unit performs the task of translation from the address space of the host processor to the address space of REDEFINE. MAT has access to the information regarding CRs maintained by RT. MAT does the address translation and packages instruction and data pages with appropriate headers, so that they are loaded into appropriate DGMs of CRs.

MAT is aware of the Start-HyperOp and End-HyperOp of the kernel and hence their locations in the DGM. Initial data (for a kernel) is placed in an appropriate address; i.e., in the context frame of Start-HyperOp by the MAT. Thus execution of the kernel is kick-started. CEs start executing HyperOps of the kernel. Each CR that completes execution of HyperOps allocated to it, sends “idle” signal to RT. RT in turn updates the status of CR. “Idle” signal received from the CR which is allocated End-HyperOp indicates end of execution of the kernel. This also indicates that the output of kernel is available. MAT does the “reverse” address translation and sends the output to RT, which in turn sends it to the host.

Note: In order to do the address translation, MAT maintains an internal table that has the base address for each DGM on REDEFINE.

For ease of implementation, it is MAT module that communicates with the REDEFINE execution fabric.

2.2 Interfaces

The following section describes interfaces that exist between the device driver and RRM and between the two modules RT and MAT.

2.2.1 Signature of the Call Function

In the application code (running on the host), any call to REDEFINE is translated as a call to a user program (having similar functionalities as a device driver) running on the host processor. Kernels that need to be accelerated on REDEFINE have a function
signature that passes the necessary information to the device driver-like user program. The function signature, which we will call CALL_REDEFINE(...), should provide the following information to the user program - ‘Address of an array that contains the identity of CRs needed by a kernel’, ‘Pointer to the first instruction page (on the Host Processor’s Address space)’, ‘Count of the number of Instruction Pages’, ‘Pointer to the first data page (on the Host Processor’s Address space)’, and the ‘Count of the number of Data Pages’

2.2.2 Sequence of Execution

Below we list the sequence of steps during execution of kernels on REDEFINE.

1. During execution of an application on the host, a call to the device driver is made when a kernel has to be executed on REDEFINE. The device driver sends the request to RRM by sending the kernel id and CRs required for execution of the kernel.
   Note: In this implementation, we use a general purpose host connected to REDEFINE, emulated on a FPGA device, by PCIe. In this particular implementation (on proFPGA boards), transfers on PCIe are effected through calls to MMI 64.

2. RT receives the request and evaluates whether the kernel can be executed on REDEFINE or not. A negative acknowledgement is sent from RT to host in case the requested kernel cannot be launched on REDEFINE.

3. Based on the response received from REDEFINE, device driver (running on the host) takes appropriate action.

4. In case of a positive acknowledgement signal, instruction and data pages are transferred form host to REDEFINE (using MMI 64 calls, in this particular implementation). In case of negative acknowledgement, driver awaits a positive acknowledgment from RRM, while the host continues to do other operations. Upon receiving positive acknowledgment, the host is interrupted and the driver starts the programming phase. Device driver blocks further requests, in case there is a call to a another kernel (from the application running on the host) when REDEFINE is getting programmed (with the 1st kernel).

5. Input data for a kernel (that is already loaded on REDEFINE) is sent by the device
driver to REDEFINE in a way similar to loading of instructions (of the kernel). While the kernel is under execution (on REDEFINE), host can continue execution of the application. In addition, host can programme other kernels on REDEFINE, if any, while one kernel is getting executed (on REDEFINE).

6. During programming, while nothing can interrupt REDEFINE, a “flush” command can.

7. In order to resolve contention, priority order of signals (in decreasing order) are:
   Output of a kernel from REDEFINE, Programming Kernel, Request for a Kernel.

8. Upon completion of execution of a kernel, RRM sends a signal to the driver, so that the driver accepts the output data (identified by the corresponding kernel ID). It is upto the device driver to further process the output (or leave it as is).

2.2.3 Format of Requests

Note: The following can be pushed to the implementation section:

   **Start Mechanism:** A message is identified as signal if it starts with a string of 1’s i.e.- 8'b1111 1111. All other messages are not considered as signals (could either be data or garbage values) and appropriate actions are taken based on previous signals.

   Message Header (for the signals):

   - Request for kernel (from Driver) - 8'b0000 0001
   - Positive Acknowledgment (from RRM) - 8'b0000 0010
   - Negative Acknowledgment (from RRM) - 8'b0000 0100
   - Completion of Programming Phase (from Driver) - 8'b0000 1000
   - Completion of Execution (from RRM) - 8'b0001 0000
   - Completion of Outputs (from RRM) - 8'b0010 0000
   - Flush (from Host) - 8'b0100 0000
   - Flush (from RRM) - 8'b1000 0000
If a message header is none of those defined above, the message is not considered as a signal. It is treated as data to be stored in the DGM. End request is signaled by a string given as 8’b1111 1110 (1’s complement of the message header).
3 Resource Tracker - A Detailed Description

3.1 Introduction

The Resource Tracker (RT) needs to maintain state information of every kernel in REDEFINE and manage the transitions in between the states. To do this two data structures can be maintained – a list of the state of each CR on fabric (called CR STATE TABLE) and a table maintaining information of the CR’s comprising a kernel (called KERNEL TABLE).

CR STATE TABLE – This can be implemented as an array which is indexed by the CR ID. The value (1/0) of the array at CR Id x tells us whether CR no. x is free (1) or not (0).

KERNEL TABLE – This table can maintain information on the CR’s that comprise a kernel and the state of the CR’s inside a kernel. This table will also hold other information like whether the kernel is of streaming type or not

The Resource Tracker (RT) can be conceptualized as composed of the front end and the back end. The front end takes in the request delivered by the device driver and decides whether the request can be serviced based on the values in the CR STATE TABLE (In the preliminary model, the compiler provides the list of CR’s that are in a region executing a kernel). The back end maintains information of which CR’s are in each kernel and the state of every kernel. The back end makes updates to the KERNEL TABLE based on signals sent by the MAT.

3.2 Legend for this section

The following abbreviations will be used throughout this document.

RTP : Ready-to-program state
RTE : Ready-to-execute state
EXEC : Execute state
RT : Resource Tracker
MAT : Memory Address Translator
3.3 State transition diagram

The RT is a Finite State Machine (FSM) that performs journaling activities to maintain an accurate information of each CR on REDEFINE fabric at all times. Two cases will be discussed in this section. The first case depicts the state diagram for the case that REDEFINE executes only one kernel at a time. The second case deals with execution of more than one kernel simultaneously on the accelerator. Our preliminary model follows the second case. The first model has been introduced in the document to make the reader familiar with the basic operations that are required for book-keeping.

3.4 CASE I: One kernel at a time

The state transition diagram for a CR in this scenario is shown in Fig. 2. In the beginning, all the Compute Resources (CRs) of REDEFINE are in the Free state. When a request for a kernel is delivered to the RT, the RT checks whether the the CRs requested by the kernel are available or not. In the case of a single kernel executing on REDEFINE at a time the check logic is simple - we just need to ensure that the number of resources needed by the kernel are not more than the total number of CRs available on fabric. Once it has been decided that a request can be serviced, the necessary code and data is loaded into the DGMs of the CRs and the CRs are then sent to the Ready-To-Execute (RTE) state. In the RTE state, the CRs wait for the arrival of the inputs to the kernel to begin execution. Once the inputs have been placed in the context frame of the Start-Hyperop, the CRs can begin execution and they are sent to the Excution (EXEC) state. After execution, the behavior of the CRs is varied depending on whether the kernel was a streaming or non-streaming application. In the case of Non-Streaming applications, the CRs are simply sent back to the Free state once the execution of the kernel is finished. In the streaming case, the CRs of a kernel continue to remain in the EXEC state. Such kernels repeatedly perform the same operations repeatedly on different input sets. The transition of streaming kernels from the EXEC state to the Free state is initiated by a TERMINATE_KERNEL command issued by the Host Processor.

The front end of the RT for the state diagram in Fig. 2 has been depicted in Fig. 3. The front end takes in requests (from the compiler) delivered by the device driver and enqueues them in a request queue. The front checks the CR STATE TABLE to decide
whether it can service the request and sends back the relevant information to a user program running on the host (similar to a device driver for REDEFINE).

![State diagram for a kernel](Image)

**Figure 2: State diagram for a kernel**

The back end of the RT maintains a KERNEL TABLE and makes updates to the FREE LIST whenever necessary (to free idle CR's). The KERNEL TABLE can be realized as a lookup table that holds state information of every CR. Figure 4 depicts the KERNEL TABLE.

The state diagram for the Back end of the RT is shown in Fig. 5. In Figure 4, the state transitions happen whenever the device driver provides the RT with a signal. The `PROGRAMMING_START(Kernel Id)`, `EXEC_START (Kernel Id)` and `EXEC_OVER(Kernel Id)` signals are provided by the DMU. The `TABLE_UPDATE` signal is issued by the front end of the RT and not the device driver. The `PROGRAMMING_START` signal given to the RT by the DMU when it starts address translations. The `EXEC_START(Kernel Id)` signal is sent when the DMU writes the input data into the context frame of the Start-Hyperop and the `EXEC_OVER(Kernel Id)` signal is given out when the result is produced by the End-Hyperop. The case of Flush commands is also considered in Fig. 5.
Figure 3: front end of RT

Figure 4: kernel table
An assumption made regarding the back end is that in case of a conflict between a read of the free list array and a write to it, the write will precede the read. This is because the write operation basically frees up resources (adds CR’s to the free list) and this will ensure that requests are not rejected unnecessarily.

### 3.5 CASE II: Multiple Kernels at the same time

The state transition diagram for this case will be more involved than that of Case I. The diagram is depicted in Fig. [6].

The above state diagram is slightly different from the state diagram for Case I. The addition of the extra state, called Partially Programmed (PP) State, allows for partial programming of kernels, i.e., the programming of kernels can begin even when all the CRs requested by a kernel are not free at the moment the request was received. This state also enables simultaneous programming of kernels while other kernels are either executing or waiting for inputs to start execution. In the PP state, there is a loop that continuously checks whether all the CRs needed by a kernel are available or not. As soon as the requirement of a kernel is completely satisfied the kernel is sent to the Completely
Programmed (CP) state. This state is similar to the Ready-To-Execute state of Case I. The behaviour of streaming kernels is same as that depicted in Case I. In case of Non-streaming application, the behaviour is slightly different as depicted in Figure 6.

The state transition diagram for the front end of the Resource Tracker (RT), shown in Figure 7 is almost similar to the one depicted in Figure 3. The only difference lies in the mechanism for checking of resource availability. In Case I, the approach was to check whether the number of resources claimed by a kernel is less than the total compute capacity offered by REDEFINE. In Case II, since multiple kernels can execute at the same time, the approach is to program kernels even if a single CR requested by the kernel is in the Free state. This mechanism allows is similar to a pipelined implementation of programming kernels. The programming of kernels need not wait until all the CRs needed by the kernel are in the free state.

The state transition diagram for the back end of the Resource Tracker (RT) is shown in Figure 8. The diagram has been modified from its equivalent diagram for Case I to accommodate for the extra Partially Programmed (PP) State.
Figure 7: front end for RT for Case II

Figure 8: back end of RT for Case II
4 Implementation Details

4.1 Anatomy of Signals

The communication between the host and REDEFINE HPC is achieved using MMI64 protocol [4]. In this protocol, all data that is exchanged between the host processor and REDEFINE is broken down into packets of 32 bits. Thus, any signal is a stream of such 32 bit packets. In the subsequent sections we would be discussing signals which includes requests, programming and outputs.

Requests

In the Interfaces section, we had defined the signature of the Call function. Among other information the call to redefine contained information of the CRs (addressed by their CR Id) needed to execute a kernel on REDEFINE. A request to the Resource tracker consists of this CR related information. The request is sent from a user program running on the host (similar to a device driver for REDEFINE) to the Resource Tracker (RT) module of the RRM. The request is sent in the form of a linear array of size $n$ (where $n$ is equal to the number of CRs on the REDEFINE fabric). The array is indexed by the CR Id and the value of the CR states whether a particular CR is needed for a kernel’s execution.

To identify any data as a signal we will be following a convention. Any data exchanged between the host and REDEFINE must have a particular bit pattern in the first packet of the signal. The first eight bits of the first 32 bit packet of any signal must be $8'b11111111$. The presence of this string is checked to identify data as signal (this mechanism has been adopted to handle cases of garbage values being exchanged between the host and REDEFINE).

For a request, the next eight bits of the first packet must be $8'b00000001$. After this first packet, the next $N$ 32 bit packets will contain information on the CRs needed for executing a kernel (where $N \equiv (NumberOfCRs) \mod 32$). The next 32-bit packet will be the end of the request signal. The first eight bits of this packet would be $8'b11111110$ (one’s complement of $8'b00000001$).

The $N$ packets are accumulated in a linear array and this is the request that reaches the Resource Tracker (RT). Based on the availability of resources an appropriate acknowledgement signal is sent back to the Host processor (the format of both positive and
negative acknowledgement has been mentioned in the Interfaces section).

Programming Phase

If the outcome after processing a request is a positive acknowledgement then the pro-
gramming phase begins. In this phase, all the instructions and data for a particular kernel
are transferred from the Host processor’s address space to that of REDEFINE’s.

As in any signal, the first eight bits of the first 32-bit packet of programming data
would be $8'b1111111$. For the signal to be programming related data the next eight
bits need to be $8'b00000100$. The last 16 bits indicate the number of CRs required by
the kernel that has to be programmed on REDEFINE. The programming phase is a
big operation that cannot be pre-empted. Only a Flush command (not included in the
preliminary model) can interrupt a programming operation.

After the first 32-bit packet, the next $2 \times N$ 32-bit packets will provide information
specific to each CR required in the kernel (where $N =$ number of CRs required by the
kernel). For every CR in a kernel two 32-bit packets are needed to convey information
about the CR Id, number of 32-bit chunks of instructions, number of 32-bit chunks of data
and number of 32-bit chunks of context frame. In the first 32-bit packet for a CR, the
first 8 bits are reserved for CR Id and the next 24 bits are for the number of data chunks.
In the next 32-bit packet, the first 16 bits are reserved for the number of instruction
chunks and the last 14 bits convey information about the number of chunks for Context
Frames. This per CR information is used while populating the necessary DGMs later in
the programming phase.

After collecting information about every CR in a kernel, the actual transfer of in-
structions, data and context frames begins. The programming of kernel occurs in a
sequential manner. All the contents for a particular DGM are populated before moving
on to the next DGM. The per CR information described in the previous paragraph is
used to maintain an exact count of the number of 32 bit packets needed to populate a
particular DGM. This process is repeated for the number of CRs in a kernel. At the end
of the programming phase the next 32-bit packet will signify the end of the programming
operation (this is introduced only as a checking mechanism. The count is the major tool
used for ensuring correct programming of a kernel).
The preliminary model allows execution of one kernel and servicing requests/programming of another kernel to take place concurrently. As a result, it is important that we constantly keep checking whether a particular kernel has finished execution. In the current model, the CR that executes the End-HyperOp of a kernel notifies the MAT module of REDEFINE Resource Manager (RRM). The MAT is notified in the following manner. First, the CR executing the End-HyperOp writes the output data to a FIFO that is periodically checked by the MAT unit. If there is data in this FIFO then the MAT unit reads the output from the kernel through an access router. This data is then transferred to a temporary store by the MAT. The RT then reads the output data from this temporary store and sends it across to the Host processor.

In order to handle cases wherein multiple kernels might be producing output data concurrently, we have a two dimensional array with as many rows as the number of CRs and a column size which would be bigger than any output produced by REDEFINE. This is the temporary store that was referred to in the previous paragraph.

### 4.2 Jounaling activities of the Resource Tracker

The primary function of the RT is to maintain an accurate description of the states of all the CRs on the REDEFINE fabric. To this end, the RT maintains two structures - CR state table and Kernel table - which are modified after every major operation. In this section, we would be discussing the importance of each structure.

The CR state table is maintained for the purpose of deciding whether requests can be processed on REDEFINE or not. This structure is manipulated whenever a kernel can be programmed on REDEFINE or when a kernel finishes execution.

The kernel table maintains information regarding which CRs comprise a kernel, the state of a kernel and the type of kernel (Non-streaming for the preliminary model). The kernel table is modified after the following three events:

1. A request for a kernel has been positively acknowledged
2. The programming phase of kernel completed
3. Kernel completed execution
4.3 REDEFINE Emulation on FPGA

In this section we give implementation details of REDEFINE HPC. REDEFINE HPC is emulated on a Xilinx Virtex-7 FPGA board (of proFPGA). All the components are built on the FPGA. Host is emulated on a general purpose processor (GPP) running Linux. Host is connected to REDEFINE (emulated on the FPGA board) via PCIe. Communication on PCIe is realized by MMI 64 calls provided by the vendor. Figure 9 give a high level diagram of the emulation.

A MicroBlaze is instantiated on the FPGA device and Linux kernel is installed on this. RT and MAT, which are the two modules of RRM are realized as pthreads running on Linux kernel. RT talks over the PCIe (using MMI 64 calls) with the host. And the MAT talks to the REDEFINE execution fabric. To recap, execution fabric of REDEFINE consists of a number of compute resources interconnected by a NoC in a mesh-toroidal topology. It is front ended by the Redefine Resource Manager or the RRM. Each Compute Resource comprises a Transporter, an Orchestrator, Compute Element(s) and Distributed

Figure 9: Emulation of Host and RRM
Global Memory or DGM. MAT and execution fabric talk to each other using special routers (called access routers).

As many MicroBlazes as the number of CRs are instantiated and Linux kernel is installed on each of them. Functionalities of the Transporter and Orchestrator are realized as pthreads (written in C language). A CE (within a CR) is realized as a Rocket-Chip (an instance of RISC-V processor. The part of DGM within each CR is realized as a Block RAM. Refer to figure 10 for details of emulation of CR.

Figure 10: Emulation of Router and CR

Microblaze (running Transporter and Orchestrator on the Linux kernel) is connected to the Rocket-Chip and the DGM through AXI. The Rocket-Chip and the BRAM are also connected to each other by AXI. Microblaze is connected to its Access Router via two FIFO structures, say FIFO1 and FIFO2. These FIFOs are n-bit wide (n = size of packet) and 1 row deep. Access Router puts packets in FIFO1, while it gets packets from FIFO2. Transporter can read or write in the FIFOs and do basic FIFO functions.

**Kavitha, you need to review the following:**

**Transporter and Orchestrator**: Transporter and Orchestrators are implemented
as threads running on linux kernel built of MicroBlaze. Both threads start to run when the CR powers on or when runtime of the Host Application begins and run continuously from then on. The Transporter thread's primary job is to check the status of FIFO1, read from it the payload when it is full, decode the address and data from it and then store the data in the address. It is also responsible for putting the output packet in FIFO2. The Transporter thread runs, checking for data in FIFO1. This does not involve reading the data but just checking if there is some valid data (say, by using isEmpty() function). The RRM sends information in form of packets to Access Router, the Access Router in turn puts them in FIFO1. A packet comprises an Access Router relative address, a VC Number and a Payload. The Payload consists of an absolute address of the BRAM but in the range of the associated DGM, and the information which is to be stored in that address. When a packet is put in FIFO1, the Transporter thread reads it, thus resetting the FIFO. It then goes back to checking for the status of the FIFO. At the same time, it decodes the address and stores the information (instructions, data or context frames) in that location.

The Orchestrator thread is responsible for Context Memory and Ready Queue management. The Ready Queue is a queue of HyperOps which have all the inputs that they require to begin execution. To know whether a HyperOp is ready or not, the Orchestrator has to check its context frames. Each context frame comprises of four fields, viz. the HyperOp it belongs to, a required bit, a valid bit and data (or a pointer to data in the DGM). It keeps checking the allocated context frames if all the data which is required by a particular HyperOp is valid. This doesn't necessarily have to be done all the time, but only when the Transporter stores some data to context memory. When the Orchestrator finds all required data to be valid, it puts the HyperOp in the Ready Queue. The input data which is required by a HyperOp has to be mapped to the appropriate registers of the Rocket-Chip's input register set. This mapping can be done explicitly by Load instructions which are added by the compiler statically. However a direct mapping approach can also be used wherein all the context frames of a HyperOp have a pre-existing map to the registers of the Rocket-Chip. After this the Orchestrator commands the execution of the instructions. The output is then packed by the Orchestrator along with the address using the Access Router look-up table. Transporter places the packets in FIFO2. The Access Router gets these packets and sends them to their destination through the NOC.
5 RESULTS

As mentioned in the introduction, we aimed to design a hardware system that would emulate REDEFINE replete with the RRM acting as a frontend to REDEFINE HPC. We also developed a software simulator to test the functional accuracy of our Resource Manager. This section provides details after running the Software simulator. The design of the Hardware Emulation on FPGA is in the final stages and we will be updating the section soon thereafter.

As far as the software simulation is concerned we wrote C++ codes of the Resource Tracker, Memeory Address Translator, Orchestrator and Transporter. We performed unit tests on each of the above mentioned modules and successfully verified the functional correctness of each unit. Finally, we clubbed each of the above mentioned modules in a software simulator that we developed. The Software simulator integrated the individual modules designed by us with a simulator for the Network-On-Chip (NOC). The aim of this experiment was to provide a proof of concept for our proposed design.

To that end, we tested our simulator with various test cases and verified that the hardware accelerator performs the computations correctly and returns the results back to the Host processor properly. We tried many test cases to check for various conditions. My focus area was on the Resource Tracker and to a small extent the Memory Address Translator (MAT). Some important test cases which checked the functional accuracy of the RT were:

- **Sending requests for the same CRs one after the other**: This test case verified that the RT does not allow loading of a kernel when insufficient resources are available.

- **Loading kernels that finish execution almost simultaneously**: This was a test case primarily designed for the MAT unit. We performed this test to check whether the RRM can handle receiving outputs from two kernel simultaneously.

- **Loading large kernels to REDEFINE**: This test was designed to check whether the context frames of kernels were populated properly. The kernel produces the desired result after computation proving that inter-HyperOp (finest granularity of processing instructions in REDEFINE), which uses Context Frames, worked correctly.
The above test cases worked as desired proving that the Resource Manager was able to successfully manage the execution of computation kernels on REDEFINE HPC.
6 CONCLUSIONS

The aim of the thesis was to develop the case for a resource manager for REDEFINE HPC. The importance of such as resource manager has been highlighted throughout the document. The results obtained from the software simulator show that the resource manager is functionally correct. The results obtained from the Hardware Emulation will combine the benefits of having dedicated resource manager with that of a Reconfigurable Coprocessor. We expect to witness vast speedups in certain applications as compared to when they execute only on a General purpose host.

This work can be extended into the future by designing the modules in a HDL language like Verilog. We can have a more robust resource manager that can provide a wide range of acceleration options - Increased parallelism, reduces energy consumption, etc.
Appendix

Appendix A: Hardware Design using Vivado 2014.4

In this section we will provide details related to designing hardware systems using Vivado 2014.4, a tool developed by Xilinx. The objective of this section is to list the steps involved in the design a MicroBlaze processor which would be capable of booting Linux [1]. The MicroBlaze, thus produced, would also be capable of running bare metal applications (code running directly on the processor in the absence of any Operating System) that have been cross compiled for the microblaze ISA.

There is a list of design constraints that need to be honoured for being able to boot Linux on a MicroBlaze processor. The requirements for Linux to boot on MicroBlaze are:

- External memory controller with at least 32MB of memory
- Dual channel timer with interrupt connected
- UART with interrupt connected

The MicroBlaze that we would be making in Vivado must contain the above mentioned peripherals. The Vivado Design Suite Tutorial[?] provides a detailed description of the steps involved in making designs using Vivado.

During configuration of the MicroBlaze processor, the template for Linux with MMU must be selected. In the subsequent pages the number of protected regions must be at least two. The rest of the steps are as mentioned in the document listed above. The document specifies the procedure to add GPIOs to the design and interface them to the MicroBlaze processor. The number and size of GPIOs can be modified as per the needs of the projects. There is a limit to the number of slaves that can be attached to the MicroBlaze processor using one AXI peripheral interconnect. If the number of GPIOs or other peripherals increases more than this limit then we need to add one more AXI peripheral interconnect. The second AXI peripheral interconnect needs to be connected as a slave to the primary AXI peripheral interconnect. All extra peripherals can be connected to this second interconnect. The clock input to MicroBlaze comes from the MMI64 clock.
After checking the design for validity, we will have an .hdf file (hardware description file) as an output. This file will be used by the Petalinux tools to build a linux image for MicroBlaze. Before we move our design to synthesis and implementation, we need to ensure that the First Stage Bootloader (FSBL) is initialized to the BRAM memory attached to MicroBlaze. The fs-boot.elf file will only be generated after the Petalinux process has been completed. This is achieved by going to Tools - Associate elf and adding the correct elf file. After this step we can click the Generate Bitstream option to get a bitstream with fs-boot initialized to the BRAM.

Appendix B: Building Linux using Petalinux 2014.4

In this section, we delineate the steps involved in building a linux image using Petalinux tools (made by Xilinx). The input to this tool would be an hdf file that was generated using Vivado. Finally, we will get a linux image (named image.elf) complete with a device tree file (.dts).

The Reference Guide by Xilinx contains the steps involved in building linux for MicroBlaze [3]. There are no modifications required to the design flow provided in the document. After the build process, we will get three files - fs-boot.elf, u-boot.elf and image.elf. These files need to downloaded to the FPGA using petalinux commands. A JTAG cable can be used for this purpose.
References


