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

2016

MSc THESIS

Exploiting the Reconfigurability of ρ-VEX Processor for Real-Time Robotic Applications

Muhammad Muneeb Yousaf

Abstract

CE-MS-2016-10

Autonomous mobile robots generally have limited computational power on-board, and they have to perform their tasks in real-time in order to interact with their surroundings effectively. Therefore, there is a need to utilize the available computational capabilities efficiently. The ρ-VEX is a run-time reconfigurable VLIW processor. This unique processor allows separation of its issue lanes to form independently operating processing cores. Switching between these configurations during run-time allows optimizing the computing resources for the task(s) it is performing.
In this project FreeRTOS is ported to the ρ-VEX processor and a control layer is developed. FreeRTOS manages the applications based on given real time parameters. The control layer decides the number of active cores (hardware contexts) and issue width of each core to best match the processing requirements of the applications. In this way, FreeRTOS and the control layer together can reconfigure the number of active cores at run-time. This is a very unique feature of this thesis project and can not be found in any other multicore implementation of FreeRTOS. The control layer along with FreeRTOS provides the user a facility to run applications under real-time constraints and with the best possible efficiency.
In order to evaluate the performance, the overhead of the FreeRTOS is quantified and a performance comparison is made between several configurations of this system. Moreover, impact of reconfigurability of the ρ-VEX on the schedulability of real-time applications is measured and it is concluded that (indeed) reconfigurability of ρ-VEX improves the schedulability of the real-time applications.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Exploiting the Reconfigurability of ρ-VEX Processor for Real-Time Robotic Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Muhammad Muneeb Yousaf
born in Vehari, Pakistan

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

Exploiting the Reconfigurability of ρ-VEX Processor for Real-Time Robotic Applications

by Muhammad Muneeb Yousaf

Abstract

Autonomous mobile robots generally have limited computational power on-board, and they have to perform their tasks in real-time in order to interact with their surroundings effectively. Therefore, there is a need to utilize the available computational capabilities efficiently. The ρ-VEX is a run-time reconfigurable VLIW processor. This unique processor allows separation of its issue lanes to form independently operating processing cores. Switching between these configurations during run-time allows optimizing the computing resources for the task(s) it is performing.

In this project FreeRTOS is ported to the ρ-VEX processor and a control layer is developed. FreeRTOS manages the applications based on given real time parameters. The control layer decides the number of active cores (hardware contexts) and issue width of each core to best match the processing requirements of the applications. In this way, FreeRTOS and the control layer together can reconfigure the number of active cores at run-time. This is a very unique feature of this thesis project and can not be found in any other multicore implementation of FreeRTOS. The control layer along with FreeRTOS provides the user a facility to run applications under real-time constraints and with the best possible efficiency.

In order to evaluate the performance, the overhead of the FreeRTOS is quantified and a performance comparison is made between several configurations of this system. Moreover, impact of reconfigurability of the ρ-VEX on the schedulability of real-time applications is measured and it is concluded that (indeed) reconfigurability of ρ-VEX improves the schedulability of the real-time applications.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2016-10

Committee Members :

Advisor:	dr.ir. Stephan Wong, CE, TU Delft
Chairperson:	dr.ir. Stephan Wong, CE, TU Delft
Member:	dr.ir.A.J.van Genderen, CE, TU Delft
Member:	dr.ir.Chris Verhoeven, ELCA, TU Delft

Dedicated to the memory of my mother

Contents

List of Figures

List of Tables

List of Acronyms

TCB	Task Control Block
CE	Computer Engineering
OS	Operating System
RSC	Requested Software Context
CSC	Current Software Context
SOC	System on Chip
RTOS	Real Time Operating System
RMA	Rate Monotonic Algorithm
EDF	Earlier Deadline First
EEMCS	Electrical Engineering, Mathematics and Computer Science
GRLIB	Gaisler Research Library
VLIW	Very Long Instruction Words
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
CCR	Context Control Register
CHC	Current Hardware Configuration
ISR	Interrupt Service Routine
WCET	Worst Case Execution Time
API	Application Programming Interface
LCM	Least Common Multiple
IPC	Instructions per Cycles
CRR	Context Reconfiguration Request
SCCR	Saved Context Control Register
SAWC	Sleep And Wake-up Control
WCFG	Wake-up Configuration
BCRR	Bus Reconfiguration Request

DSP Digital Signal Processing

Acknowledgements

There are a number of people who helped and supported me in the course of this project. First of all, I would like to thank my supervisor, Stephan Wong, for granting me the opportunity to work on this project. During the course of this project, he helped me to maintain a high level view of the project. Secondly, I want to thank Joost for guiding me along the way, supporting with the low level details of this project and proofreading my entire thesis report. I really appreciate his guidance and energetic support.

I also want to express my thanks to all my lab mates and class mates in EWI, especially my *Band of Brothers (Haji, Koray and Chris)*, who have been a lot of fun to hangout. I cherish every moment that I spent with these guys during my stay in Delft. They were my family and friends away from home. I want to say thank you for everything.

My family has been a constant support for my entire life. My parents and my siblings have given me the love and support necessary for me to reach my goals. I treasure their support.

Muhammad Muneeb Yousaf
Delft, The Netherlands
August 15, 2016

Introduction 1

This thesis describes a MSc project that aims to evaluate the impact of the ρ-VEX reconfigurability on the performance of real-time robotic applications. This chapter first provides context and motivation behind this project, and then the problem statement is introduced. Subsequently, project goals and methodology to achieve those goals is discussed. In the last part of the chapter, the structure of this thesis report is described.

1.1 Context

Liquid architectures is one of the main research topics of the Computer Engineering (CE) group of TU Delft. The ρ-VEX processor [4] is one of the active projects in this direction. It was initially designed to be used as a co-processor in heterogeneous processing platforms such as the MOLEN machine [7]. However, after years of development and extension in the ρ-VEX, it has now become an experimental processing platform. This thesis project is carried out in the context of this larger ρ-VEX project. In this project FreeRTOS is ported to the ρ-VEX processor and a control layer is developed. FreeRTOS manages the applications based on given real time parameters. The control layer decides the number of active hardware contexts and issue width of each hardware context to best match the processing requirements of the applications. In this way, FreeRTOS and control layer together can reconfigure the number of active hardware contexts (threads) at run time. This is a very unique feature of this thesis project and can not be found in any other multi-core implementation of FreeRTOS. The control layer along with Real Time Operating System (RTOS) provides the user a facility to run applications under real time constraints and with the best possible efficiency.

1.2 Motivation and Problem Statement

In order to fully understand and exploit the capabilities of the ρ-VEX, there is a need to understand the impact of its unique features on different conditions (high Instruction Level Parallelism (ILP) and low ILP) of workload. Some robotic applications like arm movement can be very complex and have dynamic computational workload. In the arm movement, the motion planning is a crucial step. In motion planning, some time there is a need to move the arm from one point to the other in 2D space. This scenario is very simple and involves very little computations. But for some complex movements, there is need to move the arm in 3D space and it might also require avoiding obstacles in its path. All these things make this scenario of motion planning very complex and it may take minutes to compute the exact arm movement. Most of the time, motion planning is also required to be done in real time[10], so that robot can respond to its surroundings effectively. Therefore, if resources are allocated according to worst case execution requirements, then resources will be wasted if there is not much to compute. Logically, it seems beneficial that if resources can be acquired when there is high computation demand and if there is not much to process then extra resources should be released, so that other applications can use them. In such kind scenario ρ-VEX fits very well, as its processing resources can be changed according to the requirements of the applications at the run time. In order to efficiently and conveniently execute different applications on the the ρ-VEX in real time, there is a need of RTOS, and to configure the processor according to the requirements of the applications, a control layer is required. So the problem statement of this thesis is:

How to design and implement RTOS and control layer for the ρ-VEX in order to (best) meet the processing requirements of the real-time applications ?

This is very high level description of problem. In order to tackle this problem within the given time frame, there are some goals that are defined. The description of these goals and the method followed to achieve them is discussed in the Section 1.3.

1.3 Methodology and Project Goals

In order to find a good solution of the above problem, it is first necessary to investigate what solutions already exist and which are suitable for the ρ-VEX platform. Plethora of RTOS is already out there, therefore, instead of developing the RTOS from scratch, it is decided to port already existing solution for ρ-VEX. After a thorough investigation, FreeRTOS has been selected for porting on the ρ-VEX. The reasons for selection of this RTOS are following:

- Complete source code is available online.
- Source code is properly documented.
- It has already been ported for a number of other platforms, So performance of this RTOS on ρ-VEX can be compared with others without much effort.

As the ρ-VEX is an experimental processor, and it is also run time reconfigurable, therefore, the development of the control layer and porting of FreeRTOS exhibits unique challenges that can not be faced in any other platform. This leads to first goal of this project:

1. Design and implementation of the control layer, and porting of FreeRTOS for ρ-VEX.
   To reach this goal, following approach is taken:
   - Identify how the ρ-VEXs unique features impact the porting of FreeRTOS.
   - Selection of suitable scheduler for FreeRTOS.
   - Identify how to interface control layer and FreeRTOS, so that both can work seamlessly with each other

- Design and implementation of the control layer.
- Integration of FreeRTOS and the control layer.

After the porting of FreeRTOS, and its integration with the control layer, there is need to test that both work as intended. This objective leads to the second goal of this thesis project:

2. Making sure that the modified FreeRTOS and the control layer work as intended.

   In order to achieve this goal, the following steps are taken.

   - Test the functionality of the scheduler in the modified FreeRTOS.
   - Test the behavior of the control layer.
   - Test that the control layer reconfigures the hardware based upon the requirements of the tasks.

   This thesis project is carried out to exploit the reconfigurability of the ρ-VEX in order to improve the performance of real-time applications. Every real-time application has an associated deadline (before that it should produce the results). The fact that all the deadlines are met is one of performance metrics of real-time applications [18] and this metric is also called schedulability. If the reconfigurability of the ρ-VEX improves the performance of the real-time applications then in turn it should improve the schedulability of the these applications. This leads to the next goal of this project:

3. Identify the impact of reconfigurability of the ρ-VEX on the schedulability of the real-time robotic applications.

   The following steps are taken to achieve this goal:

   - Select suitable robotic applications.
   - Measure Worst Case Execution Time (WCET) of the selected applications on different configurations of the ρ-VEX.
   - Calculate utilization of each task for different hardware configurations.
   - Measure the performance and schedulability of the real-time applications on different variants of the $\rho - VEX$ which include homogeneous, heterogeneous and run-time reconfigurable.

1.4 Overview

In the Chapter 2, background topics of this thesis project are discussed and related work is described. Chapter 3, describes concepts very specific to RTOS and FreeRTOs. Moreover, this chapter also describes the functional and non-functional requirements of the ρ-VEX ported version of FreeRTOS. In the chapter 4, all the modifications and implementations are described which are carried out to meet the all the requirements. In the chapter 5, functional testing of FreeRTOS is described and its performance is evaluated. In order to evaluate the performance, the overhead of the FreeRTOS is quantified and impact of reconfigurability of ρ-VEX on the schedulability of real-time applications is measured. Additionally, a performance comparison is made between several configurations of this system. Chapter 6, summarizes this the project, list the main contributions and suggests the future work.

Background 2

In this chapter, introductory topics related to porting an OS to the ρ-VEX are discussed. The goal of this chapter is to introduce and refresh those topics that are necessary in understanding the goals and background of this thesis project.
Field Programmable Gate Arrays (FPGAs) will be introduced in Section 2.1. This technology is used as a platform for rapid prototyping and to get a rough estimation of the performance. Then the VLIW class of processors is discussed to whom the ρ-VEX processor belongs. After that, in Section 2.3 the basics of the ρ-VEX and the components of this processor that relevant for the porting of the OS will be explored. Subsequently, Operating Systems (OS), their classes and the factors that derive those classes will be given. After that, scheduling of real-time tasks on uni-processors and multiprocessors is discussed. This chapter concludes on the discussion of characteristics of real-time embedded control systems as autonomous robots belongs to this class of real-time embedded systems.

2.1 FPGA

An FPGA is an integrated circuit whose functionality can be reconfigured after production or in the field. This technology allows designers to describe the desired behaviour of the chip using a Hardware Description Language (HDL) such as Verilog or VHSIC HDL (VHDL) in a similar manner to software implementation.

FPGAs fill the performance gap between Application-Specific Integrated Circuits (ASICs) and software that is being run on a general purpose processor. ASICs are the fastest solution for computationally intensive applications because they are tailor made for a specific workload and can also utilise a lot of parallelism. But their production is only affordable if they are produced in high volumes. This is because of high cost of manufacturing process. On the contrary, software solutions are cheap and easy to implement. But they are very slow as compared to ASICs due to their inherent sequential nature. FPGA combines the parallel nature of ASICs with the ease of implementation of software. But of course, due to some drawbacks, FPGAs still can not replace software or ASICs. Due to the reconfiguration nature, the FPGA occupies almost 20 times as much area as occupied by functional equivalent ASIC implementation [35]. Moreover, FPGAs are slower than their ASIC counter part by an order of magnitude due to large delays in the reconfigurable routing. In comparison with software solutions, FPGA solutions have better performance but this improved performance comes at the price of more expensive ICs and higher development cost.

FPGAs are often used for prototyping when designing ASICs. Because of high cost of production, multiple design cycles are not possible in silicon. Moreover, FPGAs are also used to accelerate software solutions. In that case (major) portions of the software is run by the general purpose processor but computationally intensive parts of the application are run by the FPGA fabric. Here, as there is a need of just small FPGA, therefore, cost per unit remains within the limit. Moreover, it is also possible to implement an entire processor in FPGA fabric. In this case, processor is called soft core processor. By taking advantage of the reconfigurable logic, soft cores can be modified and tuned to the specific requirements. Moreover, custom instruction can be added even after a system has already been designed.

2.2 VLIW Processors

Processors are employed to process the data. They perform operations on data that are dictated by a stream of instructions. There are basically two ways to speed up this process.

1. By increasing operating frequency

2. By processing more instructions per clock cycle, raising the Instructions per Cycle Instruction (IPC) by means of extracting Instruction Level Parallelism (ILP).

VLIW processors belongs to the second case. Contemporary general purpose Central Processing Units(CPUs) are of the superscalar class. These types of processors try to increase the IPC by dynamically checking data dependencies and executing instructions concurrently and sometimes out of order [27]. This process is referred to as *dynamic instruction scheduling*, which is a complex process and requires extensive hardware support, and also consumes a lot of power. Because of the complexity and high cost of implementing such a system is generally not used for softcores [2].

The other way is to assign the compiler the responsibility to extract ILP from the program (statically, at the compile time). VLIW is a processor that uses compiler techniques to improve performance. This moves the effort from processor to compiler and saves a lot of instruction scheduling overhead and power. But this technique has major draw back that programs needs to be compiled specifically for certain platforms, thus code for VLIW is less portable. As a result of this, VLIW processors are most useful in an embedded system where the software and hardware are usually tailored for target application. VLIW (softcore) processors can achieve high performance for a low price especially in applications that have a high level of parallelism such as Digital Signal Processing (DSP) applications.

2.3 ρ-VEX

In the previous sections concepts of soft cores and VLIW processors have been introduced in general. Now the target platform of this thesis project, ρ-VEX can be discussed. The ρ-VEX stands for reconfigurable VLIW Example(VEX). The ρ-VEX processor is not the first VLIW softcore processor. The other implementations of VLIW as a softcore can be

Table 2.1: Static configuration parameters supported by the ρ-VEX processor

Resource	**Parameters**
General	Issue Width, Number of hardware contexts
Functional units	Number, Type and location of functional units, Support operations
Register File	Register File Size
Interconnect	presence of forwarding logic, memory bandwidth
caches	presence of caches, cache size and cache line size

found in [19] [21] [24] [8]. But all these implementation suffer from one of the following draw backs:

- Either relies on closed source compiler or processor design.
- Tool chain lacks in good support
- Limited parametric customization or extensibility.

In order to overcome all these drawbacks, the ρ-VEX was introduced in [36]. This processor is entirely developed by students and Ph.D candidates at the computer engineering group at the TU Delft. Currently, ρ-VEX is in its third version. It is parametrised processor that can be configured at the design-time and reconfigured at run-time. The parameters of the ρ-VEX that can be configured at design-time are shown in Table 2.1

The aspect that is most innovative about the ρ-VEX processor is its capability to be dynamically reconfigured. This feature is first introduced in [5]. Dynamic reconfiguration involves splitting and merging cores creating VLIW processors with narrower or wider issue widths during run time. In the default setup of the current version of the ρ-VEX, the core is configured as a single wide 8-issue VLIW processor. This core can be split up into a maximum of four 2-issue cores. Each core has its own control and general purpose registers. As there could be maximum four cores, therefore, there is a need of 4 register files. These register files complemented with a set of control registers that determine a program's state, are also called hardware contexts. Therefore, cores can execute completely independent of the each other. The possible configurations are one 8-issue, two 4-issues, four 2-issue, or a combination of two 2-issues and one 4-issue. Some configurations of ρ-VEX to which ρ-VEX can be configured are shown in Figure 2.1.

These different combinations allow the core to adapt to the application processing demands at run-time.

When there is program which has very high ILP and it is required to be solved quickly then the core can be configured as a wide processor, thereby drastically reducing the number of cycles required for program completion. If there is not enough ILP to fill all the issue slots and a core executes a large amount of No Operations (NOPs), then the core can be split into two or more smaller ones. This allows multiple processes to run in parallel, increasing the processor's Task Level Parallelism (TLP). When the decision is made to split the core, then the original process keeps continue its execution on a narrower core while another process is started on the core that has become available. When no processes are available to run on the newly freed core, it can be shutdown.

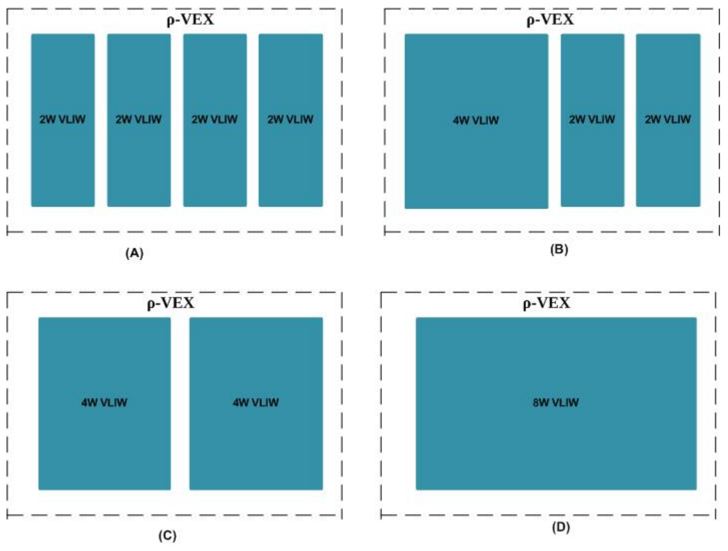


Figure 2.1: ρ-VEX configurations

These run time configurations are only possible because of generic binaries that were presented in [6]. These pieces of executable enable ρ-VEX to be run in any configuration (independent of issue width). Without this technique, multiple versions of programs must be available for different configurations, that must be switched upon a reconfiguration.

As this thesis project is about porting FreeRTOS, and developing a control layer for ρ-VEX, we will discuss only those parts of this processor that are relevant to this goal. These parts of the ρ-VEX include *Program State Storing and Loading*, *Sleep and Wake up* and *Run-Time Reconfiguration*

2.3.1 Program State Storing and Loading

In the ρ-VEX, program state storing and loading is performed by using traps. Upon entering a trap, it is up to the software to save and restore the processor state. Specifically, the software must ensure that the state of the general purpose registers, branch registers and the link register should be same at the Return From Interrupt (RFI) instruction execution as it was at the entry of trap handler. The details of all the registers that are mentioned in the rest of this section can be found in [33]. The hardware handles saving and restoration of the context in Context Control Register (CCR) and the program counter registers, as both of these are modified immediately upon entering the trap handler. CCR is saved in and restored from Saved Context Control Register ($SCCR$), the program counter is saved in and restored from Trap Point (TP) register. Apart from restoring the state of the currently running task, an operating system environment may also wish to restore the state of a different task. In this case, the complete state of a task is defined by the contents of the general purpose register file, the branch register file, the link register, the program counter and the context control register.

There is a small difference in trap generation for the contexts other than context 0. Here, Requested Software Context (RSC) and Current Software Context (CSC) registers are used to generate the trap. These registers do not exist on context 0. The encoding of these registers is at the users discretion, but it is intended that this points to a memory region that contains the task context to be loaded. When the *RSC* value does not equal the value in the *CSC* for any hardware context and context switching is enabled in *CCR* for that hardware context, then *TRAP_SOFT_CTXT_SWITCH* trap is generated for that context. Then, that trap is used to save and restore the context as explained earlier.

2.3.2 Sleep and Wake-up System

The sleep and wake-up system refers to two context control registers that only exist on context zero. By using these registers, processor can be set up to automatically request a reconfiguration when the interrupt request input of context zero is asserted. More specifically, the wakeup system will activate when all of the following conditions are met.

- The *S* flag in Sleep And Wake-up Control (*SAWC*) register is set.
- An interrupt is pending on context 0.
- Context 0 is not already active in the current configuration.
- There is no reconfiguration in progress.

When activated, the following actions are performed.

- A reconfiguration to the configuration stored in Wake-up Configuration (*WCFG*) register.
- *WCFG* is set to old configuration
- The *S* flag in *SAWC* is cleared

2.3.3 Requesting a Reconfiguration

In the ρ-VEX, there are three ways in which a reconfiguration can be requested.

1. Writing to the *CRR* context control register from a program running on the core.
2. Writing to the Bus Reconfiguration Request (*BCRR*) global control register from the debug bus. This mechanism is equivalent to the first, except it is triggered from outside the core.
3. Using the sleep and wake-up system, as described in Section 2.3.2.

Usually, when a reconfiguration is requested, the new configuration will be committed within the order of tens of cycles, depending on how long it takes the reconfiguration controller to pause the affected contexts. But some time reconfiguration may be rejected. The reasons of this rejection can be following:

- Another context or bus is requesting a new configuration simultaneously with reconfiguration request by the intended context. This behavior cause the lost in arbitration.
- The requested configuration could be invalid.

The requested configuration is valid if it complies following guidelines.

- Any context should be mapped to the power of two contiguous lane groups. For example, 0x2131 and 0x2221 are invalid configuration words because they violate this rule.
- The digits in the configuration word should start from zero to the number of hardware contexts minus one in order to select a context. The digit 8 can be used to disable the pipe lane group. This digit does not map the lane group to any context. For the ρ-VEX which does not support 8 hardware contexts the configuration word 0x7777 is invalid but 0x1238 is a valid one.
- The digits in the configuration word corresponding to the non existent pipe lane groups should be set to zero. For instance, configuration word 0x00008321 is valid for the ρ-VEX that is designed time configured for the four hardware contexts but 0x88888321 is invalid.
- A set of lane groups that are assigned to single context should aligned properly. For instance, 0x0112 is not valid word but 0x1122 is valid one.

2.4 Operating System

An operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware [29]. On the basis of purpose operating systems are classified into three categories [3]:

1. Time sharing operating system.
2. Real-time operating system.
3. Distributed operating systems

These categories are briefly explained below:

2.4.1 Time Sharing Operating System

Time sharing allows many users to share computer resources simultaneously. In other words, time sharing refers to the allocation of computer resources in time slots to several programs. The time sharing system provides the direct access to a large number of users where CPU time is divided among all the users on the scheduled basis. The OS allocates a fraction of time to each user. When this time is expired, it passes control to the next user on the system [34]. The time allowed is extremely small and the users are given the impression that they each have their own CPU and they are the sole owner of the CPU. These operating systems are preferably used for menu driven programs , dialogue driven programs etc. Time sharing operating system provides the advantage of quick response time, less CPU idle time and avoid duplication of software [14]. There are also some disadvantages of time sharing system like problem of data communication and problem of reliability etc [32].

2.4.2 Real-Time Operating System

This operating system executes the applications in real time. Real-time operating systems often use specialized scheduling algorithms, so that they can achieve a deterministic nature of behavior. The main objective of real time operating systems is their quick and predictable response to the events [23]. These kind of operating system have an event driven design or a time sharing one. An event driven system switches tasks only when an event of higher priority needs servicing, while time sharing operating systems switch tasks based on clock interrupts [13]. A real time systems reacts to outside events with in a time that is specified by the application. If this deadline is not met, then the answer can be considered erroneous. There are three types of real time systems [16]:

1. *Hard Real Time System*:

   Hard real-time systems must absolutely hit every deadline and if it misses just even one deadline, then the system is considered a failed one. Very few systems have this requirement. Some examples of such systems are nuclear systems, some medical applications such as pacemakers, a large number of defense applications, avionics, etc

2. *Soft Real Time System*

   Soft real time system can sustain some deadline misses and deadlines misses does not cause devastating results. For example, Streaming audio-video.

3. *Firm Real Time System*

   This kind of system falls between hard and soft real time systems. In such kind of systems, occasional deadline failures can be tolerated but if this issue persists then system may experience failure.

Well known real time operating systems are RTlinux , VX Works and Windows CE etc.

2.4.3 Distributed Operating System

A distributed operating system handles many independent computers which may be geographically apart. These independent computers communicate which each other through a communication network and exchange messages, data, programs etc between the cooperating computers. Each individual computer in distributed computing environment is called a *Node*. Distributed operating systems are very complex and difficult to write due to following reasons:

- There is a need to effectively manage a large number of distributed resources.
- Smooth communication between various *nodes* is needed.
- Security and privacy of these distributed resource is required.

The advantages of the distributed operating system includes flexibility of resource usage, reliability and shorter response time etc [14]. The typical applications of the distributed operating system includes electronic mail facility, a computerized banking system and air line reservation system.

As in this thesis project, we want to see the impact of the ρ-VEX reconfigurability on the real-time robotic applications, Real-Time Operating System (RTOS) class will be used. As it is already discussed that RTOS achieve deterministic nature of the behavior by using specialized algorithms. The major algorithms for the RTOS are discussed in the next section.

2.5 Real-Time Task Schedulers

Task scheduling essentially refers to determining the order in which tasks are to be taken-up for execution by the operating system. Every operating system employs one or more schedulers to schedule different tasks. Each scheduler is characterized by the scheduling algorithm it uses. The different concepts and terminologies that are used in scheduling algorithms are following:
Feasible schedule: A schedule is called a feasible schedule, if all the tasks in the schedule meets their respective timing constraints.
Proficient Scheduler: A scheduler (sch.1) is called a proficient scheduler than the other scheduler (sch.2), if it can feasibly schedule all the task sets that are scheduled by the other scheduler(sch.2) and this scheduler(sch.1) can schedule at least one more task set than the other(sch.2).
Optimal scheduler A real-time task scheduler is called optimal, if it can schedule all the task sets which can be scheduled by any other scheduler.
Scheduling Points: The scheduling points of the schedule are points on the time-line where the schedule decides which task is to be run next.
Preemptive Scheduler: A preemptive scheduler is the one which suspends lower priority task when a higher priority task arrives for execution. Thus, in preemptive scheduler it cannot be the case that higher priority is ready and waiting for execution and lower priority task is executing.
Utilization: The processor utilization or simply utilization of a task is the average time for which it executes per unit time interval. If T_i is a periodic task i, and e_i and p_i are execution time and period (respectively) of that task, then the utilization of task i will be $U_i = e_i/p_i$.
Jitter: jitter is the deviation of a periodic task from its strict periodic behavior. This can be due to imprecise clock, network congestion or bus contentions etc.

Scheduling algorithms for real-time operating system are broadly classified into three classes[25]:

1. Clock Driven

2. Event Driven
3. Hybrid

2.5.1 Clock Driven Schedulers

Clock driven schedulers make their scheduling decision only on the clock interrupt points. Therefore, in these schedulers scheduling points are determined by the timer interrupts. These schedulers are also called offline scheduler because all the details of the task schedule are fixed before running the tasks. As all the details of the task schedule is determined offline, therefore these scheduler have very less run-time overhead. As these schedulers have a fixed schedule for the tasks, they can not handle aperiodic and sporadic tasks. Therefore, this kind of schedulers are also called static scheduler. The most important clock driven schedulers are *Table-driven* and *Cyclic* schedulers.
Table Driven Schedulers: Table driven schedulers usually pre-compute which task to run when and store this schedule in the table at the time the system is configured or designed. It is sufficient to store schedule entries for the duration of one *major-cycle* or *hyper-period.* A *major-cycle* of set of tasks is an interval of time on the time line such that in each major cycle different tasks recur identically . if p_1, p_2, p_3 and p_4 are periods of the tasks in task set then *major-cycle* will be the $LCM(p_1, p_2, p_3, p_4)$.
Cyclic schedulers: These kind of schedulers are very popular for small embedded applications and are used extensively in the industry. The cyclic schedule repeats a precomputed schedule. This precomputed schedule is needed to store for one *major-cycle.* The *major-cycle* is further divided into one or more minor cycles which are called frames. The scheduling points of the cyclic scheduler occur at the frame boundaries. This means that a task can start it execution only at the start of the frame. The frame boundaries are defined by the interrupts of the timer.

The difference between *table-driven* and *cyclic* scheduler is that in *table-driven* scheduler timer is re-initiated at the start of each task and in *cyclic* the timer is initialised just once at the start of the application and then it keep repeating itself in a periodic manner. Therefore, *cyclic* are more efficient. In *table-driven*, next task can start its execution as soon as current task finishes its execution but in *cyclic* no task can start its execution before the start of the next frame. Therefore, if we ignore the overhead of timer initialisations then *table-driven* schedulers are more proficient than the *cyclic* ones.

2.5.2 Event Driven Scheduler

Clock driven scheduler are very efficient. But the prominent short coming of this class of schedulers is that they can not handle aperiodic and sporadic tasks. Moreover, as the number of tasks increases, it becomes very hard to find suitable schedule and frame size. Moreover, in each frame some of the processing time gets wasted which results in sub-optimal schedules. Event driven schedulers are designed to overcome all these short coming of *clock-driven* schedulers.

In *event driven* schedulers, scheduling points are determined by task completion and new task arrival events. The schedulers that belong to the *event driven* class are mostly preemptive in nature. Although *event driven* schedulers are more proficient than the *clock driven* ones but less efficient because they employ more complex scheduling algorithms. The most important *event driven* are *Earlier Deadline First (EDF)* and *Rate Monotonic Algorithm (RMA)*. Now, these two scheduler are discussed briefly:
EDF: In this scheduler, at every scheduling point the task having the earliest deadline is taken-up for the scheduling. A task set is schedulable under EDF, if and only if the total processor utilization due the task set is less than 1. For a periodic task set $T_1, T_2, ...,T_n$ the *EDF* schedulability condition can be expressed as:

$$\sum_{i=1}^{n} e_i/p_i = \sum_{i=1}^{n} u_i \leq 1 \tag{2.1}$$

where e_i, p_i and u_i are execution time, period and utilization of the respective tasks. *EDF* has been proven to be an optimal uni-processor algorithm [31]. Although *EDF* is an optimal algorithm, it has some short comings such as the transient overload problem.

Transient overload denotes an overload of the system for a very short time. Transient overload occurs when a task takes more time than was planned at design-time. When *EDF* is used to schedule the tasks, a task can miss its deadline due this transient overload. If it happens then there is no way to predict in *EDF* about how many and which tasks will also be suffered by this transient overload.

RMA: In this scheduler, on each scheduling point task with the highest priority is taken-up for the scheduling. In RMA, the priority of a task is directly proportional to the rate of occurrence. The higher the rate of occurrence, the higher will be the priority of the task. This also means the priority will be inversely proportional to the period of the task. These priorities are assigned to the tasks statically at the design or configuration time. RMA has been proved to be an optimal static priority algorithm [15], when tasks deadlines are equal to their periods.

2.5.3 Hybrid Scheduler

It is already discussed that scheduling points for the *clock driven* schedulers are determined by the clock interrupts. In case of *event driven* schedulers, scheduling points are determined by the arrival and completion of the tasks. In Hybrid schedulers, scheduling points are defined through both clock interrupts and event occurrence. Time-sliced round robin is a popular example of hybrid schedulers. It is a preemptive scheduling method. In round robin scheduling, tasks are selected for the execution in order from a queue. Then they are executed for a specific time which is called a slice. On the expiry of the slice, the task is sent back to the queue. It treats all the task equally, therefore, it is less proficient as compared to cyclic and table driven scheduling algorithms. This proficiency can be improved with just a very simple modification. It can be modified, so that it can change the slice according to priority or deadline of the task. In this way, high priority tasks will get more CPU time as compared to the lower priority ones.

2.6 Scheduling of Real-Time Tasks on Multiprocessors

The use of multiprocessors in real-time applications is becoming very common because of fault tolerance and faster response time they produce. To find an optimal scheduler for real-time independent tasks is much more difficult in multiprocessor systems as compared to uniprocessor ones that are discussed in the previous section. Determining an optimal scheduler for a set of independent real-time tasks on multiprocessors is an NP-hard problem [11].

Scheduling problem of real-time tasks on the multiprocessor systems consists of two sub-problems: allocation of tasks to the processors and scheduling of tasks on the individual processors. The problem of allocation of tasks to the processors is concerned with how to partition a set of tasks and then how to assign these tasks to the processors. This allocation of tasks can be static or dynamic. In the static allocation scheme, allocation of the tasks is fixed and does not change with the time. In this scheme, a bin packing approach is used to bind tasks to the processors. These kind of task allocation algorithms are also called partitioning algorithms. In the dynamic scheme, tasks are assigned to the processor as they arise. In this scheme, different instances of the task can be assigned to the different processors. Therefore, this kind task allocation scheme is also called *global allocation* scheme. After the successful assignment of tasks to the processors, we can consider the tasks on each processor individually. Therefore, the second phase of scheduling of task multi-processor systems becomes just like the scheduling of the tasks on uni-processor as discussed in Section 2.5.

In the partitioned task allocation scheme, tasks are assigned to the processors at design-time and they cannot migrate to the other processors during the run time. Each instance of the task will be run by the same processor. While in global task allocation scheme, tasks which are ready for execution are kept in the in one common priority queue and dispatched for the execution as soon as any of the processors is available. Therefore, different instances of the processor can run on different processors.

2.7 Characteristics of Embedded Real-Time Control Systems

Generally, embedded real-time control systems has a complex task sets consisting of both control and user applications. Typically embedded real-time control systems have the following scheduling objectives [37]:

- *High predictability*: All the deadlines for the hard real-time tasks should be guaranteed and high responsiveness should be provided to the soft real-time tasks. Moreover, predictability should also be guaranteed for the situation of transient loads.
- *Low scheduling overhead*: Embedded resources such as memory, computation capacity and power are limited. Therefore, scheduling overhead should be low.
- *Capability of handling hybrid task sets*: Embedded control applications are mostly consists of hybrid tasks (period, aperiodic and sporadic tasks). The scheduling

policy should be able to deal with such kinds of task sets.

- *Incorporating resource sharing*: Ideal scheduling algorithms are developed by assuming tasks are independent of each other and data sharing is a key characteristic of the typical control systems. Resource sharing in priority based preemptive system can cause priority inversion and unpredictable blocking of the tasks should be avoided.

In order to achieve above scheduling objectives, the scheduler for these systems should be chosen very carefully.

2.8 Conclusion

In this Chapter, background information of this thesis project was presented. In the first section, FPGAs were introduced, and the working and advantages of this technology were briefly discussed. In addition, it was explained how processors, designated as softcores, can be implemented in this technology. Subsequently, VLIW processors were introduced and their advantages were discussed.

This background established the basis for the introduction of the ρ-VEX. The ρ-VEX is a VLIW softcore processor, implemented on FPGA reconfigurable fabric. Being a VLIW processor, the ρ-VEX issues instructions in the bundles of two or more. This increases the IPC and, thereby, improves the performance of the applications. This processor is design-time configurable and run-time reconfigurable. This run-time reconfiguration feature can be used to exploit TLP and ILP of the programs. After this short discussion about ρ-VEX in general, those parts of the ρ-VEX were discussed in detail that were related to porting of an OS. These parts include the *run-time reconfiguration* mechanism, the *context loading and restoring* mechanism, and the *sleep and wake-up* system.

In the second part of this chapter, first the concept of an OS was given and different classes of an OS were introduced. Moreover, different factors which derive those classes were also discussed. In that discussion, more focus was given to Real-Time Operating Systems (RTOS). After that scheduling polices that are usually employed by the RTOS in a uni-processor and multi-processor environment were given. Moreover, properties, and pros and cons of the different scheduling polices were discussed.

This chapter finished with the discussion of scheduling objectives of real-time control systems. This discussion provided us the goals that we wanted to achieve by selecting a specific scheduling policy.

3 General Concepts and Analysis of FreeRTOS

In Chapter 2, the background of this project was explained. The ρ-VEX processor was introduced and its unique features were explained. Then an overview of OS was given and Real-Time Operating System (RTOS) was introduced. Subsequently, different schedulers for real-time systems and their pros and cons were discussed. That chapter concluded with the discussion of the characteristics of embedded real-time control systems.

In this chapter, the working of RTOS and FreeRTOS is explained in detail. The goal of this chapter is to establish the current status of FreeRTOS and what is needed to be done for the porting of this RTOS to the ρ-VEX.

General concepts and data structures of a typical RTOS are given in Section 3.1. Subsequently, reasons behind selection of FreeRTOS and working of the existing version of FreeRTOS are explained. After that, the requirements of the ported version of FreeRTOS, and challenges that are being faced to meet those requirements are discussed. This chapter concludes with the discussion of the real-time scheduler that is being employed for this thesis project.

3.1 Real-Time Operating System(RTOS)

A real-time operating system is supposed to run the applications in the real-time. The basic building block of any application written under RTOS is called a task. In RTOS, whenever an event occurs, it triggers the task that handles that event to execute. In other words, a specific task is generated in order to handle a particular event. Therefore, real-time tasks recur many times depending upon the happening of the relevant events. Most of the time real-time tasks recur after a certain fixed period of time but these tasks may also recur at random instants. So, the first time a task occurs, it is called first instance of that task. The second occurrence of that task is termed as the second instance and so on. Each task is expected to complete its execution within specified time that is called its deadline. In literature, two kinds of deadlines are used: absolute deadline and relative deadline. The absolute deadline is absolute value of the time (counted from zero) by which the results from the tasks are expected. Hence, an absolute deadline is the time interval from count 0 to the actual happening of the deadline as measured by some physical clock. Relative deadline is the interval between arrival of the task in the ready queue and instant at which deadline occurs. Every task should meet its deadline in order to run the application as expected. In order to ensure that all the tasks meet their deadlines, a scheduler is used. This scheduler tries to run the tasks in such a way that no task miss its deadline. The time duration between occurrence of the task generating event to the producing of the results by the task is called *Response Time*.

3.2 RTOS Architecture

Typical architecture of the RTOS is divided into three layers, as shown from the figure 3.1:

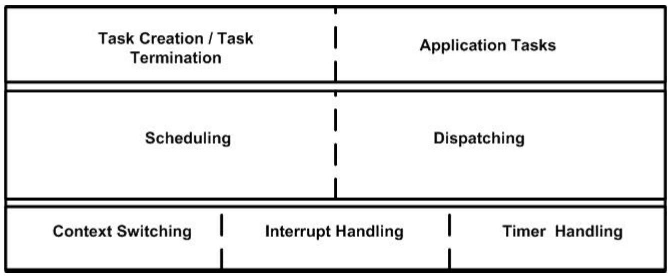


Figure 3.1: Typical RTOS Kernel Architecture

The top layer in this architecture is also called application layer. The responsibility of this layer is to create task and to terminate the task at the end of its execution. Moreover, user application tasks also reside in this layer. The responsibility of the middle layer is to schedule the tasks according to the scheduling algorithm and to assign the processing resources based upon the requirements of the tasks. Therefore, the middle layer consists of scheduler and dispatcher. The job of the scheduler is to determine which of the tasks should be in the running state. The dispatcher based upon the available processing resources moves some or all of them into the running state. The bottom layer interacts with the hardware platform. It consists of three components: context switching mechanism, interrupt handler, and timer handler. The context switching mechanism stores the context of already running tasks into the memory and loads the context of the tasks that are selected by the dispatcher into the hardware registers. The interrupt handler service the hardware and software interrupts. Based upon the source of interrupt, the interrupt handler runs the specific (piece of) code to service it. The timer handlers service the hardware timers. The RTOS executes based upon timer ticks of the hardware timer. The frequency of the timer ticks can be changed at run time to meet the system requirements. The following data structures are typically employed by a typical RTOS:

- Task Control Block (TCB): This is a data structure in which information about the task is stored. In particular, TCB contains all the parameters specified by the programmer at the creation of the task and also some temporary information. Typically, a TCB in RTOS consists of following parameters [9]:
  - Task ID: (It is) character string that is used by the system to refer the task.
  - The memory address corresponding to the first instruction of the task.
  - The worst case execution time of the task.
  - Current State (Ready, Running, Suspended etc).
  - Task period: it signifies after how much time task should execute again from its initial state.
  - Task priority: It tells the importance of the task with respect to the other tasks of the application.

  - A pointer to the stack, where the task context is stored.

- Blocked Queue: This data structure contains tasks which are blocked and are waiting for some event to happen. This event can be an I/O or timer tick etc. The tasks which are in blocked queue are not schedule by the scheduler.

- Ready Queue: contains tasks which are waiting for execution. These tasks are available for the scheduling and it is up to the scheduler to decide at what time instant it wants to run which one.

- Dormant Queue: This queue contains all the tasks which have finished the execution and are waiting for their next period or next task instance generating event.

- Running Task List: This list contains all the tasks which are running at the current instant of time. In a uni-processor system only a single task is in running task list but in multiprocessor system there could be more than one running tasks at the same time.

In its life cycle each task goes through the following states[9].

1. Spawn
2. Ready
3. Running
4. Dormant/Blocked

In the spawn state, tasks are created based upon the parameters like deadline, period, stack and stack address etc which are given by the programmer. Subsequently, tasks are stored in the ready queue after the creation. The tasks which are in the ready queue are said to be in the *ready* state. The *ready* state tasks can be scheduled by the scheduler. The scheduler, based upon the scheduling algorithm, selects a task from the *ready* queue and assigns it to the processor. The task that is being run by the processor is said to be in the running state. From the running state the task can proceed to these states:

1. Back to the *ready* state in case scheduler determines that there is a more important task in the *ready* queue that is needed to be run.

2. If the task has finished its execution then it will go to *dormant* queue.

3. If it is waiting for some event to happen then the the task will go to the blocked state.

It is clearly portrayed from Figure 3.2.

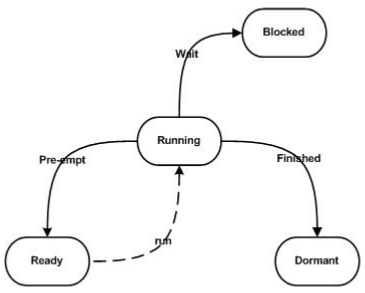


Figure 3.2: Traversing of the task from running state to other states

3.3 FreeRTOS

It is mentioned in Section 1.2 that there is a need for a RTOS for the ρ-VEX. Therefore, one of the options was to develop a RTOS from scratch but it would have taken a lot of time and energy. As there are large number of RTOS are already available in the market, it was decided to port an existing solution for the ρ-VEX. The following criteria had been established to select an existing RTOS for porting to the ρ-VEX:

- Complete Source code should be available.
- Source code is properly documented.
- Requires reasonable amount of time to port to the ρ-VEX.
- Easy to evaluate the performance of ported solution.

The first option that came under consideration was *Wind River*'s *VxWorks*. This RTOS has long history of being used in real-time mission critical applications. But this option was immediately dismissed because of its proprietary nature and source code is not freely available.

eCos RTOS was considered next. All the source code is freely available online under a non-commercial licence. Moreover, it is being using in a number of other platforms like arm, intel etc. But it was assessed that it will take a lot of time to port to the ρ-VEX because of its customizability features. As this RTOS can be customized for the specific applications. Therefore, this option was dismissed.

Linux was another solution that was considered as a potential candidate for the porting. Although it has complete and well-documented source code online, but in order to port Linux for the real-time requirements needs considerable amount of time. Moreover, most of the Linux kernel needs virtual memory support. Virtual memory support has been implemented in [20] for the ρ-VEX but this is not tested thoroughly by the users and not matured enough to be used in an other MSc thesis. Therefore, porting of Linux for this project is dismissed for this project.

FreeRTOS was the next option that was considered for porting on the ρ-VEX in order to meet the real-time requirements of the applications. FreeRTOS, not only, has well documented source code available online but also has already been used for a number of other platforms. Moreover, by analysing the code it was projected that the porting

Table 3.1: RTOS selection criteria for porting

Requirements	**eCos**	**VxWorks**	**RT- Linux**	**FreeRTOS**
Complete source code available	Yes	No	Yes	Yes
Source code properly documented	Yes	N/A	Yes	Yes
Required effort and time	high	N/A	very high	moderate
performance evaluation	easy	N/A	easy	easy

of this FreeRTOS can be done within the time frame of this thesis project. Hence, FreeRTOS was selected to port on ρ-VEX. This selection procedure is summarized in the Table 3.1.

3.3.1 Architecture of FreeRTOS

In order to port the FreeRTOS for the ρ-VEX, there is a need to develop understanding of components of the FreeRTOS and how they interact together to achieve required functionality. This section will provide brief description of the how FreeRTOS works and what are components of it.

The Figure 3.3 depicts how a task travels through different states during its life time.

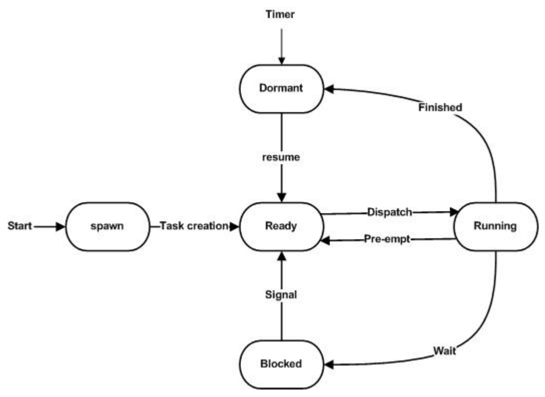


Figure 3.3: State diagram of the FreeRTOS

This figure portrays that FreeRTOS has almost same flow as of typical RTOS discussed in Section 3.1. Like typical RTOS, after the initializations, the tasks are created in the *spawn* state and are stored in the ready queue. From the ready queue, based upon the scheduling algorithm, tasks are assigned to the processor. As it is clear from Figure 3.3, the running task can go either in *dorman*t state or *blocked state*. The task go to the *dormant* state after finishing its execution and jump to the *blocked* state when it is waiting for some external event to happen. The task remains in the *dormant* queue until next period. The tasks remains in the *blocked* queue until the required external event has happened. As long as tasks are in the dormant or blocked queue, they are not considered for scheduling. From the Figure 3.3, it is also evident that there are two major components of the FreeRTOS:

1. Task-Resume/Activate mechanism.
2. Scheduler.

On each tick, the *Task-Resume* mechanism scans the *dormant* queue to find period of which one of the tasks has arrived. If this mechanism finds out that the period of any task has been arrived, then it moves that task to the *ready queue*, so that task can be available for for the scheduling. If some external event happens like a UART interrupt then the *Task-Activate* mechanism scans *blocked* queue in order to find whether that event is for any of the blocked tasks. If the event is for any of the blocked tasks then that task is released and moved into the *ready* queue. Once tasks are moved into the ready queue, (now) it is job of the scheduler to assign tasks to the processor. On each tick, like the *Task-Resume* mechanism, the scheduler is also called from the Interrupt Service Routine (ISR). Then the scheduler, based upon the scheduling policy, selects a task for execution and assigns tasks logically to the processor. After logical assignment of the task to processor, the dispatcher (is the one which) physically loads the task onto the processor.

3.4 Requirements

The principal objective of this thesis project is to create a version of FreeRTOS that can execute and schedule the real-time tasks efficiently by employing the the unique features of the ρ-VEX. These unique features of the ρ-VEX are discussed in Section 2.3. After thorough discussion with the members of *CE* group and by employing *requirement analysis* techniques, functional and non-functional requirements are identified for this project. Note that, in the following subsections, all the requirements are numbered sequentially regardless of their type and "functional" status.

3.4.1 Functional Requirements

Functional requirements specify the functionality that is expected from this project after its completion. The functional requirements of the ρ-VEX ported FreeRTOS are listed below:

1. FreeRTOS must provide the capability to change the ρ-VEX configurations at run-time.

   Description: In the hardware dependent part of FreeRTOS modification will be made, so that ρ-VEX reconfiguration capabilities can be accessed by the kernel of FreeRTOS at run time.

   Rationale: As the ρ-VEX has the capability to change issue width and the number of hardware threads at the run time, with this modification in FreeRTOS, ILP and TLP can be exploited at the run time.

2. Application tasks must have access to an API providing capability to make requests for the changes in processing resources.

Description: An API will be added to hardware independent layer of FreeRTOS, so that task can register their requirements to the FreeRTOS.

Rationale: By using these API, tasks can explicitly request to change the processing resources.

3. FreeRTOS must allow concurrent scheduling of tasks on the ρ-VEX.

   Description: Modifications will be made to the FreeRTOS, so that it is able to schedule tasks on the multiple hardware contexts of ρ-VEX concurrently provided that enough resources are available.

   Rationale: In ρ-VEX multiple hardware contexts are actually virtual cores. These virtual cores can execute work load independent of each other with different issue widths. Therefore, to utilize the capabilities of the processor, FreeRTOS must be able to schedule tasks on these hardware contexts concurrently.

4. Concurrent tasks should not interfere with each other

   Description: Modification in the hardware independent layer of the FreeRTOS will be done, so that each virtual core can execute task without interfering with the others.

   Rationale: The hardware of the ρ-VEX has the capability to handle the concurrent tasks, so the FreeRTOS should have have some mechanism to exploit this feature. In the single-core version of FreeRTOS, only one task can be scheduled at a time and there is only one suspended queue. Due to this kind of OS architecture, the full power of ρ-VEX cannot be exploited. So, there is a need of modifications which will enable it to handle concurrent tasks.

5. If any task misses its deadline then it must be killed immediately.

   Description: Modification in the scheduler of the FreeRTOS will be done, so that all the resources of the tasks can be taken back if that task cannot complete within the stipulated time.

   Rationale: In a real-time OS, even correct output does not have any value if it arrives after the required time. Therefore, if any task fails to produce output within the given deadline then it should be killed immediately. Because its output does not have any value after the deadline miss and this action will also prevent the effect of the deadline miss to propagate to (the) other tasks.

6. The cache affinity of the tasks may be maintained.

   Description: Modifications will be made in the FreeRTOS, so that while reconfiguring the processing resources at the run-time, the cache affinity of the already running tasks will be maintained.

   Rationale: As the ρ-VEX can be reconfigured at the run-time with respect to the processing resources being utilised. Therefore, while reconfiguring these resources the cache area used by these resources must be taken into consideration. Owing to the fact that, if cache utilisation is not considered while the resource reconfiguration then it may result in a lot of cache misses. Consequently, the reconfiguration may degrade the performance of the system instead of improving it.

7. If some higher priority task needs more resources then processing resources should be taken back from the lowest priority running tasks.

   Description: Modifications in the kernel will be made to enable it to take back processing resources from a lower priority running task, if more resources are needed by a higher priority tasks.

   Rationale: (High priority task means the task which has high importance). If high priority task needs more resources in order to meet its deadline and more resources are not available, then in order to satisfy the requirements of the higher priority task, resources should be taken back from the lower priority ones.

8. If multiple tasks are running and it is required to preempt a task in favour of a higher priority task, then the lowest priority task should be preempted among all the running tasks

   Description: Modifications in the FreeRTOS scheduler will be made, so that it can preempt the lowest priority task among all running tasks in the favour of the high priority task (that is needed to execute).

   Rationale: If any task is preempted, its context is needed to be stored back into the memory before the context of the new task can be loaded from the memory into register. This process can cause a lot of overhead. If multiple tasks are running then lowest priority task should bear all this overhead, so that higher priority tasks can complete its execution with the minimum overhead of the system.

9. Resource reconfiguration requests must be fulfilled as quickly as possible.

   Description: Modifications in FreeRTOS will be done in order to reconfigure the hardware of the ρ-VEX in the shortest possible time.

   Rationale: The request for the more resources comes at the moment when that task needs more resources. So, in order to fulfill the task requirements effectively resource reconfiguration should be done as quickly as possible. Otherwise, if reconfiguration takes more time, then task requirements for more resources might be over by the task get more resources.

10. Modifications made to FreeRTOS must scale transparently for use with the n hardware contexts of the ρ-VEX.

    Description: All the modification should scale transparently with n number of virtual cores of the ρ-VEX.

    Rationale: ρ-VEX is an experimental processor and in future it can have any number of virtual cores depending upon the future design and application of this processor. In order to make this implementation relevant for all future designs of the ρ-VEX , this implementation should scale transparently with number of virtual cores.

3.4.2 Non-functional Requirements

Non-functional requirements specify those characteristics which do not change the behavior of the system but makes it easy to use and upgrade. The non-functional requirements of the modified FreeRTOS for the ρ-VEX is listed below:

1. Source code must be properly documented

   Description: source code will be properly documented with the comments, so that other programmer can easily understand.

   Rationale: All this effort will be of little use, if nobody else can use it later. In order to make changes and improve the current implementation, it should be properly documented, so that others can contribute to this work in future.

2. Task development must not be made more difficult

   Description: The process of converting a task of between 250 and 350 lines in length designed for use with the unmodified version of FreeRTOS, to make it compatible for use with the modifications made in this project must take a developer familiar with both versions no longer than 5 minutes [26].

   Rationale: One of the virtues of FreeRTOS its simplicity. In order to minimise the effect on application development time, it is important that this aspect of the operating system is preserved through the modifications made in this project

3. Standby power consumption must be minimum.

   Description: Modifications will be made in the FreeRTOS, so that it can shutdown those resources which are not utilised by any task. It will keep the stand by power consumption to the minimum level.

   Rationale: The ρ-VEX is mainly intended for embedded systems. Most of the embedded system are battery powered. Keeping standby power to the minimum level will help to last long the battery and in result improve the usability of the embedded devices.

3.5 FreeRTOS Porting Challenges on the ρ-VEX

The existing status and working of FreeRTOS is already briefly described in Section 3.3. In order to create a newer version of FreeRTOS which meets all the requirements of Section 3.4 the following challenges are identified:

1. As the baseline FreeRTOS is designed to run a single task at a time, but the ρ-VEX has four virtual cores. This means this processor has the capability to run at max four tasks in parallel. In order to enable the the existing FreeRTOS to run multiple tasks, there is need of new queue handling mechanism, a new strategy to bring task from the dormant state to ready state and to bring back the running tasks to the dormant or ready state.

2. As the ρ-VEX is run time reconfigurable in term of issue width and some other parameters, there is need to develop a mechanism which can change these parameters while processor is running.

3. As there are four contexts in the ρ-VEX, which can be considered as four virtual cores. The coordination among these four cores is a real challenge. Here coordination means, there is need to devise mechanism which can instructs task on different cores about the pre-emptions, allocation and de-allocation of resources etc.

3.6 Scheduling Policy for the ρ-VEX Ported FreeRTOS and Extended RM Algorithm

The different scheduling policies for the uni-core and multicore systems are already discussed in Section 2.5. The pros and cons of each policy is also discussed there. Moreover, Section 2.5 also summarized the scheduling complexities of multi-core systems. As already discussed, the task scheduling problem on multi-core systems consists of two sub problems: (1) Task allocation to the processors and (1) Scheduling of tasks on the individual processor. The ρ-VEX is also multi-core processor in a sense that it can run multiple hardware threads. Therefore, scheduling real-time tasks on the ρ-VEX also consists of the same sub-problems. There are mainly two types of task allocation schemes: global allocation and partitioned allocation. The details about these allocation schemes are also discussed in Section 2.5.

Although the predominant scheme in the multicore hard real-time systems is partitioned allocation[1], but for task allocation on the ρ-VEX global allocation scheme has been selected. The reasons behind the selection of this scheme for the ρ-VEX are the following:

- In the ρ-VEX, the number of running hardware contexts (virtual cores) may vary at run-time with processing demand of the executing tasks. If we use a partitioned allocation scheme then tasks which are assigned to a hardware context will keep on waiting even though other hardware contexts are free to schedule tasks. Global allocation scheme allows us to allocate tasks to any available virtual core. In this way, task schedulability of a task set may be improved on the ρ-VEX.

- In the global allocation scheme, tasks are selected from the common queue to execute on any available processor. So the migration of different task instance can happen from on processor to the other. This migration can cause a lot of overhead due to cache misses and reading from the memory. But in the ρ-VEX, if once the task is assigned to one of the hardware contexts and another task comes in, (then) instead of migrating the current running task to the other cores, the new task can be assigned to other virtual cores. In addition, the ρ-VEX has two levels of task preemption.

    1. Task context remains in the hardware registers just processing resources are taken back to satisfy the requirements of the higher priority task.

2. Currently running task is taken out from the hardware registers to make room for the higher priority tasks.

At the first level only processing resources assigned to certain hardware are assigned to the other hardware context. In that case, context of the task remains in the registers of hardware context. And when processing resources are free then the task can resume it execution without any overhead of memory accesses because there is no context saving and restoring mechanism is involved.

In the second level, if there are tasks which are already occupying all the hardware contexts and a new higher priority task wants to run then one of already running task should be send back to the memory in order to load new task into the registers of the hardware context. This involves saving current running task in the memory ad loading a new task from memory into the hardware registers. This may cause a lot of overhead. But in ρ-VEX, this overhead can be minimized by employing different techniques. One of the the technique is to preempt the lowest priority task among the currently running ones, so that higher priority tasks can always complete its execution without much overhead.

- It is well know result of queueing theory that single-queue scheduling produces better average response times than queue-per-processor scheduling [22].

Based upon the above reasons, global task allocation has been selected for this thesis project. So, now all the tasks to be executed on ρ-VEX will reside in a single ready-queue. Now, there is a need of a scheduler which can pick the tasks from a single queue and can assign them to the virtual cores efficiently. At first *EDF* was considered for this thesis project. Although it is considered as a optimal dynamic priority algorithm but it loses its predictability in the event of transient loads. The details about this problem can be seen in Section 2.5. As it is already discussed in Section 2.7, high predictability is one of the major scheduling objective in embedded real-time control systems and we are also aiming this implementation for real-time systems. Therefore, *EDF* is rejected for this thesis project. *RM* algorithm was considered next. It is a fixed priority optimal algorithm. This algorithm has a number characteristics that can be beneficial for real-time control systems. Some of the salient features of this algorithm are listed below:

- *Predictability*: High predictability is one of the proven superiorities of RM algorithm and a deterministic schedule can be calculated offline. Moreover, RM scheduler guarantees predictable response in the situations of transient overload. This makes RM scheduler favorite in applications where predictability is very important.

- *Low scheduling overhead*: As compared to dynamic priority schedulers, RM algorithm has very small scheduling overhead, since all the priorities are computed offline.

- *Engineering simplicity*: There are extensive studies and implementations of RM algorithm in embedded real-time operating systems. These experiences are quite helpful in the perspective of engineering simplicity.

Subsequently, *clock driven* schedulers were considered. These schedulers are more efficient than *RM* because complete schedule is computed offline but can not handle aperiodic and sporadic tasks. The details about cyclic algorithms is already discussed in Section 2.5.1.

As we have seen that each scheduling algorithm algorithm has its own merits and demerits. For this thesis project, we want the efficiency of *cyclic* algorithm and predictability and responsiveness of RM algorithm.Therefore, we decided to implement an hybrid approach, where scheduling points are both depends upon timer interrupts like *cyclic* algorithms and also at the completion of the tasks like pure *RM* algorithm.

Therefore, scheduling event in this thesis project will happen both at each timer interrupt and also at the completion of each task. So our scheduling algorithm is not pure *RM* but extended with *cyclic* algorithm.

3.7 Conclusion

This chapter started with the discussion of basic concepts of the RTOS. In the first section of this chapter, RTOS jargon was introduced and the architecture of typical RTOS was presented. The architecture of the typical RTOS consists of three layers: application layer, kernel layer and hardware specific layer. In the application layer, user tasks resides and, moreover, task creation and termination is also initiated from this layer. The control layer performs management of all the resources of the system and hardware specific layer interacts with the underlying hardware. Moreover, how an application task traverses through different states during its life time was also explained in this section.

Subsequently, there were a number of different types of RTOS that were considered and a selection criteria for the best candidate for porting on ρ-VEX was established. All those potential candidates were compared with each other based upon the selection criteria. From this selection procedure, FreeRTOS came out be a clear winner. Therefore, FreeRTOS was selected as an RTOS that would be ported in this project to the ρ-VEX.

In the next section, functional and non-function requirements of the ρ-VEX ported version of FreeRTOS were given. These requirements were elicited from the project description and from the discussions within the CE group of TU Delft.

Subsequently, from the architecture of the FreeRTOS and requirements of the ρ-VEX ported version of FreeRTOS, the challenges in porting the FreeRTOS were described. In the last section of this chapter, the procedure for the selection of the task scheduling policy on ρ-VEX was described. As the ρ-VEX has four virtual cores, the task scheduling problem like in other muticore-core processors was subdivided into (1) task allocation and (2) task scheduling on the individual processors. For this thesis project, the global task allocation scheme was selected due to the run-time reconfigurability of ρ-VEX, and RM scheduler was selected in order to schedule tasks on the ρ-VEX. The RM scheduler was selected because of its predictable response to every type and condition of workload. Moreover, in order to make the RM scheduler efficient, it was enhanced with the clock driven scheduler.

Implementation 4

In the previous chapter, concepts related to RTOS and reasons behind the selection of FreeRTOS as our target RTOS were given. Subsequently, the functional and non-functional requirements for the ρ-VEX port of FreeRTOS were described. In the next section, the challenges that are being faced in order to meet given requirements were described. The previous chapter concluded with the motivation behind the selection of the RM scheduler and necessary changes in the pure RM scheduler.

In this chapter all the implementation and modifications are described which are done in FreeRTOS to meet the requirements of Section 3.4. The goal of this chapter is to describe all the work that is done to create a running version of FreeRTOS for ρ-VEX.

All the modifications that are done in the existing version of FreeRTOS, are described in Section 4.1. These modifications include the implementation of *Extended RM* scheduler, modifications in the *Task Resuming* mechanism to accommodate four hardware contexts of the ρ-VEX and the *Run Time Reconfiguration Request* mechanism. Subsequently, the architecture of the *control layer*, and its complete implementation is described in detail. After that, the implementation of the *Standby power saving mechanism* is described. This chapter concludes with the description of how all these modifications are combined together to get working FreeRTOS for ρ-VEX.

4.1 Modifications and Porting of FreeRTOS

Before getting into the details of all the modifications, there is a need to explain some variables and data structures that are used in these modifications. The name and purpose some of important variables is explained below:

- *configWords*: This variable contains the active hardware configuration. Its value could be; e.g. 0x3210, 1st digit from right tells which hardware context is running at the 1st and 2nd lanes, 2nd digit tells which hardware context running at 3rd and 4th lanes, 3rd digit for the lanes 5th and 6th, and 4th digit for the lanes 7th and 8th. Each digit is corresponding to issue width of 2. If some digit positions have the same context value then this means that all those lane groups are being utilized by a single context. For example 0x1120 means context 0 is being run on lanes 1 and 2 and context 2 is on lanes 2 and 3. (Moreover) Each one of them has the issue width of 2. But the 2nd and 3rd digit of the configuration has value 1. This means context 1 is being run by 5-8 lanes and has the issue width of 4.

- *STHCM[HardwareContexts]*: This variable is an array of structures of type *TCB* (and typical *TCB* is explained in Section 3.1). The *STHCM* data structure keeps track of which task is being run by which hardware context. Each position in this array represents a hardware context number and the *TCB* at that position is the task that is being run by that hardware context. For example, the task at position 0 of this array is run by *hardware context 0.* That is why the size of this variable is equal to the number of hardware contexts.

- *availCores*: This variable keeps track of the issue-width currently in use.
- *LanGroups*: Lane groups availability or unavailability is tracked by this variable.
- *vTCB*: This variable keeps track of the highest priority task in the ready queue.
- *RSC_Register[HardwareContexts]*: This is an array of pointers in which each index except 0 points to the RSC register of the corresponding hardware context. For instance, the 1st index of this array points to to RSC register of the *hardware context 1.* Whenever the RSC value is different from the CSC register value in any context , then a trap is generated that is used to load and restore the task context in the corresponding hardware context registers.
- *CRR_Register*: This variable stores the intermediate values of hardware configurations while final value is being computed for the ρ-VEX. Once the final value is computed then value from this value is written to hardware Context Reconfiguration Request (*CRR*) register.

The following sub-sections explains the modifications that are done in order to port FreeRTOS for ρ-VEX.

4.1.1 Task-Resuming Mechanism

The job of the *Task-Resuming* mechanism is to bring the tasks from the dormant queue into the the ready queue based upon the periods of the tasks.

In the un-modified FreeRTOS, the task go to the dormant queue after completing its execution and wait for the next period. On each OS-tick, the *Task-Resuming* mechanism is called from the timer ISR. For each tick, it scans the *dormant* queue in order to determine whether any task's period has arrived or not. If the *Task-Resuming* mechanism find out that any task's period has been arrived then it moves that task from the dormant queue into the ready queue.

ρ-VEX has multiple hardware contexts that are actually virtual cores. The tasks can be run by these hardware contexts independently from each other. After completing the execution, the running tasks should be sent to the dormant queue. As the ρ-VEX processor can have more than one tasks in the running state, it is also possible that they will finish their execution at the same time. Therefore, they should be sent in the *dormant* state at the same moment. If there is just single *dormant* queue then the hardware contexts running those just finished tasks might try to access that *dormant* queue simultaneously. In that case they can block the access of each other or can corrupt each other in the *dormant* queue.

In order to solve this problem two options were considered:

1. A *mediation* mechanism, that if determines that more than one tasks from the different hardware contexts are trying to get access to the same *dormant* queue,

then it blocks all the tasks except one. When that task completes its access to the *dormant* queue then *mediation* mechanism can allow one of the already blocked tasks to get access to the *dormant* queue. This continues until all the tasks are moved into the dormant queues.

2. Create as many *dormant* queues as the number of hardware contexts. So that finished tasks from each hardware context can go to their own separate queue.

The problem with the first option was the waiting time to get access to the *dormant* queue. It depends upon number of tasks which are waiting to get access. If RTOS is running a number of tasks then those tasks might have to wait a lot before they can be stored in *dormant* queue. Therefore, this option was dismissed.

The second option seems more elegant. Here, tasks don't have to wait before they can get access to the *dormant* queue. But, in this approach, *Task-Resuming* mechanism has to scan all the *dormant* queues instead of just one, in order to move the tasks from the *dormant* queues to the *ready* queue. Number of the *dormant* queues depends upon the hardware contexts that are constant. Therefore, scanning time of all the *dormant* queues will not involve too much overhead.

So, FreeRTOS is modified with multiple *dormant* queues. So that every task after finishing its execution can go to separate *dormant* queue without interfering with tasks coming from the other hardware contexts. The modified *Task-Resuming* mechanism is depicted with the highlighted part of the Figure 4.1.

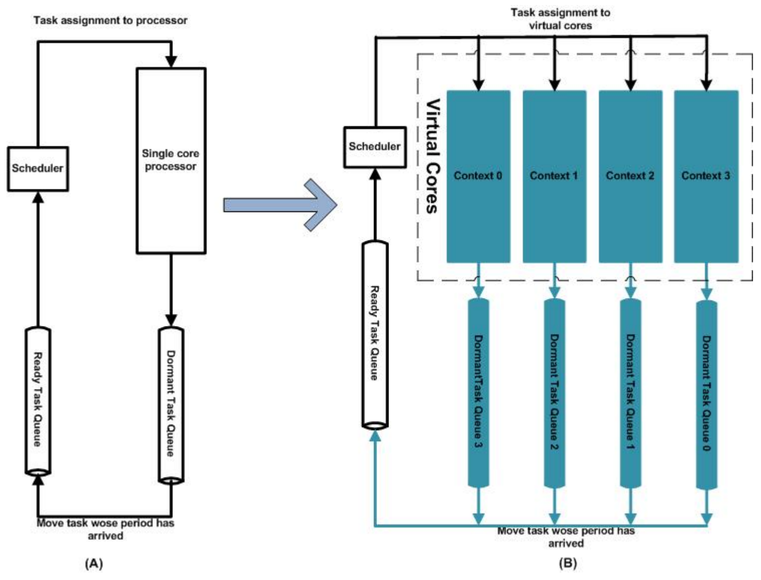


Figure 4.1: Modified Task-Resuming mechanism

In Figure 4.1, diagram *A* depicts the *Task-Resuming* mechanism and the *Scheduler* without modification and in "B" the highlighted portion is the modified *Task-Resuming* mechanism. There are 4 virtual cores and each core has its own *dormant* queue. All these queues are scanned in order to move the tasks from the *dormant* queue to the *ready* queue.

4.1.2 Extended RM-Scheduler

The scheduler for the ρ-VEX decides how many tasks should be in the running state and which task should be run by which hardware context. This scheduler is called on each OS-tick instead of on arrival or completion of the tasks (as in the pure RM scheduler). Like the original RM Scheduler, this modified RM-Scheduler also runs highest priority tasks first. As there could be multiple tasks available in the ready queue that are needed to be assigned to the available hardware contexts, the scheduler first selects the highest priority task and tries to assign it to the suitable hardware context. This assignment of the tasks to the hardware contexts continue until any of the following conditions are met:

- There are no more tasks available in the ready queue.

- All the hardware contexts are assigned with the tasks.

Whenever, the scheduler tries to assign tasks to the hardware contexts, then there could be three scenarios:

1. *Assign task to the first available hardware context*: This scenario happens when any of the contexts is free. Then new task pointed by *vTCB* is assigned to the available latest position in *STHCM*. After the assignment of the task to the *STHCM*, this task is removed from the ready queue. FreeRTOS stores all tasks in the *ready* queue in the descending order of their priorities, so, whenever ready queue is accessed again then the highest task would be next task that has lower priority than the just assigned task but higher than the rest of the tasks in the ready queue.

2. *Preempt the already running task*: This scenario happens when the task pointed by *vTCB* has higher priority than any of the already running tasks and no hardware context is free. Then the lowest priority task in *STHCM* is pushed back into the *ready* queue and at its position in *STHCM* data structure new task is assigned. After the assignment, just like previous scenario, the task pointed by *vTCB* is removed from the *ready* queue.

3. *No change in the already assigned assigned tasks to the hardware contexts*: When already running tasks have the higher priority than the tasks available in the *ready* queue and no hardware context is free. At that moment, nothing happens and the current tasks have to wait for the already running tasks to finish.

This assignment of the tasks to the hardware contexts is depicted with the following pseudocode:

Algorithm 1 Scheduling Based upon RM Algorithm

```
1: function RMSCHEDULINGWITHMULTIPLECONTEXTS(void)
2: $CRTs = GET\_NUM\_of\_CURRENT\_READY\_TASKS()$
3:     for $j = 0$ to $(HardwareContexts - 1)\&\& \ (CRTs > 0)$ do
4:         $vTCB = SELECT\_HIGHEST\_PRIORITY\_TASK()$
5:         if $(STHCM[j] == Null)||(vTCB \to priority >= STHCM[j] \to priority)$ then
6:             if $(STHCM[j]! = Null) \ ||(vTCB \to priority >= STHCM[j] \to priority)$ then
7:                 $VC = j$
8:                 for $h = j$ to $HardwareContexts$ do
9:                     if $(STHCM[h] == Null)$ then
10:                        $VC = h$
11:                        $break$
12:                    else if $(STHCM[VC] \to priority > STHCM[h] \to priority)$ then
13:                        $VC = h$
14:                    end if
15:                end for
16:                if $(STHCM[VC]! = Null)$ then
17:                    $temp1 = STHCM[VC] \to allocR$
18:                    $temp2 = STHCM[VC] \to allocL$
19:                    $STHCM[VC] \to allocR = ZeroResource$
20:                    $STHCM[VC] \to allocL = ZeroLaneGroups$
21:                    $AddTaskToReadyList(STHCM[VC])$
22:                    $CRTs = GET\_NUM\_of\_CURRENT\_READY\_TASKS()$
23:                end if
24:                $decrement(CRTs)$
25:                $STHCM[VC] = vTCB$
26:                $STHCM[VC] \to allocR = temp1$
27:                $STHCM[VC] \to allocL = temp2$
28:                $RemoveTaskFromReadyList(vTCB)$
29:            end if
30:        else
31:            $decrement(CRTs)$
32:            $RemoveTaskFromReadyList(vTCB)$
33:            $STHCM[j] = vTCB$
34:        end if
35:    end for
36: end function
```

Scheduler access the *ready* queue in order to get the total number of tasks in that queue. In the *ready* queue all the tasks are stored in descending order of their priority. *Ready* queue is a linked list of TCBs. Scheduler transverses the linked list (from beginning until the end of list) and counts every TCB along the way. Then it assigns the count value to the variable *CRTs*. This variable is used to keep track of total number of tasks in the *ready* queue. Now, the tasks are assigned to the hardware contexts. In order to keep compliance with the *RM* scheduler, the highest priority tasks in the *ready* queue should be assigned to the hardware contexts. In order to get the highest priority task, the ready queue is accessed again. But instead of going through the complete ready queue, the first available task is selected from the *ready* queue. First available task is selected because tasks are always stored in descending order of the priority in the *ready* queue. Then this selected task is assigned to the variable *vTCB*. Subsequently, depending upon the three scenarios that are discussed above, this *vTCB* is either assigned to any of the hardware contexts or its assignment is deferred.

The modified *RM-Scheduler* is shown with the highlighted part of the Figure 4.2.

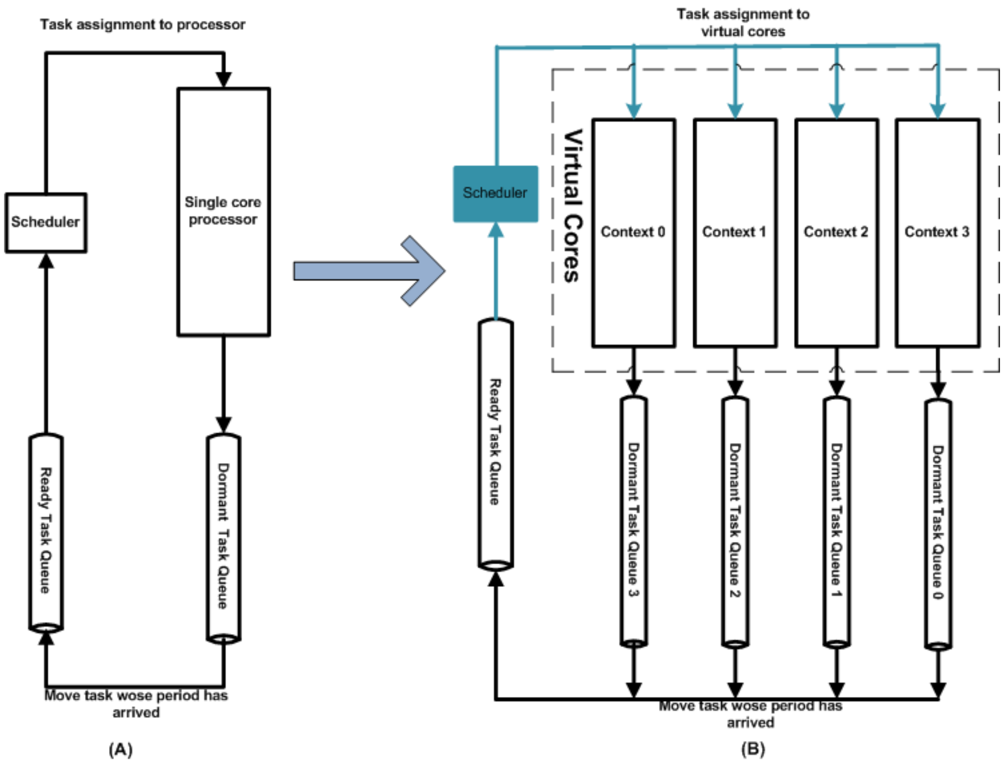


Figure 4.2: Highlighted portion shows modified the RM Scheduler

In this figure, diagram *A* depicts the unmodified *scheduler* and the *Task-Resuming* mechanism, and *B* depicts the modified versions of them. The highlighted part of this figure, portrays the modifications that are made in the original implementation of the *scheduler*. In the original implementation, the *scheduler* reads the *ready* queue only one time and assigns that task to the single core processor. In the modified implementation, the *scheduler* keeps reading the *ready* queue until there is no task in the *ready* queue or all the hardware contexts are occupied by the higher priority tasks (than in the *ready* queue).

4.1.3 Run Time Reconfiguration Request

As ρ-VEX is reconfigurable processor, therefore, any task can request for more or less issue width than it currently has. Issue width is the number of lane groups being utilised by the hardware context. In the current implementation, the tasks can make lane groups requests at any time during their execution but this request, for the time being, is processed at the next OS-tick. The request is fully or partially fulfilled, or declined based upon following factors:

- The priority of the resource requesting task with respect to other running tasks.
- The number of available lane groups that comply lane group assignment guidelines, discussed in Section 2.3.3.

All the running tasks resides in *STHCM[HardwareContexts]* data structure. So, in order to upgrade the resources at the run time, the resource requirements of the respective task is upgraded just in this data structure and rest of the things are handled by the *Control Layer* at the next OS-tick.

4.2 Control Layer

The control layer assigns (proper) lane groups to the hardware contexts. This allocation of the lane groups depends upon the priority and the resource requirements of the tasks. After the allocation of lane groups, the control layer configure the ρ-VEX by requesting a new configuration as explained in Section 2.3.3. The functionality of the control layer can be divided into following sub tasks:

- Retrieving the released resources from the running hardware contexts.
- Allocation of the lane Groups.
- ρ-VEX reconfiguration.

These sub-tasks of the control layer are also depicted from the Figure 4.3. From the figure, it can be seen that first freed resources are retrieved by the control layer then assignments of the lane groups to the hardware contexts are done based upon the priority and requirements of the tasks. While assigning lane groups to the hardware contexts, the control layer also keeps following things in consideration:

- Cache Affinity of the tasks.
- The priority of the tasks that are being run by the hardware contexts.

This figure also shows that after the allocation of the lane groups to the hardware contexts, ρ-VEX is configured accordingly.

The important data structures that will be used in the control layer are already discussed in Section 4.1. The sub-tasks of the control layer which are briefly discussed above are explained in more detail in the next sections.

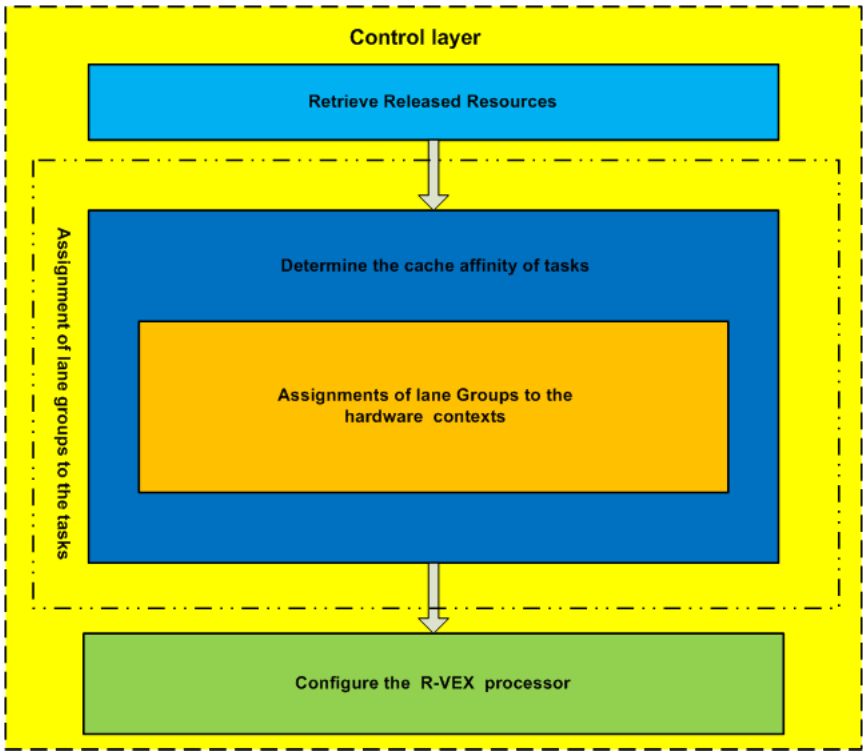

Control layer
Retrieve Released Resources
Assignment of lane groups to the tasks
Determine the cache affinity of tasks
Assignments of lane Groups to the hardware contexts
Configure the R-VEX processor

Figure 4.3: Control layer Architecture

4.2.1 Retrieving the Released Resources

The first action that is performed by the control layer is to get back the resources that are released by the hardware contexts. After finishing the assigned tasks, each hardware context replaces *TCB* in the *STHCM* data structure with the *NULL*. It is just a flag to the control layer that indicates that task does not need hardware resources any more. After asserting this flag, each context starts running the *idle* task. The *idle* task is nothing more than infinite loop. As position of each TCB in the *STHCM* array is the number of the hardware context that runs that task (corresponding to the TCB). So, when control layer sees *NULL* in any position of the *STHCM* array, it realizes that lane groups corresponding to that hardware context has been released. Then the control layer accesses the *STHCM* data structure to get back the lane groups from that hardware contexts. After getting access to the (to be freed) lane groups, the control layer replaces the context number at the corresponding position in *configWords* with the value *8*. This special value clock gates the corresponding portion of the ρ-VEX. Then the control layer increments *availCores* by the number of released issue width divides by 2. For example, if some context has 4 issue width then after freeing of the resources 2 will be added to the *availCores* and lane groups are added to *LanGroups* variable.

4.2.2 Assignment of the Lane Groups to the Tasks

By this moment tasks are already assigned to the *STHCM* data structure by the *scheduler*. Now the control layer will assign the lane groups to the tasks that resides in *STHCM*. The lane groups are assigned to the hardware contexts in the descending order of the priority of the tasks that resides on those hardware contexts. The highest priority task is assigned the lane groups first and then comes the turn of second highest priority task and so on. This assignment of lane groups continues until any one of the following conditions are met:

- No more lane groups are available
- All the tasks that resides in the *STHCM* data structure are assigned with the lane groups.

While assigning the lane groups to the tasks following scenario may happen:

1. The lane groups requested by the task are less than or equal to available lane groups.
2. The requested lane groups are greater than available ones.

Scenario #1: when requested lane groups are less than the available lane groups, then there is no need to suspend any of the already running hardware contexts in order to get the required resources.
This means that current resource requesting task can fit with the already running tasks. But in order to efficiently assign the lane groups to all the hardware contexts, the control layer may change the assigned lane group positions, if cache affinity is not considered. As each lane group is assigned with specific area of the cache, if a lane group from a different position is assigned then the data and program cache will be populated again. Then any task that is already running will keep on running along with the new tasks but this may cause a lot of cache misses and may result in performance degradation. So while assigning the lane groups to the task, it should be kept in consideration that this assignment of the lane groups does not effect cache performance of the other already running higher priority tasks. In order to keep cache affinity of the higher priority tasks, following scenarios are considered while designing control layer:

- If the current task does not already have lane groups then that task does not need any cache affinity, so any of the available lane groups can be assigned.
- If the current task already has some lane groups and at the current moment requested lane groups are equal to allocated lane groups, then there is no need to switch to the other lane groups. In order to avoid the cache misses the same lane groups are retained.
- If the current task has already allocated lane groups but presently requested resources are less than the allocated ones. Then during the allocation of the new lane groups all the available lane groups are ignored and control layer only removes extra lane groups from the already allocated lane groups.

- The current task already has resources but new requested resources are greater than already allocated ones. In this case, more resources takes preference over the cache affinity and requested resource are allocated without considering the cache affinity.

The implementation of these scenarios is depicted in Algorithm 2. In this algorithm, lines 2-4 cover the case when there are no lane groups already assigned to the hardware context and there is no need for the cache affinity. The lines 6-8 implement the scenario when already allocated lane groups are equal to the requested lane groups. So the already allocated lane groups should be retained. Lines 10-15 cover the case when requested lane groups are less than allocated ones. In that case, cache affinity is maintained by leaving extra sources. The 17-22 lines cover the case when allocated lane groups are less than the requested lane groups. In the last case, cache affinity is not considered, only preference is given to more resources. The output of this step is *cacheAffinitMask*. This mask hides all those lane groups from allocation which can affect the cache affinity of the other tasks.

Algorithm 2 Preserving cache affinity, when the available resources are greater than or equal to requested ones.

```
1:  function cacheAffinityForSufficientAvailableResouces(void)
2:      if $STHCM[ordered[j]] \rightarrow allocL == 0U$ then
3:          $reqR = STHCM[ordered[j]] \rightarrow reqR$        ▷ Requested issue-width of the task
4:          $cacheAffinitMask = 0xFFFF$        ▷ Select any of the lane groups
5:
6:      else if $STHCM[ordered[j]] \rightarrow reqR == STHCM[ordered[j]] \rightarrow allocR$ then
7:          $reqR = STHCM[ordered[j]] \rightarrow reqR$
8:          $cacheAffinitMask = STHCM[ordered[j]] \rightarrow allocL$
9:      else
10:         if $STHCM[ordered[j]] \rightarrow reqR < STHCM[ordered[j]] \rightarrow allocR$ then
11:             $temp = STHCM[ordered[j]] \rightarrow allocL$        ▷ Already allocated issue width
12:             $configWords = configWords\ \&\sim (temp * 0xF)$
13:             $configWords = configWords | temp * 8$        ▷ clock gate the allocated lane groups
14:             $cacheAffinitMask = temp * 0xF$
15:             $reqR = STHCM[ordered[j]] \rightarrow reqR$
16:         end if
17:         if $STHCM[ordered[j]] \rightarrow reqR > STHCM[ordered[j]] \rightarrow allocR$ then
18:             $temp = STHCM[ordered[j]] \rightarrow allocL$
19:             $configWords = configWords\ \&\sim (temp * 0xF)$
20:             $configWords = configWords | temp * 8$
21:             $cacheAffinitMask = LanGroups * 0xF$
22:             $reqR = STHCM[ordered[j]] \rightarrow reqR$
23:         end if
24:     end if
25: end function
```

Scenario #2: When the number of requested lane groups is higher than the available lane groups, then the control layer may suspend some of the already running hardware contexts (that are executing lower priority tasks than the task that is requesting more resources) to get required resources. If an already running task requests for more resources and all of the following conditions are met, then some of the already running contexts may be suspended in order to meet the task requirements:

1. There are not enough lane groups available or lane groups can not be assigned due to lane groups assignment guidelines as discussed in Section 2.3.3.

2. The priority of the task which requires more resources is higher than at least one other running task.

If the control layer decides to suspend any of the contexts then it follows this procedure:

- Determines how many running tasks have lower priority than the task that is requesting more resources.

- Calculates how many contexts are needed to be suspended in order to satisfy the task's request.

- Suspends the contexts starting from the task context with the lowest priority (and move in ascending order).

Algorithm 3 depicts the suspension of the contexts in order to get resources for the higher priority task. In this algorithm, lines 2-13 cover the acquiring of resources by suspending the already running hardware contexts. This acquiring of lane groups from the other contexts continues until any of the following conditions are met:

- Lane groups are taken back from all the hardware contexts that are executing lower priority tasks than the priority of the resource requesting task. In the algorithm, line 2 implements this functionality.

- Required resources are acquired. Lines 10-12 implements this scenario, when available lane groups becomes equal to or greater than the requested ones then the loop to acquire resources from the other tasks is broken.

Algorithm 3 Calculations of cache affinity, when the available resources are less than the requested ones.

```
function CACHEAFFINITYFORINSUFFICIENTAVAILABLERESOUCES(void)
    for $i = hardwareContexts - 1$ to $i > j$ do
        if $ordered[i]! = unusedContext$ then
            $vLG = vLG|STHCM[ordered[i]] \rightarrow allocL$
            $STHCM[ordered[i]] \rightarrow allocL = ZeroLaneGroups$
            $vCores = vCores + STHCM[ordered[i]] \rightarrow reqR$
            $availCores = availCores + STHCM[ordered[i]] \rightarrow reqR$
            $STHCM[ordered[i]] \rightarrow allocR = ZeroResources$
        end if
        if $availCores >= STHCM[ordered[j]] \rightarrow reqR$ then
            $break$
        end if
    end for
    $reqR = STHCM[ordered[j]] \rightarrow reqR$
    $LanGroups = LanGroups + vLG$
    $configWords = configWords \ \& \sim (LanGroups * 0xF)$
    $configWords = configWords|LanGroups * 8$
    $cacheAffinitMask = LanGroups * 0xF$
end function
```

These suspended contexts are resumed when the following condition are met:

- Lane groups are available.
- The task which is assigned to the suspended context now has higher priority than already running tasks and the tasks in the ready queue.

Note: if there are more than one suspended contexts then they are resumed in the descending order of the priority of the tasks which are occupying these hardware contexts.

The above algorithm provides information to the rest of the control layer about requested resources and the cache affinity of the tasks by using *reqR* and *cacheAffinitMask* variables. The *reqR* variable contains the requested resources in term of issue-width of the task. Its value can be equal to the actual requested resources by the task but can also modified by the cache affinity calculation algorithm depending upon the available resources. The value of *reqR* variable is modified (according to the available issue width), so that lane group allocation mechanism of the control layer can just allocate resources based upon the cache affinity without thinking about whether the requested resources or available or not (as requested resources are already modified according to the available resources). The *cacheAffiniMask* contains the value that determines which of the lane groups can be assigned in order to maintain the cache affinity of the already running tasks.

Before, getting into the details about how lane groups are assigned to the hardware contexts, there are some data structures that need some attention. The data structures and how they are used while assigning lane groups to the hardware contexts are discussed below:

The first variable is *LanGroups*, it is already defined at the beginning of this chapter. The values and meaning of the values that can reside in this variable is explained below:

LanGroups	Meaning
0x1111	All lane groups are available
0x0011	1st and 2nd lane groups are available but 3rd and 4th are not available
0x1100	3rd and 4th lane groups are available but 1st and 2nd are not available
0x0001	1st lane group is available
0x0000	No lane group available

The next variable is *requestedLaneGroups*, it follows the same format as *LanGroups* variable. The values which resides in this variable represents the issue width requested by the task. The possible values of this variable and their meanings are explained below:

requestedLaneGroups	Meaning
0x1111	4 lane groups or 8 issue width is requested
0x0011	2 lane groups or 4 issue width is requested
0x0001	4 lane groups or 8 issue width is requested

The *config_mask* is the next variable that is used in the lane group allocation algorithms. This variable is used to configure *configWords* according to the allocated resources of the task.

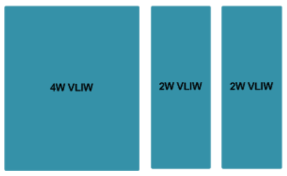
config_mask	Meaning
0x000F	1st lane group allocation mask
0x00FF	1st and 3rd lane groups allocation mask
0xFF00	3rd and 4th lane groups allocation mask
0xFFFF	All lane groups allocation mask

The assignment of the lane groups to the hardware contexts is further divided into two categories:

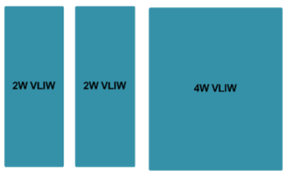
1. Assignment of the lane groups for the issue width 4.

2. Assignment of the lane groups for 2 or 8 issue width.

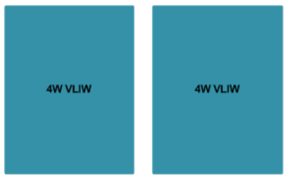
The possible configurations of the ρ-VEX, when any task is assigned with 4-issue width is portrayed in the Figure 4.4. The algorithm that is used in the control layer to assign 4-issue width is depicted with the Algorithm 4.



(a) ρ-VEX is configured as 4-issue 1-core and 2-issue 2-cores



(b) ρ-VEX is configured as 2-issue 2-cores and 4-issue 1-core



(c) ρ-VEX is configured as 4-issue 2-cores

Figure 4.4: Possible configurations of the ρ-VEX when 4-issue width is assigned to any task

Algorithm 4 Assignment of Lane groups for 4 Issue Width

```
 1: function LaneGroupAllocationfor4IssueWidth(void)
 2:     for $k = 0$ till $k < 2$ do
 3:         $temp\_result = requestedLaneGroups \,\&\, LaneGroups \,\&\, cacheAffinityMask$
 4:         if $temp\_result == requestedLaneGroups$ then
 5:             $resourceAcquired = TRUE$
 6:             $break$
 7:         else
 8:             $requestedLaneGRoups = requestedLaneGroups << 8$
 9:             $config\_mask = config\_mask << 8$
10:         end if
11:     end for
12:     if $resourceAcquired! = TRUE$ then
13:         $requestedLaneGroup = 0x00001$
14:         $config\_mask = 0xF$
15:         $reqR = 1$
16:     end if
17: end function
```

In this algorithm, *LaneGroups* variable contains all available lane groups, *requested-LaneGroups* contains the number of the requested lane groups and *cacheAffinityMask* variable does not allow to allocate those lane groups that can disturb the cache affinity of the other tasks.

The 4-issue width case is treated separately because the middle two lane groups cannot be merged together due to architectural constraints of the ρ-VEX. This constraint is explained in Section 2.3.3. The invalid configuration of the ρ-VEX due to this architectural constraint is also portrayed in Figure 4.5.

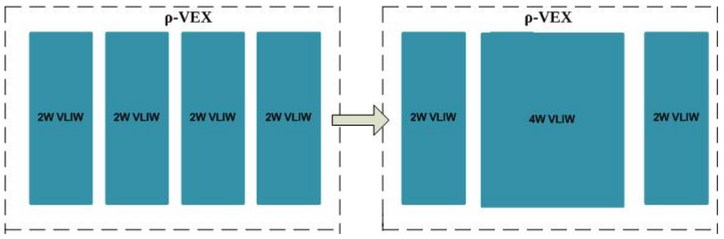


Figure 4.5: Invalid ρ-VEX configuration

In this algorithm, if requested lane groups cannot be assigned because of cache affinity or due the the ρ-VEX architectural constraints then task requirement is decreased to 2-issue width from the 4-issue width. The 12-16 lines set all the variables that are necessary to decrease the issue-width to 2. Once the issue width requirements are decreased then the lane group assignment is handled by Algorithm 5.

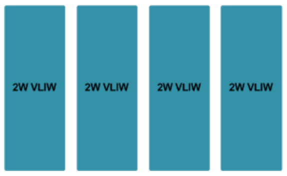
The assignment of lane groups for the cases of 2 or 8 issue width is implemented by using Algorithm 5. The sample configuration of the ρ-VEX for 2 and 8 issue width is also shown from Figure 4.6.

Algorithm 5 Assignment of Lane groups for 2 or 8 Issue Width

```
1: function LaneGroupAllocationfor2Or8IssueWidth(void)
2:     for $k = 0$ till $k < HardwareContexts$ do
3:         $temp\_result = requestedLaneGroups \; \& \; LaneGroups \; \& \; cacheAffinityMask$
4:         if $temp\_result == requestedLaneGroups$ then
5:             $resourceAcquired = TRUE$
6:             $break$
7:         else
8:             $requestedLaneGroups = requestedLaneGroups << 4$
9:             $config\_mask = config\_mask << 4$
10:        end if
11:    end for
12: end function
```

For issue width of 2, if any lane group is available and also comply with cache affinity requirements then it is assigned to the task. For the case of 8-ssue width, if all the lane groups are available only then they are assigned to requesting tasks.

The Algorithm 6 depicts how cache affinity Algorithms 2 and 3, and lane group assignment Algorithms 4 and 5 integrate together in order to achieve the proper lane group assignments with the preserved cache affinity.



(a) ρ-VEX configured as 2-issue 4-cores

(b) ρ-VEX is configured as 8-issue 1-core

Figure 4.6: Sample 2-issue and 8-issue configurations of the ρ-VEX

Algorithm 6 Lane Groups Assignment Algorithm

```
function ControlLayer(void)
    for $j = 0$ to $(HardwareContexts - 1)$ do
        $availCores = availeCores + STHCM[ordered[j]] \rightarrow allocR$
        $LanGroups = LanGroups | STHCM[ordered[j]] \rightarrow allocL$
        if $STHCM[ordered[j]] \rightarrow reqR <= availCores$ then
            cacheAffinityForSufficientAvailableResouces(void)
        else
            cacheAffinityForInSufficientAvailableResouces(void)
        end if
        if $reqR == 1$ then
            $requestedLaneGroup = 0x0001$
            $config\_mask = 0xF$
        else if $reqR == 2$ then
            $requestedLaneGroups = 0x0011$
            $config\_mask = 0xFF$
            $way4Config = TRUE$
        else if $reqR == 3$ then
            $requestedLaneGroups = 0x0011$
            $config\_mask = 0xFF$
            $way4config = TRUE$
            $reqR = 2$
        else if $reqR == 4$ then
            $requestedLaneGroup = 0x1111$
            $config\_mask = 0xFFFF$
        end if
        $temp = LanGroups$
        if $way4config == TRUE$ then
            LaneGroupAllocationfor4IssueWidth(void)
        end if
        if $way4config == FALSE$ then
            LaneGroupAllocationfor2Or8IssueWidth(void)
        end if
        $STHCM[ordered[j]] \rightarrow allocL = requestedLaneGroups$
        $STHCM[ordered[j]] \rightarrow allocR = reqR$
        $availCores = availCores - STHCM[ordered[j]] \rightarrow allocR$
        $LanGroups = LanGroups \oplus requestedLaneGroups$
        $temp\_config = requestedLaneGroups * ordered[j]$
        $configWords = configWords \ \& \ \sim config\_mask$
        $configWords = configWords + temp\_config$
    end for
end function
```

build 0.18

The basic philosophy that is used by Algorithm 6 for the control layer is that each lane group is assigned to the hardware contexts for one OS-tick. In the next tick the resource assignment is reviewed and this assignment of resources can change or remains the same depending upon resource requirements and cache affinity of the tasks. Each time, when Algorithm 6 executes, it goes through the hardware contexts in the descending order of priority of the tasks. So, the hardware contexts which executes the highest priority task is processed first. At first, if that context already has some resources then those allocated resources are added to the available resources as depicted from 3-4 line of this algorithm temporarily. Then available resources are compared with requested resources of the tasks. Depending upon whether requested resources are less than or greater than the available resources Algorithm 2 or 3 is called in order to calculate the cache affinity of the task. The results of the cache affinity are stored in *cacheAffinitMask* variable. After calculating the cache affinity, the values of the *mask* and *config_mask* variables set according to the resource requirements of the tasks. The combination of these two variables defines the number of the required lane groups. Here, it is worth noting that these two variables along with *cacheAffinitMask* variable determines assigned lane groups and position of those lane groups. After setting these variables, either Algorithm 4 or 5 is called. If the Algorithm 4 is unable to assign lane groups due to the architectural constraints of the Section 2.3.3, then the control layer decreases the resource requirement so that task can get some of the available resources. For example, if 4 issue width can not assigned due to the architectural constraints then control layer assign it 2 issue width. This assignment of the lane groups continue until control layer is done with all hardware contexts or no more resource available.

4.2.3 Possible Benefits from the Current Control Layer

1. If a new task comes in ready queue and has a higher priority than any of the already running tasks then the lowest priority task will get preempted. This will save higher priority tasks from the preemptions and penalty of context loading and storing.

2. The highest priority task will always keep its cache affinity. So the highest priority task will also has lowest number of cache misses.

3. As the priority of the any running task increases relative to the other running, it experiences less and less preemptions and less cache misses.

4.3 Standby Power Saving Mechanism

In order to save the power when there is nothing to process, a *Standby power saving mechanism* has been added to the FreeRTOS. This mechanism uses *Sleep and Wake-up System* which is already explained in Section 2.3.2. As *Sleep and Wake System* only applicable for the *context 0*, control layer and scheduler in the current implementation always execute on *context 0* and are called from the ISR. If there is nothing to process then control layer clock gates the whole processor in order to save the power. when the *context 0* is clock gated then the *scheduler* and the control layer execute under *sleep* *and wake system*, So the scheduling event happens on every tick. If it is determined that there is a task to process, the control layer wakes the corresponding portion of the ρ-VEX and if there is nothing to do it clock gates the processor again.

4.4 FreeRTOS and Control Layer Integration

In the current implementation, hardware contexts 0 acts as a master context. The control layer and FreeRTOS is actually run by by this hardware context.

This logical interaction of different hardware contexts is shown from the figure 4.7.

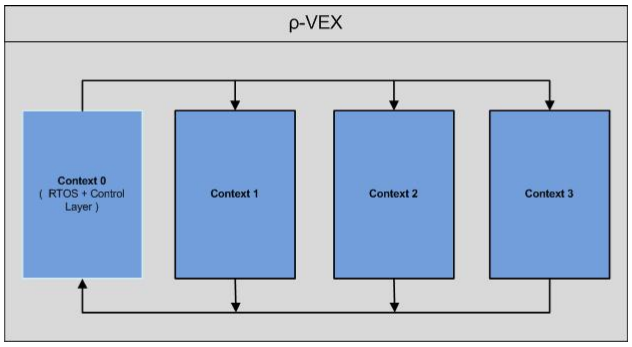


Figure 4.7: Logical Interaction of Different Hardware Contexts

The figure 4.7 depicts that control layer and RTOS are run by the context 0 and other contexts act as slave. These are called slaves because which tasks should be run by which hardware contexts is decided by the scheduler that runs on the hardware context 0. After this, the workload is distributed to the designated hardware contexts and then each hardware context processes the assigned tasks independent from each other. Other hardware contexts only come into the active state when tasks and lane groups are assigned to them by the context 0. After finishing the assigned task, each context releases its resources.

The Figure 4.8 depicts how the modified FreeRTOS and control layer are integrated in order to meet all the requirements which are discussed in Section 3.4.

At the startup all the tasks are created based upon the parameters given by the programmers and stored in the *ready* queue as depicted from the Figure 3.3. From the *ready queue* tasks are scheduled by the *RM scheduler* to the hardware contexts and then dispatcher assigns the lane groups to the hardware contexts based upon the requirements and priority of the assigned tasks to these hardware contexts. It is clear from the Figure 4.8, this scheduling and dispatching event happens on each interrupt of the *Timer ISR* , so the the frequency of dispatching and scheduling is control by *Timer ISR*. Once the lane groups are assigned to the hardware contexts then the dispatcher reconfigures the ρ-VEX accordingly. The hardware contexts, to whom lane groups are assigned are termed as active hardware contexts. The contexts of the tasks that are assigned to active hardware contexts is loaded into the respective hardware register with *Program State Storing and Loading* mechanism as explained in Section 2.3.1. After loading the task contexts, each hard context starts running the the task independent of each other. It may happens that

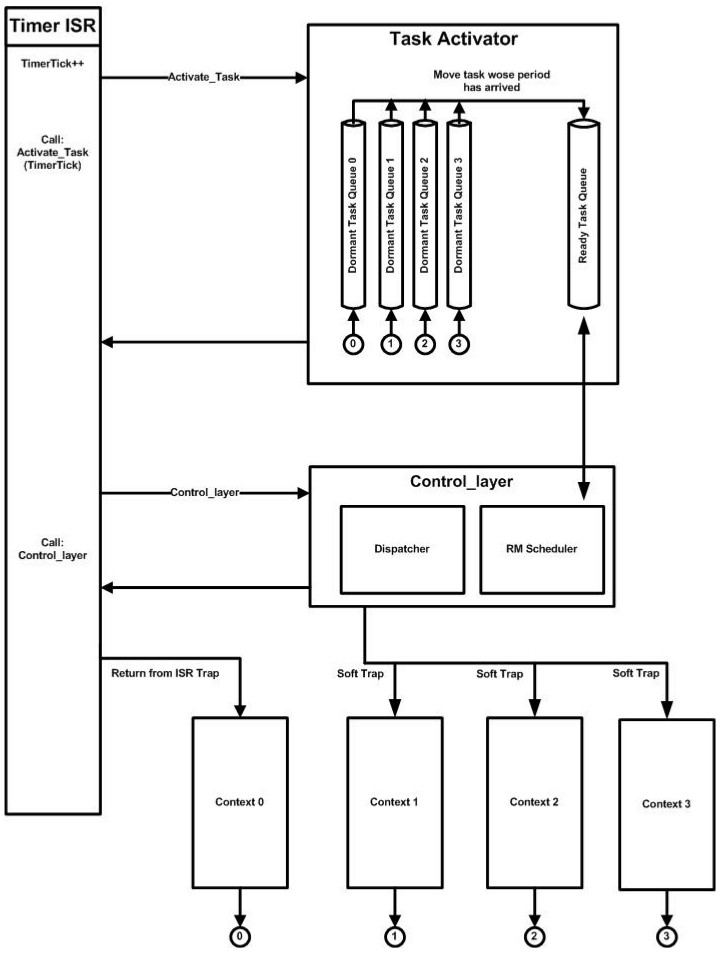


Figure 4.8: Control layer integrated with the FreeRTOS

another high priority task wants to run and no other hardware context is available for the assignment, then the scheduler removes the lowest priority task send back it to the ready queue and on its place assign the higher priority task that is needed to be run. This scenario is shown with the double headed arrow between *Ready Task queue* and the *Control_layer* in Figure 4.8. Once the task is finished on any hardware context then it is sent to respective *Dormant Task Queue.* For instance, the tasks which finishes their execution at the *hardware context 0* will be sent to *Dormant Task Queue 0* and which finishes at *hardware context 1* are sent to *Dormant Task Queue 1* and so on. In the dormant queues the task wait for their next period of the execution. On each timer tick, *Task Activator* is called from the *Timer ISR.* This *Task Activator* scans though all the *dormant queues* and tries to check, the execution period of which one of the tasks has been arrived. Task periods are actually multiple of the *Timer Ticks.* So *Task Activator* compares the timer tick number with with the period of the task. If both are equal in any *Dormant Task Queue* then it transfers that task into the *ready queue.* The tasks in the ready queue are also called active tasks.

4.5 Conclusion

All the modifications that were carried out to port FreeRTOS to the ρ-VEX were described in this chapter.

This chapter started with the description and purpose of the data structures that were used in the implementation and porting of FreeRTOS. After the description of the data structures, all the modifications were described. The first modification was about the *Task resuming* mechanism. In the original implementation of FreeRTOS, there was just one *dormant* queue and after finishing its execution the task used to go in that queue. In the new implementation, the number of *dormant* queues is equal to the number of hardware contexts, so that tasks do not have to wait for each other, if they finish their execution on different contexts at the same time. The second modification was about the implementation of the RM scheduler extended with the cyclic scheduler. This scheduler schedules the tasks according to the priority of the tasks, and the scheduling event happens at each timer-tick. Subsequently, the run-time reconfiguration request mechanism was described.

In the next section, the architecture and working of the control layer was described. The purpose of the control layer is to assign the issue width to the tasks based upon the task requirements and availability of the resources. In addition to the assignment of the lane groups, the control layer also gets back the freed resources of the tasks. While assigning the lane groups to the tasks, the control layer takes into consideration the priority of the tasks, the cache affinity of the tasks and the availability of lane groups. The control layer tries to assign resources to the task in a way that cache affinity of the already running tasks remains intact. The control layer assigns the resources to the tasks based on the best effort algorithm. This means that if the task requires mores resources but enough resources are not available then it assigns as many resources as are available. After the assignment of resources, the control layer reconfigures the ρ-VEX accordingly.

Subsequently, the *standby power saving* mechanism of the control layer was described. This control layer clock gates that part of the ρ-VEX that is not being used at the current moment to save the power. If there is nothing to do for the whole core then the control layer clock gates the entire core, then the scheduler and the control layer run under the *sleep and wake up* system.

The integration of the modified FreeRTOS and the control layer was described in the last part of this chapter. The control layer and the scheduler execute under the timer ISR. On each timer-tick, the *task-resuming* mechanism brings the task(s) from the *dormant* queue to the ready queue whose period has arrived. Then, based upon the priority of the tasks, the scheduler places them on the hardware contexts. Subsequently, based on priority and cache affinity, the control layer assigns the issue width and reconfigures the ρ-VEX accordingly. The FreeRTOS and the control layer always run on context 0.

5 FreeRTOS Testing and Evaluation

In chapter 4, all the modifications in FreeRTOS and implementation details were discussed. First, the modifications in the *task resuming mechanism* were given, then the implementation of the scheduler was described. Subsequently, the architecture and implementation of the *control layer* was discussed in detail. That chapter concluded with the discussion of integration of control layer with the FreeRTOS.

In this chapter testing and the evaluation of FreeRTOS are discussed. The goal of this chapter is to make sure that FreeRTOS works as intended and to measure its performance.

The testing and evaluation setup is described in the Section 5.1. Subsequently, functional testing of FreeRTOS is described. After the functional testing, evaluation of FreeRTOS is performed. In Section 5.3.1, the overheads due to FreeRTOS are measured. After that, the effect of OS-tick time at the task execution is discussed. This chapter concludes with the discussion of the impact of reconfigurability of ρ-VEX on the schedulability of real-time applications.

5.1 Evaluation and Testing Setup

The hardware platform used for the evaluation and testing of this thesis project is the Ml605 evaluation board [12]. It features a Virtex 6 FPGA (that is used to program our reconfigurable hardware) and a number of other input and output peripherals. Moreover, this hardware platform can also be programmed over Universal Synchronous Bus (USB) without needing an external programmer. This evaluation board is portrayed in Figure 5.1.

The ρ-VEX is coupled with Gaisler Research Libray (GRLIB) [17] by the students who already worked on the projects related to this processor. GRLIB Intellectual Property (IP) library is an integrated set of reusable IP cores, designed for System on Chip (SoC) projects. It is built around a common AMBA bus and this library was originally built by ARM.

The sample SOC built with GRLIB is portrayed in Figure 5.2.

In this diagram, the LEON3 behaves as a bus master but in our project ρ-VEX takes its place (as a bus master). The relevant components of GRLIB for our project are the Universal Asynchronous Receiver/Transmitter (UART), JTAG, Timer unit and interrupt controller. The UART is used to upload programs, and to observe output and status information. JTAG port is used for debugging. The interrupt controller of GRLIB merges interrupts from many different sources to the lines of ρ-VEX. The timer unit produces timer ticks which are used to run FreeRTOS. These ticks acts a time reference for FreeRTOS.

Figure 5.1: The ML605 development board [12]

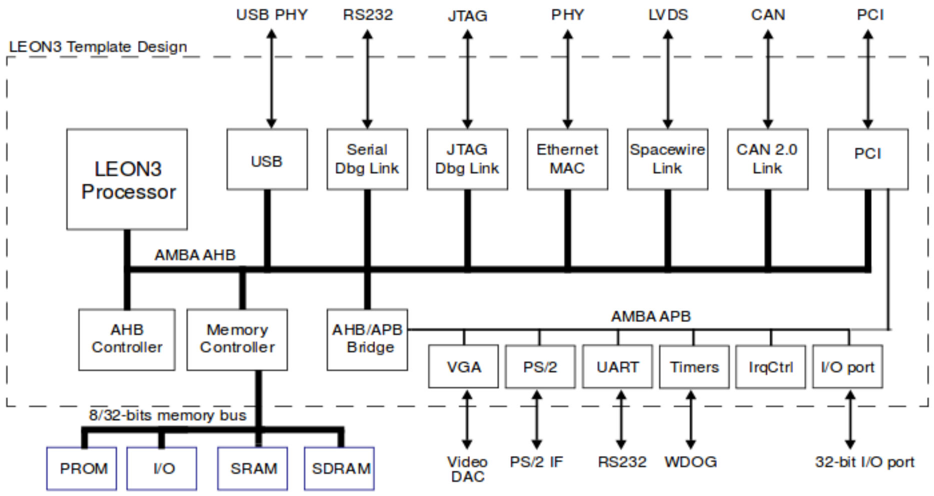


Figure 5.2: Example SoC built with GRLIB [17]

The ml605 evaluation board is connected to the computer which hosts Ubuntu 16.4. ρ-VEX Debugger (RVD) executes on this Linux distribution. This debugger is used to upload the programs to the ml605 evaluation board and to debug running programs on this board.

5.2 Functional Testing of ported FreeRTOS

The ported FreeRTOS is tested thoroughly and and its functionality is verified. Here, only testing of the major parts of the ported FreeRTOS is briefly described. This functional testing is done in three parts:

1. Scheduler testing
2. Control layer testing
3. Complete RTOS testing

In the current ported version of FreeRTOS, the scheduler and the control layer are decoupled from each other. This means that the scheduler assigns the tasks to the hardware contexts without considering which one of them will get processing resources first. The job of the scheduler is to place the task in hardware contexts with respect to the priority from the ready queue. Whereas, the control layer actually assigns processing resources to the tasks placed in the hardware contexts. The control layer first assigns the processing resources to the highest priority task among all the tasks placed in the hardware contexts. If the control layer still has the resources then it assigns resources to the hardware context which executes the next highest priority task. This process continues until all the hardware contexts are assigned with processing resources or the control layer is out of resources. If some hardware contexts are not assigned with processing resources, then they have to wait until the resources are released by the higher priority tasks.

As the scheduler and the control layer work independent of each other, their functionality can be independently tested.

5.2.1 Scheduler Testing

The RM scheduler is used to schedule tasks on the ρ-VEX, and there are 4 hardware contexts in the ρ-VEX that are treated as four virtual cores by the scheduler. This means that on each scheduling event the four highest priority tasks, among all the available tasks, are assigned to these virtual cores. The highest priority task is assigned first and then comes the the turn of second highest priority task and so on. If some higher priority task comes in the ready queue and no hardware context is available then the lowest of all the tasks that reside in the hardware contexts is removed and sent back to the ready queue.

In order to test that the current FreeRTOS version schedules tasks with respect to the RM scheduler, a task set of 8 tasks is created with every task having a distinct priority and almost the same work load. It is verified by the visual inspection of the output, the highest priority task starts its execution before the lower priority tasks. Hence, tasks are scheduled according to their priority which is characteristic of the RM scheduler.

In order to test that higher priority tasks preempt the lowest priority task among all the tasks assigned to hardware contexts, a scenario is created. In that scenario, all the hardware contexts are occupied by tasks and another higher priority task comes in the ready queue. From the visual inspection of the output it was verified that among all the tasks that are occupying the hardware contexts the lowest priority task is replaced with the higher priority one.

5.2.2 Control Layer Testing

The control layer assigns the processing resources to the hardware contexts based upon the processing requirements of the tasks. On each OS-tick, it checks for the resources that are needed by the tasks. Based upon the requirements, resources are assigned first to the highest priority task, then comes the turn of the second highest priority task and so on. The control layer functionality is tested by creating different scenarios.

The first property that is tested of the control layer that the hardware context with the highest priority task is given the processing resources first. It is verified by making required issue width of all the tasks that resides in the hardware contexts as 8. This means that all the processing resources would be occupied by one hardware context at a time. It is verified by the visual inspection that the highest priority task produced the output first, then output from the second highest priority task is produced and so on. It means that highest priority task was given the processing resources first, then came the turn of second highest priority task and so on.

Resources to the hardware contexts are assigned in the descending order of the priority of the tasks and if there are not enough resources available then the lower priority tasks have to wait until the availability of the resources. This characteristic is tested by making the resource requirements of the highest priority task as 4-issue width and rest of the tasks as 2-issue width. It is verified by the visual inspection and from the execution time of each task that indeed resources are assigned in the decreasing order of the priority. The following observations support this claim:

- Three tasks started running at the same time. This means that three out of four got the processing resources.
- Execution time of the task with the highest priority was less than other tasks even though all the tasks have almost the same workload.

From these observations, it can be concluded that highest priority task got more resources as compared to the other running task. (Therefore, it has less execution time as compared to the other.)

Now indeed we can say that highest priority task got the issue-width of four and rest of the two tasks got issue width of two each. The fourth hardware context did not execute because the control layer did not have any more resources. This is due to total issue-width (eight) of the ρ-VEX was distributed among the three hardware contexts.

5.2.3 Complete RTOS Testing

After testing the major components that are modified in FreeRTOS for porting to the ρ-VEX, the complete functionality of ported FreeRTOS is tested to see whether it works as intended or not. For that purpose, a set of 8 tasks is created with the same workload but different priorities, periods and resource requirements.

First the tasks are tested whether they execute according to their periods or not. This property is verified by measuring the number of ticks and the start time of the execution of each task. It turned out that each task started its execution after exactly the same number of ticks as the period of that task. So all tasks are executed according to their periods.

From the sequence of execution of the tasks it was verified that each task started its execution exactly according to the priority. The highest priority task started its execution before any other task and the lowest task priority started its execution at the last.

In order to test that resource allocation is done preferably according to the requirements of the tasks, the execution time of each task is measured. It is observed that tasks

Table 5.1: Total overhead of FreeRTOS on each OS-tick

Details of the overheads					
Issue Width	Context Store	Context Load	Scheduler + Control layer	Total Cycles	Total Time
	Processor cycles				milli sec
8	306	258	4549	5113	0.136
4	2139	2097	6512	10748	0.287
2	4739	4612	25897	35248	0.940

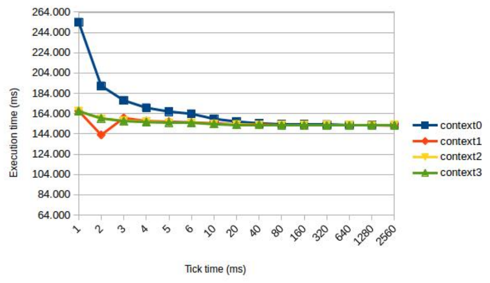
with more resource requirements complete their execution in shorter time as compare to the tasks with lower resource requirements even though each task has almost same execution time.

5.3 Evaluation of FreeRTOS

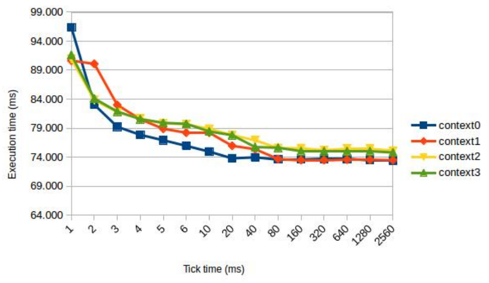
In Section 5.2, testing of the functionality of FreeRTOS was discussed. After making sure that FreeRTOS works as intended on the ρ-VEX, in this section its performance is evaluated. In order to evaluate the performance of FreeRTOS, different kind of experiments are carried out. The first experiment is about quantification of the overheads that are caused by this implementation of FreeRTOS. This quantification is very important because it helps us to evaluate the performance of our system. Subsequently, the impact of the OS-tick time on the execution time of the tasks on different hardware contexts for different configurations of ρ-VEX is measured. After that, the impact of reconfigurability of the ρ-VEX on the schedulability of real-time applications is measured.

5.3.1 Measurements of Overheads

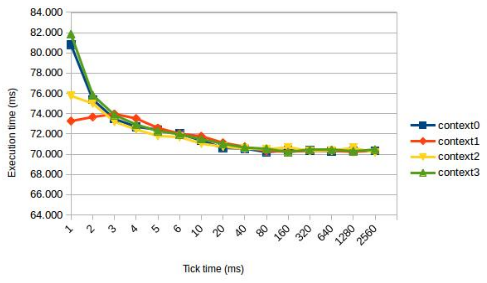
Although FreeRTOS and the control layer allow us to utilize the ρ-VEX resources efficiently, they also utilize resources of the processor. This utilization of resources, in addition to user applications, is the overhead on the processor. In order to measure the performance of the developed system, there is a need to quantify all the overheads incurred by the system. The control layer and the scheduler execute from the ISR on each tick. This execution of the control layer and the scheduler on each tick is the overhead. This overhead is incurred by the system to perform scheduling and reconfiguration at run time. The quantification of this overhead is absolutely necessary in order to measure the performance of FreeRTOS for different applications. The details of the overheads that are incurred by the system on each OS-tick are given in Table 5.1. The first column contains the issue widths of the ρ-VEX that are used in the measurements of the overheads. The second and third columns depict the actual time taken by the hardware to store the task context from the registers into the memory and to load task context from the memory into the registers respectively. The fourth column provides the scheduling and the control layer overhead. The last column gives the total overhead incurred by the system.



(a) Task execution times vs OS-tick Time period for 2 issue-width



(b) Task execution time vs OS-tick time period for 4 issue width



(c) Task execution time vs OS time period for 8-issue width

Figure 5.3: The effect of OS overhead on the execution of application tasks

5.3.2 Task Execution Time VS OS-Tick Time

As already discussed, each OS-tick causes overhead. Therefore, the frequency of the OS-tick (OS-tick time period) can effect the execution time of the tasks. The effect of OS-tick time on the execution time of a task for 4-hardware contexts with different issue-widths is portrayed in Figure 5.3.

The Figure 5.3a depicts task execution time with respect to the OS-tick time, when 2-issue width is assigned to each hardware context. It can be seen from this diagram that execution time of the task executing on the *context 0* is the highest when the OS-tick time is set to 1 ms. But this execution time starts decreasing as the OS-tick period increases, and after 80 ms, it becomes constant. While for the other hardware contexts execution time remains constant independent of OS-tick period. Moreover, it can be seen from this diagram that after 80ms the execution time of the task becomes equal and constant for all the hardware contexts.

The execution time varies on *context 0* but does not on any other context because the scheduler and the control layer executes on the context 0. On each OS-tick (on the *hardware context 0*) the context of the task is stored in the memory in order to run the control layer and the scheduler, and after its completion, task context is loaded again into the registers. This loading and storing of the contexts cause a lot of overhead. The details of this overhead is given in the Table 5.1. Hence, when OS-tick time period is small then the task has to suffer more overhead as compared to when OS-tick time is large. Therefore, the task execution is high when the OS-tick time period is small. It is interesting to note that OS-tick time does not have any effect on the tasks running on the other contexts. This is because tasks on those contexts runs independent of whatever is happening on *context 0*. Therefore, scheduling and control overhead does not have any effect on the tasks running on the contexts other than *context 0*. Moreover, task execution time becomes constant on all the hardware contexts after 80ms of the OS-tick time because then tasks complete their execution within one tick time, so overhead does not have any effect.

Figure 5.3b depicts task execution time with respect to the OS-tick time when 4-issue width is assigned to hardware contexts. If we compare this diagram with the Figure 5.3a, it can be immediately noticed that execution time is almost half in this diagram. Moreover, task execution time on *context 0* is still dependent on the OS-tick times but execution time variations are far less than in Figure 5.3a. This is due to the larger issue width, tasks complete their execution in less time, so the number of ticks that come within the execution also become less (with tick time). Therefore, now tasks have to bear less overhead and task execution time also varies less. From this diagram, another interesting thing can be noticed, although other contexts (excluding context 0) have lesser execution times as compared to Figure 5.3a but these execution times are not as constant as in 5.3a. Owing to the 4-issue width assigned to each hardware context, the control layer and the scheduler will also run on 4-issue width. So, some of the lanes might also be used by the scheduler and the control layer on each OS-tick which are actually used by the other hardware contexts. Consequently, on each OS-tick some part of the cache may be populated again by the scheduler and the control layer. So it may cause more cache misses for the other contexts, therefore, task execution also varies (with tick time) on the other hardware contexts (context1, context2 and context 3), when OS-tick time is small.

Figure 5.3c depicts task execution time with respect to the OS-tick time when 8-issue width is assigned to hardware contexts. It can be seen that task execution times are the smallest in this configuration as compared to the others but very much dependent upon the OS-tick time. Now all the contexts are assigned with 8-issue width, so when ever OS-tick happens the scheduler and the control layer will also executes at the issue width of 8. Therefore, as long as control layer and scheduler are running, tasks executing on the other context remains suspended. Because all the resources are now being used by the control layer and scheduler. This means overhead that happens on each OS-tick will effect the task running on every other context. So, execution time of task on each hardware get affected by the scheduling and control overhead. Therefore, execution of the task on each hardware context decreases with in the increase in the OS-tick time. The execution time of the task becomes constant after *80ms* due to the reason already discussed.

It can be concluded from this discussion that if we want task execution time to be predictable and independent of OS-tick period then we can use the same setup as we have used for the results of Figure 5.3a. This characteristic may be used to improve the response time of the system. In order to improve the response time of the system small OS-tick time is needed, so that the scheduling event can happen at a higher frequency. By using the setup for the results of figure 5.3a, we can run the scheduler and the control layer at the *context 0* at the highest possible frequency without effecting the execution time of the tasks running on the other contexts. Consequently, our system can respond to different events more quickly.

If want to complete execution as fast as possible with least concern about predictability of the tasks completion time then the setup for the results of the Figure5.3c may be preferably used.

5.3.3 Selection of Workload and Creation of Task Sets

As real robotic applications are not available at the moment, we have decided to use benchmark programs that can closely emulate behavior of these applications. For that purpose, we have chosen SNU Real-Time benchmark suite [30]. The programs in this benchmark are mostly numeric and Digital Signal Processing (DSP) algorithms, and this type of algorithms are mostly used in the autonomous robots [28].

All the programs in this benchmark suite are very small with respect to execution time. Therefore, in order to produce significant workload for our processor, each benchmark is executed in a loop for 300 times. After 300 iterations of each benchmark, Worst Case Execution Times (WCETs) are measured. These WCETs of different programs of this benchmark suite are shown in Table 5.2. These WCETs are measured while all the lane groups of ρ-VEX were fully utilized. This means in case of 2 issue width 4 tasks were running in parallel and for 8 issue width only one task was running on the ρ-VEX. These benchmarks are compiled for the ρ-VEX processor with -O3 optimizations. The second column in Table 5.2 shows the name of the benchmark. The third to fifth columns show three WCETs for 8, 4 and 2 ways respectively for the ρ-VEX. These worst case execution times also include the overhead of the scheduler and the control layer.

By using these tasks, numerous task sets consisting of 8 or 4 tasks each are generated. These task sets are used in the next section for the schedulability tests.

5.3.4 Schedulability of the Task Sets

A number of experiments are performed in order to measure the impact of the reconfigurability of the ρ-VEX on the the schedulability of task sets. In all these experiments OS-tick time is kept constant at 80ms to keep the scheduling overhead minimum as already discussed in Section 5.3.2. Three types of ρ-VEX based platforms with FreeRTOS are used in order to measure the impact of reconfigurability on real time applications. These three platforms include homogeneous, heterogeneous and run-time reconfigurable. The homogeneous platform comprises of three variants 2-issue 4-cores, 4-issue 2-cores and 8-issue 1-core. The heterogeneous platform consists of 2-issue 2-cores and 4-issue width 1-core. The heterogeneous platform has following characteristics:

- At every scheduling event the highest priority task in the ready queue is assigned to the 4-issue width core.
- In this platform, all lane groups are fixed with the hardware contexts. In the current implementation of the heterogeneous platform, 4 issue width is assigned to *hardware context 3*, and *hardware context 2* and *hardware context 1* are assigned with 2-issue width. The scheduler and the control layer executes under the *sleep and wake up system* on *hardware context 0*.

Table 5.2: Benchmark (WCETs in ms at 37.5 MHz)

No	Tasks	WCET8	WCET4	WCT2
1	adpcm-test.c	67350.797	70725.046	29094.825
2	bs.c	1.050	1.314	2.723
3	crc.c	20.667	21.341	40.893
4	fft1.c	144.638	144.149	430.863
5	fibcall.c	0.667	0.820	1.511
6	fir.c	1956.343	2022.109	4862.375
7	Insertsort.c	13.866	14.329	58.425
8	Jfdctint.c	27.249	28.852	109.294
9	lms.c	17846.312	28033.034	52543.279
10	select.c	16.194	16.660	67.196
11	sqrt.c	43.654	44.055	144.346
12	matmul.c	38.596	61.648	160.311
13	minver.c	35.261	50.852	139.090
14	qsort-exam.c	32.167	33.909	106.664
15	qurt.c	53.579	54.596	360.740

- Once a task is assigned to a hardware context, then it cannot migrate to the other hardware contexts before completing its execution. For example, if a task needs 4-issue width but at the scheduling event it is assigned to the 2-issue width hardware context owing to the lower priority. Then that task will stay there until its completion, even though 4-issue width core is available. This is due to the fact that in this platform lane groups are fixed with hardware contexts. Then in order to get more resources the task should migrate to the other hardware context, which involve the task loading and storing mechanism. Consequently, it involves a number of memory accesses which may kill all the benefits of moving to the higher issue width.

The run-time reconfigurable version is completely flexible. In this platform, lane groups are not assigned to any particular hardware context. Rather, they are assigned by the control layer at run time based upon the processing requirements of the tasks. In this configuration, if resources are available and a task requires more, then more resources are assigned to it. The task does not need to migrate to the other hardware context. In that case, at maximum tasks have to bear a small penalty in terms of cache misses (to switch to more issue width).

For the run-time reconfigurable platform, task requirements for different issue widths are assigned at compile time. This due to the reason that same benchmark is executed a number of times in one task as discussed in Section 5.3.3. Hence, processing requirements of the tasks will remain the same thorough out their execution. The worst case execution times for different configurations are already given in the Section 5.3.3. In order to assign the configuration the following criteria are used:

- Among all the benchmarks, whichever have the highest the WCET is assigned 8-issue width.

- The benchmarks which have very small difference or equal WCETs in case of 8-Issue width and 4-issue width are assigned with 4-issue width configuration.
- The benchmarks which have execution time less than or around one tick(80ms) are assigned with 2-issue width. So that for these benchmark programs TLP can be exploited at maximum.

Experiments are performed in order to evaluate the impact of reconfigurability on the schedulability of the real-time applications. Therefore, each benchmark program needs a period, priority and a deadline to execute as real-time application on FreeRTOS. Moreover, all the ρ-VEX based platforms (mentioned above) have same number of resources, therefore, for the fair comparison between different platforms, processor utilization must be the same. Hence, for all platforms the same periods, priorities and deadlines are assigned to each task in a task-set.

For all these experiments 10 task sets are selected and each consist of 4 benchmarks selected randomly from SNU benchmark suite [30].

For the experiments, each task in a task set is assigned with a unique number that is referred to as its priority. For the assignments of the periods to the tasks, randomly very high numbers are chosen. These numbers are chosen in way that every task set should be schedulable on every platform at this period. The deadline of each task is kept equal to its period.

The experiments are performed starting from a very high period. This period is chosen as already mentioned in a way that the task set is schedulable at that period. Then start decreasing the period of the tasks until that task-set becomes un-schedulable for that platform. Then the same experiments are repeated again on the next platform.

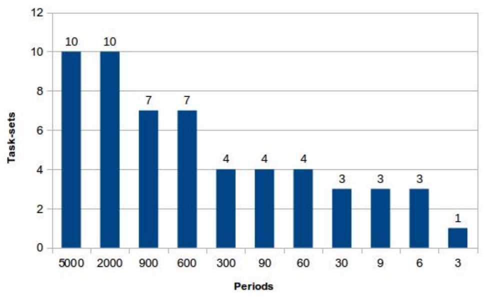
In this way experiments are performed for each task set and on each platform.

The schedulable task-sets for each ρ-VEX based platform for different periods are depicted in Figure 5.4.

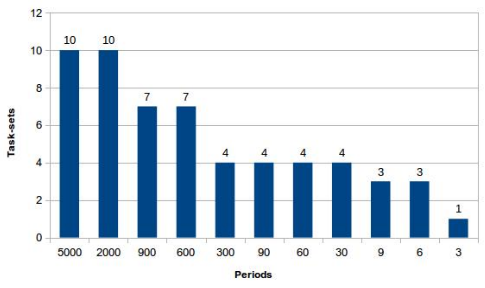
In each figure the height of bar for each period represents the number of successfully schedulable task-sets for that period.

The total number of successfully schedulable task sets for all the periods on each ρ-VEX based platform is also portrayed in Figure 5.5. It can be seen from this diagram that the lowest number of task are schedulable on 2-issue width and 2-2-4 heterogeneous platforms. In case of 2-issue width 4-cores, the *ILP* of the tasks could not be fully exploited and as there are four hardware threads running in parallel, therefore, there are more contentions at the common bus. For the heterogeneous configuration, lane groups are not assigned according to the requirements of the tasks. Some of the tasks are allocated with more resources and some of the them get less resources. So, the resources on the heterogeneous platform are not efficiently utilized. Due to these reasons the least number of task-sets are schedulable on these two platforms.

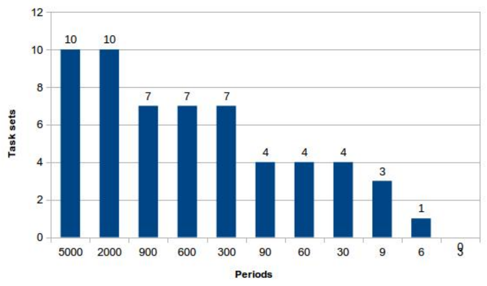
4-issue width 2-cores and 8 issue width 1-core homogeneous platforms perform better than the 2-issue width and heterogeneous configuration, and (moreover) both platforms are able to schedule same number of task sets. For the 8-issue width 1-core, every task is allocated with all the resources of the core. Therefore, resources are over allocated for the task execution, but bus contention is minimum. For 4-issue width, although processing resources are given to each task at a moderate level, there are also some bus



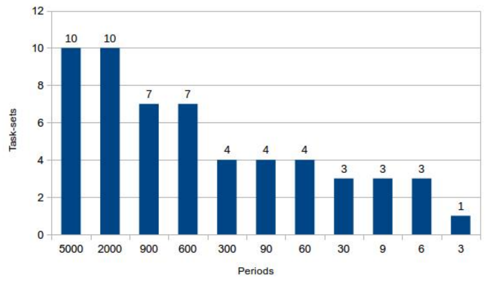
(a) Schedulable task sets for homogeneous 2-issue 4-cores



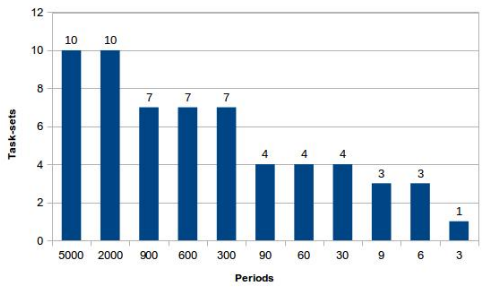
(b) Schedulable task sets for homogeneous 4-issue 2-cores



(c) Schedulable task sets for 8-issue 1 core



(d) Schedulable task sets for the heterogeneous platform (2-2-4)



(e) Schedulable task sets for the run-time reconfigurable platform

Figure 5.4: The summary of the schedulability test for all the configurations

contentions due to two threads running in parallel. Due to these reasons, these two platforms perform equally in term of task schedulability.

It can be seen from Figure 5.5 that the run-time reconfigurable platform performs better than any other. It is because in that configuration resources are utilized efficiently and according to the requirements of the tasks. During the execution of the task-sets, at one instant of the time *ILP* is exploited by giving all the resources and on the other moment *TLP* is exploited by running more tasks in parallel. As this platform assigns resources according to the requirements of the tasks, this platform is able to schedule more task sets as compared to any other platform under the same conditions of task priorities, periods and deadlines.

Hence, we can say that reconfigurability of the ρ-VEX indeed improves the schedu-

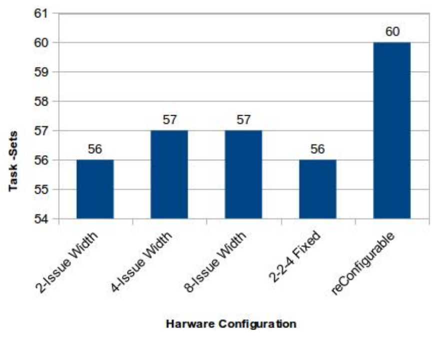


Figure 5.5: Total number of schedulable task sets for each configuration

lablity of real time applications.

5.4 Conclusion

Functional testing and evaluation of the modified FreeRTOS were described in this chapter.

This chapter started with the description of the test setup and the tools that were used in testing and evaluation of FreeRTOS. Subsequently, testing of FreeRTOS was described briefly and major focus was given to the *scheduler* and the control layer. At the end of the section, testing of the complete FreeRTOS was discussed.

After the testing of FreeRTOS, its evaluation was performed. The evaluation section started with the description of overheads that were incurred by the ρ-VEX due to the modified FreeRTOS. In the current implementation the overheads are produced on each OS-tick, so the effect of the frequency of OS-tick on the execution of a task was described next. The impact of OS-tick on the execution time of the task was evaluated for all the configurations of the ρ-VEX. Subsequently, the impact of reconfigurability of the ρ-VEX at the schedulability of real-time robotic applications was discussed. In order to measure this impact, first the selection of suitable real-time robotic applications was discussed and a number of real-time task sets were created from those applications. For the performance comparison, three ρ-VEX based platforms (homogeneous, heterogeneous and run-time reconfigurable) were discussed. These platform had same resources but different hardware configurations with respect to processing resources. A lot of experiments were performed with the real-time task sets on the three ρ-VEX based platforms. From the results of these experiments, it was concluded that indeed the run-time reconfigurability of the ρ-VEX improves the schedulability of the real-time tasks.

6 Conclusion and Future Work

This chapter summarizes and concludes this thesis project.

Section 6.1, summarizes the chapters of this thesis. In Section 6.2, the main contributions of this project will be listed. Finally in Section 6.4, recommendations for future work will be discussed.

6.1 Summary

In the Chapter 2, background information of this thesis project was presented. In the first section, FPGAs were introduced, and the working and advantages of this technology were briefly discussed. In addition, it was explained how processors, designated as softcores, can be implemented in this technology. Subsequently, VLIW processors were introduced and their advantages were discussed.

This background established the basis for the introduction of the ρ-VEX. The ρ-VEX is a VLIW softcore processor, implemented on FPGA reconfigurable fabric. Being a VLIW processor, the ρ-VEX issues instructions in the bundles of two or more. This increases the IPC and, thereby, improves the performance of applications. This processor is design time configurable and run-time reconfigurable. This run-time reconfiguration feature can be used to exploit TLP and ILP of the programs. After this short discussion about ρ-VEX in general, those parts of the ρ-VEX were discussed in detail that were related to porting of an OS. These parts include the *run-time reconfiguration* mechanism, the *context loading and restoring* mechanism, and the *sleep and wake-up* system.

In the second part of this chapter, first the concept of an OS was given and different classes of an OS were introduced. Moreover, different factors which derive those classes were also discussed. In that discussion, more focus was given to Real Time Operating Systems (RTOS). After that scheduling polices that are usually employed by the RTOS in a uni-processor and multi-processor environment were given. Moreover, properties, and pros and cons of the different scheduling polices were discussed.

This chapter finished with the discussion of scheduling objectives of real time control systems. This discussion provided us the goals that we wanted to achieve by selecting a specific scheduling policy.

Chapter 3 started with the discussion of basic concepts of the RTOS. In the first section of this chapter, RTOS jargon was introduced and the architecture of typical RTOS was presented. The architecture of the typical RTOS consists of three layers: application layer, kernel layer and hardware specific layer. In the application layer, user tasks resides and, moreover, task creation and termination is also initiated from this layer. The control layer performs management of all the resources of the system and hardware specific layer interacts with the underlying hardware. Moreover, how an application task traverses through different states during its life-time was also explained in this section.

Subsequently, there were a number of different types of RTOS that were considered and selection criteria for the best candidate for porting to the ρ-VEX was established. All those potential candidates were compared with each other based upon the selection criteria. From this selection procedure, FreeRTOS came out to be a clear winner. Therefore, FreeRTOS was selected as an RTOS that would be ported in this project to the ρ-VEX.

In the next section, functional and non-function requirements of the ρ-VEX ported version of FreeRTOS were given. These requirements were elicited from the project description and from the discussions within the CE group of TU Delft.

Subsequently, from the architecture of the FreeRTOS and requirements of the ρ-VEX ported version of FreeRTOS, the challenges in porting the FreeRTOS were described. In the last section of this chapter, the procedure for the selection of the task scheduling policy on ρ-VEX was described. As the ρ-VEX has four virtual cores, the task scheduling problem like in other muticore-core processors was subdivided into (1) task allocation and (2) task scheduling on the individual processors. For this thesis project, the global task allocation scheme was selected due to the run-time reconfigurability of ρ-VEX, and RM scheduler was selected in order to schedule tasks on the ρ-VEX. The RM scheduler was selected because of its predictable response to every type and condition of workload. Moreover, in order to make the RM scheduler efficient, it was enhanced with the clock driven scheduler.

In Chapter 4, all the work was described that was carried out to port FreeRTOS to the ρ-VEX. This chapter started with the description and purpose of the data structures that were used in the implementation and porting of FreeRTOS. After the description of the data structures, all the modifications were described. The first modification was about the *Task resuming* mechanism. In the original implementation of FreeRTOS, there was just one *dormant* queue and after finishing its execution the task used to go in that queue. In the new implementation, the number of *dormant* queues is equal to the number of hardware contexts, so that tasks do not have to wait for each other, if they finish their execution on different contexts at the same time. The second modification was about the implementation of the RM scheduler extended with the cyclic scheduler. This scheduler schedules the tasks according to the priority of the tasks, and the scheduling event happens at each timer-tick. Subsequently, the run-time reconfiguration request mechanism was described.

In the next section, the architecture and working of the control layer was described. The purpose of the control layer is to assign the issue width to the tasks based upon the task requirements and availability of the resources. In addition to the assignment of the lane groups, the control layer also gets back the freed resources of the tasks. While assigning the lane groups to the tasks, the control layer takes into consideration the priority of the tasks, the cache affinity of the tasks and the availability of lane groups. The control layer tries to assign resources to the task in a way that cache affinity of the already running tasks remains intact. The control layer assigns the resources to the tasks based on the best effort algorithm. This means that if the task requires mores resources but enough resources are not available then it assigns as many resources as are available.

After the assignment of resources, the control layer reconfigures the ρ-VEX accordingly.

Subsequently, the *standby power saving* mechanism of the control layer was described. This control layer clock gates that part of the ρ-VEX that is not being used at the current moment to save the power. If there is nothing to do for the whole core then the control layer clock gates the entire core, then the scheduler and the control layer run under the *sleep and wake up* system.

The integration of the modified FreeRTOS and the control layer was described in the last part of this chapter. The control layer and the scheduler executes under the timer ISR. On each timer-tick, the *task-resuming* mechanism brings the task(s) from the *dormant* queue to the ready queue whose period has arrived. Then, based upon the priority of the tasks, the scheduler places them on the hardware contexts. Subsequently, based on priority and cache affinity, the control layer assigns the issue width and reconfigures the ρ-VEX accordingly. The FreeRTOS and the control layer always run on context 0.

Chapter 5 describes functional testing and evaluation of the modified FreeRTOS. This chapter started with the description of the test setup and the tools that were used in testing and evaluation of FreeRTOS. Subsequently, testing of FreeRTOS was described briefly and major focus was given to the *scheduler* and the control layer. At the end of the section, testing of the complete FreeRTOS was discussed.

After the testing of FreeRTOS, its evaluation was performed. The evaluation section started with the description of overheads that were incurred by the ρ-VEX due to the modified FreeRTOS. In the current implementation the overheads are produced on each OS-tick, so the effect of the frequency of OS-tick on the execution of a task was described next. The impact of OS-tick on the execution time of the task was evaluated for all the configurations of the ρ-VEX. Subsequently, the impact of reconfigurability of the ρ-VEX at the schedulability of real-time robotic applications was discussed. In order to measure this impact, first the selection of suitable real-time robotic applications was discussed and a number of real-time task sets were created from those applications. For the performance comparison, three ρ-VEX based platforms (homogeneous, heterogeneous and run-time reconfigurable) were discussed. These platform had same resources but different hardware configuration (with respect to processing resources). A lot of experiments were performed with the real-time task sets on the three ρ-VEX based platforms. From the results of these experiments, it was concluded that indeed the run-time reconfigurability of the ρ-VEX improves the schedulability of the real-time tasks.

6.2 Main contribution

The problem statement of this thesis project was:

How to design and implement RTOS and control layer for the ρ-VEX in order to (best) meet the processing requirements of the real-time applications ?

To answer this question, the following three goals were established:

1. Designing and implementing the control layer and porting of FreeRTOS for the ρ-VEX.

2. Making sure that the modified FreeRTOS and the control layers work as intended.
3. Measuring the performance of FreeRTOS and evaluating the impact of the reconfigurability of the ρ-VEX on the schedulability of the real-time tasks.

To reach the first goal, a control layer is designed that is able to reconfigure the ρ-VEX at run-time. The control layer calculates the current configuration of this processor based upon the available lane groups, resource requirements, priorities and cache affinity of the tasks. After the calculation of the current configuration of the ρ-VEX, the control layer reconfigures this processor at run-time by writing into the *CRR* register. In order to port FreeRTOS to the ρ-VEX, a number of modifications are done in the existing version of FreeRTOS. These modifications include new design of *task-dispatcher*, modifications in the *task-resuming* mechanism, implementation of the *RM scheduler*, and integration of the control layer and modified FreeRTOS. The *task-dispatcher* loads the tasks from the memory into the registers of the different hardware contexts of the ρ-VEX by employing *program state storing and loading* mechanism. The *task-resuming* mechanism moves the tasks from the *dormant* queues to the *ready* queue whose period has arrived. The *RM scheduler* schedules the tasks and places them in the hardware contexts based upon their priorities. Subsequently, the control layer and FreeRTOS are integrated together, so that the control layer can assign the resources to the tasks that are managed by the FreeRTOS.

To meet the second goal, this system is tested in three phases. At first the scheduler is tested. It is tested that the highest priority task is always assigned to the processor first. Additionally, it is tested that if a higher priority task comes in the ready queue then the lowest priority task among all the running tasks is preempted and sent back to the ready queue. After that, the control layer is tested. It is tested that the control layer assigns the resources according to the priority and resource requirements of the tasks. The highest priority task is given the resources first, then comes the turn of the lower priority task. Subsequently, the complete functionality of the modified FreeRTOS is tested. In this phase a number of characteristics is tested. It is tested that the modified FreeRTOS and the control layer works seamlessly with each other. The tasks are scheduled according to their periods and priorities. It is also tested that if there is nothing to process then processor is set to sleep and it is waken-up if something comes up for the processing.

To reach the third goal, first the overheads of this implementation are measured. As these overheads happen on each tick of the OS, the effect of the frequency of the OS tick at the execution time of the tasks is measured for all the hardware contexts and with every issue-width. Subsequently, the impact of reconfigurability of the ρ-VEX on the schedulability of the real-time tasks is evaluated. In order to evaluate the impact of reconfigurability, three ρ-VEX based platforms (homogeneous, heterogeneous and run-time reconfigurable) are created. After that SNU benchmark suite is selected because it closely matches the behavior of robotic applications. From this benchmark suite a number of real-time task sets are generated by assigning suitable priorities, periods, deadlines and resource requirements. Moreover, deadline of each task is kept equal to the period of the task. The experiments are performed on all the hardware platforms for each task set with the different periods. After a lot of experiments, the number of the task sets scheduled by each platform are compared and it came out that run-time reconfigurable platform was able to schedule the highest number of task sets. So, the run-time reconfigurability of the ρ-VEX improves the schedulability of real-time tasks.

6.3 Additional Work

In the course of this project, some time has been spent on additional work related to the ρ-VEX processor. This additional work includes the development of two versions of demos that were prepared to show the capabilities of this processor to the management of ALMRAVI project. The first version of this demo focuses on following characteristics of this processor:

- Changes in issue width can be done at the run-time.
- Multiple virtual cores with the different issue widths can run concurrently and independently of each other.
- Large issue width can execute programs at higher speed.

The second version of the demo exhibits power saving features of this processor along with all the features of the first demo. The new version of this demo showcases the following features (in addition to the features of the first verion of the demo):

- The lane groups of the ρ-VEX can be shutdown individually.
- On shutdown of any lane group, the power consumption of ρ-VEX decreases.
- If there is nothing to process, then the whole processor can be put to sleep under the *sleep and wake-up* system.
- On the arrival of any (to be processed) task, the *sleep and wake-up* system wakes-up the required part of the processor.

6.4 Future Work

This is the first project that is aimed at the exploiting the run-time reconfigurability of ρ-VEX. Therefore, in the implementation (for the simplicity) all the tasks are assumed to be independent of each other and they do not share any resources. But in the real life applications, this assumption does not hold anymore. So, there is a need to implement semaphores and locks to meet the task requirements that are dependent on each other and share hardware resources.

Moreover, in the current implementation resource reconfigurations requests of the tasks are fulfilled only on the timer ticks. In order to better exploit the reconfigurability of this processor, there is need to develop a mechanism that can carry out reconfiguration requests as soon as requested by the running tasks without waiting for the next tick.

Bibliography

[1] On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[2] Kaveh Aasaraai and Andreas Moshovos. Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming. In *FPL*, pages 79–85. IEEE, 2009.

[3] Chirag Dawar Ajay Kumar, Bharat Kumar. Classification of operating system. *INTERNATIONAL JOURNAL FOR RESEARCH IN APPLIED SCIENCE AND ENGINEERING TECHNOLOGY (IJRASET)*, 2(IX), sep 2014.

[4] F. Anjam, M. Nadeem, and S. Wong. Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[5] F. Anjam, S. Wong, L. Carro, G. L. Nazar, and M. B. Rutzig. Simultaneous reconfiguration of issue-width and instruction cache for a vliw processor. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 183–192, July 2012.

[6] Anthony Brandon and Stephan Wong. Support for dynamic issue width in vliw processors using generic binaries. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 827–832. EDA Consortium, 2013.

[7] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and Th. Ungerer. Interrupt service threads - a new approach to handle multiple hard real-time events on a multithreaded microcontroller. In *RTSS WIP sessions*, pages 11–15, 1999.

[8] V. Brost, F. Yang, and M. Paindavoine. A modular vliw processor. In *2007 IEEE International Symposium on Circuits and Systems*, pages 3968–3971, May 2007.

[9] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Publishing Company, Incorporated, 3rd edition, 2011.

[10] Jarad Cannon, Kevin Rose, and Wheeler Ruml. Real-time motion planning with dynamic obstacles. In *SOCS*, 2012.

[11] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS.* Chapman Hall/CRC, Boca, 2004.

[12] DVI CODEC. Ml605 evaluation board. *ML605 Hardware User Guide*, 2009.

[13] Swapan Debbarma and Kunal Chakma. An approach to advance real-time os services with soft-error. *International Journal of Computer and Electrical Engineering*, 3(4):548, 2011.

[14] J.B. Dixit. *Fundamentals of Computer Programming and Information Technology.* Laxmi Publications Pvt Limited, 2005.

[15] B.P. Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns.* Doing hard time : developing real-time systems with UML, objects, frameworks and patterns / Bruce Powel Douglass. Addison-Wesley, 1999.

[16] B.S. El-Haik and A. Shaout. *Software Design for Six Sigma: A Roadmap for Excellence.* Wiley, 2011.

[17] Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, and Sandi Habinc. Grlib ip core users manual. *Gaisler research*, 2007.

[18] Wolfgang A. Halang, Roman Gumzej, Matjaz Colnaric, and Marjan Druzovec. Measuring the performance of real-time systems. *Real-Time Syst.*, 18(1):59–68, January 2000.

[19] Christian Iseli and Eduardo Sanchez. Spyder: A reconfigurable vliw processor using fpgas. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 17–24. IEEE, 1993.

[20] Jens Johansen. *Implementing Virtual Address Hardware Support on the ρ-VEX Platform.* PhD thesis, TU Delft, Delft University of Technology, 2016.

[21] Alex K Jones, Raymond Hoare, Dara Kusic, Joshua Fazekas, and John Foster. An fpga-based vliw processor with custom hardware execution. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 107–117. ACM, 2005.

[22] L. Kleinrock. *Queueing Systems: Volume 2: Computer Applications.* John Wiley & Sons New York, 1976.

[23] P.A. Laplante. *Real-Time Systems Design and Analysis.* Wiley, 2004.

[24] Andrea Lodi, Mario Toma, Fabio Campi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri. A vliw processor with reconfigurable instruction set for embedded applications. *IEEE Journal of solid-state circuits*, 38(11):1876–1886, 2003.

[25] R. Mall. *Real-Time Systems: Theory and Practice.* Pearson Education, 2009.

[26] James Mistry. *FreeRTOS and multicore.* PhD thesis, University of York, 2011.

[27] John Paul Shen and Mikko H. Lipasti. *Modern processor design : fundamentals of superscalar processors.* McGraw-Hill Higher Education, Boston, 2005. Index.

[28] Karun B Shimoga. Robot grasp synthesis algorithms: A survey. *The International Journal of Robotics Research*, 15(3):230–266, 1996.

[29] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* Wiley Publishing, 8th edition, 2008.

[30] SNU. SNU Real-Time Benchmarks. `https://web.archive.org/web/20160711085844/http://www.cprover.org/goto-cc/examples/snu.html`, 2016. [Online; accessed 11-July-2016].

[31] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* The Springer International Series in Engineering and Computer Science. Springer US, 2012.

[32] TechKnowlege. Time Sharing Operating System. `https://web.archive.org/web/20160720132616/http://techknowledge9945.blogspot.nl/2015/11/time-sharing-operating-system.html`, 2016. [Online; accessed 20-July-2016].

[33] J van Straten. A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support. Master's thesis, Delft University of Technology, the Netherlands, 2016.

[34] S. Wikipedia and LLC Books. *Time-Sharing Operating Systems: Time-Sharing, Multics, Michigan Terminal System, Burroughs Mcp, Pick Operating System, Cpcms, Ibm Cp-40.* General Books, 2010.

[35] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[36] Stephan Wong, Thijs Van As, and Geoffrey Brown. ρ-vex: A reconfigurable and extensible softcore vliw processor. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 369–372. IEEE, 2008.

[37] Z. Wu, M. Guo, C. Chen, and J. Bu. *Embedded Software and Systems: First International Conference, ICESS 2004, Hangzhou, China, December 9-10, 2004, Revised Selected Papers.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005.