Software Partitioning and Synthetic Load Generation Framework for Multicore in Industrial Control Systems

Authors: Aseem Rastogi
Rizwin Shooja
Abstract

With the current trend in migrating legacy real-time systems to multicore processors, the problem of partitioning the tasks to cores is of importance in order to guarantee reliable and deterministic behavior. This thesis tries to improve the existing partitioning algorithms as well as develop novel solutions to the partitioning problem.

The thesis presents a way to combine multiple partitioning strategies using a modified version of an existing cost based partitioning algorithm. The blocking time cost function is modified and a novel Parallelization cost function is implemented which minimizes the response times of the real time application.

Furthermore, a synthetic load generation framework is developed to test the partitioning algorithms which creates the tasks, executes them and measures the real time characteristics. The test framework is run with the developed partitioning algorithms and the results of combining the parallelization and blocking time optimization strategies results in a partition which is the best compromise in terms of minimizing the blocking time and the response time.
Preface

First of all, we would like to thank our supervisors at the ABB Corporate Research, Kristian Sandström and Hongyu Pei Breivold for giving us this opportunity to be a part of this exciting project. Dr. Sandström was always there to guide us through the various phases of the project and giving guidance at the right time.

We are grateful to our supervisor at university, Hamid Reza Faragardi, and examiner, Prof. Thomas Nolte for giving this opportunity to pursue this interesting topic as our Master’s thesis. Also, we thank the Head of the Embedded Systems division, Moris Behnam for his moral support and along with Adnan Causevic, the co-ordinator of the Master’s program in Intelligent Embedded System, for administrative supports during the project.

Furthermore, we extend our sincere gratitude to our colleague Cezar Chiru for his valuable suggestions in the project.

We also appreciate our parents and friends who supported us morally for pursuing this thesis.

Last but not the least; we thank our colleagues at ABB for wonderful moments that we shared there.
Nomenclature

Abbreviations
AMP – Asymmetric Multi-Processing
API – Application Peripheral Interface
CPU – Central Processing Unit
FPGA – Field Programmable Gate Array
GNU – GNU’s Not Unix
ICC – Intel C++ Compiler
ICS – Industrial Control Systems
IPC – Inter-Process Communication
LTTng – Linux Trace Toolkit Next Generation
OS – Operating System
POSIX – Portable Operating System Interface
RTOS – Real Time Operating System
RTP – Real time Process
SoC – System on Chip
SMP – Symmetric Multi-Processing
XML – eXtensible Markup Language
MPCP – Multiprocessor Priority Ceiling Protocol
## Contents

Chapter 1. Introduction ................................................................................................................. 1  
  1.1 Problem Definition .............................................................................................................. 1  
  1.2 Problem Analysis .............................................................................................................. 2  
  1.3 Research aspects ............................................................................................................... 3  
  1.4 Structure of report ........................................................................................................... 4  

Chapter 2. Background ................................................................................................................. 5  
  2.1 Software Partitioning ........................................................................................................ 5  
  2.2 Multi-core processor ......................................................................................................... 6  
  2.3 VxWorks .......................................................................................................................... 6  
  2.4 Real-time Linux ................................................................................................................ 6  

Chapter 3. Related Works ............................................................................................................ 7  

Chapter 4. Design ........................................................................................................................ 11  
  4.1 XML Model Definition ..................................................................................................... 11  
  4.2 Task Structure ................................................................................................................ 15  
  4.3 Framework ........................................................................................................................ 17  
  4.4 Partitioning Algorithms ................................................................................................... 21  
    4.4.1 Class structure .......................................................................................................... 21  
    4.4.2 Partitioning Algorithm Design .................................................................................. 22  
      4.4.2.1 The modified Blocking time Cost Function ...................................................... 23  
      4.4.2.2 A Novel Parallelization Cost function ............................................................. 24  
  4.5 Data Analyzer .................................................................................................................... 27  

Chapter 5. Implementation ........................................................................................................... 30  
  5.1 Task Implementation ......................................................................................................... 30  
    5.1.1 Task Entry ............................................................................................................... 31  
    5.1.2 Task Synchronization ............................................................................................... 32  
    5.1.3 Task Start ................................................................................................................. 32  
    5.1.4 Task Operations ....................................................................................................... 32  
    5.1.5 Logging Framework ................................................................................................. 33  
  5.2 Test Framework Implementation ....................................................................................... 36  
    5.2.1 Data structures used in the framework .................................................................... 36  
    5.2.2 XML reader .............................................................................................................. 39  
    5.2.3 Module Benchmarker ............................................................................................... 41  
    5.2.4 Partitioning Algorithms ......................................................................................... 44  
    5.2.5 Task Compilation .................................................................................................... 47
Chapter 1

Introduction

Industrial control systems (ICS) find its application in many domains including robotics, process automation, supervisory control and data acquisition system. Generally, control systems acquire inputs through sensors which are processed by controller software and give the corresponding output through actuators. In addition to this controller functionality, many other features like user interaction, data logging and diagnostics are required in an industrial environment. Day by day, the requirements also increase exponentially as new features are being added to the system. Eventually, it is a complex system that operates round-the-clock and interrupting such system in between is sacrificial for the industry. Hence, a real-time system that focuses not only on performance but also on scalability, reliability, predictability and safety is required.

ICS used to run on single core processors by using real-time operating systems. Nowadays as Moore’s law is getting out-dated, the speed of processing in a single core processor cannot keep pace with the demands of the industries. Hence, industries are switching from single core to multicore processors to meet the demand for faster processing of data as in other domains.

The codes for the various functionalities are implemented and maintained for years and replacing them with a new set of codes disrupts the legacy of the system. Hence, migration of tasks from single core to multicore processors results in a lot of challenges with respect to the above mentioned attributes. The purpose of this thesis is to investigate such challenges and devise measures to tackle them.

1.1 Problem Definition

This thesis focuses on analysing the problem of assigning tasks to multicores and how different allocations affect the performance and real time characteristics of the system. The following list of problems were tried to answer during the thesis work:

Problem 1: Develop a partitioning algorithm to meet a set of optimization criteria.
A plethora of literature can be found about partitioning task sets on multi-core processors. The primary problem is to study the current available partitioning algorithms and develop partitioning algorithms to optimize real time characteristics.

Problem 2: Design a model that specifies the real-time system completely.
Creation of a model that is generic and extensible is another problem to be defined. It should be possible to represent the real time systems in the model.

Problem 3: Design and implement a universal framework for generating synthetic load to test partitioning algorithms.
The framework for partitioning tasks should have the flexibility to generate synthetic load according to the user’s requirements. The synthetic load generation should be compatible to different hardware architecture and operating systems.
Problem 4: Incorporate a logging functionality in the synthetic load framework with minimal overhead
There are many open source tools available for logging information related to tasks in Linux distributions. However, they have a significant overhead on the performance of the user’s system. Hence a logging feature with minimal overhead is required and there should be a possibility to turn off such a feature when not required.

Problem 5: Compare and contrast the performance of partitioning algorithms in different environments
The partitioning algorithms should be able to run in different hardware and operating systems. With the help of the logged data, the performance of the partitioning algorithms should be compared based on real-time attributes.

1.2 Problem Analysis
The above mentioned problems have been carefully analysed and the following solutions have been proposed:

Problem 1: Develop a partitioning algorithm to meet a set of optimization criteria.
Different evaluation criteria have been listed down to evaluate the behaviour of the partitioning algorithms:
- CPU Utilization: The partitioning technique adopted can be evaluated based on the utilization of each CPU of the multi-core processor. The user can choose between load balancing and minimization of the number of cores used by the tasks. Load can be distributed across all the cores by using the worst-fit partitioning strategy. Similarly, best-fit allocation can try to pack as many tasks as possible in minimum number of cores.
- Response time: The response time of tasks when each partitioning strategy is adopted can be evaluated and compared. Partitioning of independent tasks into different cores can improve the response time of the system.
- Jitter: The jitter experienced by a lower priority tasks due to higher priority tasks should be evaluated to see the real-time performance of the system. Minimization of jitter implies faster response time of the task.
- Latency: The amount of time between receiving a request for activities like message receive, semaphore lock etc. and processing the request shows how fast the system responds to various kernel activities in the system.
- Blocking time: The time a higher priority task gets blocked by another task that uses the same semaphores can result in undesired time that the higher priority task is blocked. Algorithms can be evaluated based on how much each algorithm can minimize the blocking time.

A methodology for optimizing multiple criteria at the same time can be developed by assigning weightage to one over the other for different cases.
Problem 2: Design a model that specifies the real-time system completely.

The model can be specified as an eXtensible Mark-up Language (XML) with the real time attributes. This facilitates not only to define the characteristics clearly but also the option to add more features later on.

A model that specifies all the system attributes like the number of processes, the message queues and semaphores they use, the number of cores etc. of the system is required. The model should be complete in the sense that all the task parameters are specified. The model should be extensible to different hardware architectures like Intel, ARM etc. and compatible to various real time operating systems, both 32 bit and 64 bit flavours.

Problem 3: Design and implement a universal framework for generating synthetic load to test partitioning algorithms.

The framework should be able to compile and execute the tasks with attributes defined in the model. Also, all tasks should start their execution at the same time in order to see the worst case performance and they should run it for a specified amount of time. The tasks should be able to output a log file with information about when various events have occurred during its execution.

Problem 4: Incorporate a logging functionality in the synthetic load framework with minimal overhead

Logging tools add overhead to the system which affects the tasks’ performance. In order to avoid such overheads, a logging feature can be added to the framework that is capable of being enabled or disabled for each task. Many unwanted parameters that are saved by the logging tools utilize lots of memory can be skipped when using indigenous logging. The overhead introduced by the logging tools is that they write the time stamps into a file and file operations take up significant amount of time. This can lead to poor performance of the user tasks that have real-time requirements. If the logging data is saved into memory and not in file while tasks execute and later on dumping this data into file when tasks finish their execution, such overheads can be overcome. Also, the memory for storing the data should be statically allocated before the tasks start running and thus can avoid memory overflow issues which can occur in cases of dynamic memory allocation.

Problem 5: Compare and contrast the performance of partitioning algorithms

In order to run the partitioning algorithms in different environments, the APIs that are used in the code have to be universally accepted. The condition is accepted by following the POSIX (Portable Operating System Interface) standards defined by IEEE for the APIs used in the code. Only, the non-POSIX compliant functions have to be re-written in other operating systems. The tasks output raw timestamps data into log files which can be processed to create a more user readable form of latencies and jitters.

1.3 Research aspects

The thesis encompasses the following research areas:

- Investigate various approaches for migrating single core real time applications to multi-core.
• Research about various partitioning strategies for multi-core real time systems.
• Compare the strategies based on system behaviour and quality attributes like temporal behaviour, reliability and predictability.
• Devise partitioning strategies to optimise the system attributes and hence meet the quality requirements.
• Design a framework to evaluate the performance of the partitioning strategy based on system attributes like response time, inter-process communication and resource sharing.

1.4 Structure of report

The report is structured as follows. Chapter 2 develops the background information required for the report. Chapter 3 details the previous research and developments that have been done on similar topics. Chapter 4 details the design specifications and motivates those choices. Chapter 5 gives details about how the design specifications are implemented in the software. The results of the implementation are discussed in Chapter 6. Chapter 7 summarises the results and draws conclusions. The challenges faces during the thesis are described in Chapter 8 while Chapter 9 informs about the possible future works and improvements.
Chapter 2
Background

This chapter throws light on the basic concepts which the reader should know before continuing on to the next chapters of the report.

2.1 Software Partitioning

The urge for ICS to migrate their systems into multicore platforms pushes the embedded world to research more about partitioning applications over multiple cores in an adept manner. Allocation of tasks to various processors or cores in a multi-core processor using efficient algorithms is known as software partitioning. Blindly allocating tasks across the cores can result in worse performance than single core environment as co-ordinating the tasks residing in different cores can consume more time. Hence, the efficiency in software partitioning lies in grouping the right set of tasks into the same core. In the same way, separating totally unrelated tasks into different cores are also important thereby parallelizing the jobs on multicore processors.

Software partitioning for real-time tasks is still a topic being researched extensively due to its wide range of applications in real-time systems where predictability plays a major role. Memory and cache coherence are the challenges faced in giving a predictable output from such a platform. Also, the communication among cores can be another factor that needs to be addressed.

2.2 Multi-core processor

A multi-core processor is a computing unit with multiple processors or cores that can independently read and process data. In a hardware perspective, it can be broadly classified into homogeneous and heterogeneous multicores [2]. Homogeneous multiprocessors are processors with identical cores i.e. they have the same frequency, cache size, architecture etc. However, each core in a heterogeneous multiprocessor can differ in these stated factors. Homogenous cores are easier to develop as they all have the same hardware sharing the same instruction set. In a heterogeneous multiprocessor system, each core is specialized for some tasks and they run its own specialized instruction set on each core. Even though the design of heterogeneous multiprocessors is complex, they are remarkable for its efficiency and low power nature. Additionally, heterogeneous multiprocessors with CPU and FPGA architectures in the same silicon chip are also available like the Xilinx Zynq SoC.

From a software perspective [3], multiprocessors are divided into AMP (Asymmetric Multiprocessing) and SMP (Symmetric Multiprocessing). AMP is similar to the environment in uniprocessor systems where each processor runs an individual operating system. AMP can also be homogeneous and heterogeneous depending on whether each core runs the same type and version of OS. The OS should provide standardized mechanisms like IPC for accessing a shared resource in AMP systems. It is best suited for migrating legacy code from uniprocessor systems where the developer doesn't have to care about the task allocation. However, cores can remain underutilized because of this architecture as migrating tasks from one core to the other is not that simple as it has to go through several checkpoints while migrating. Additionally, it is close to impossible if the cores
run different OSs. It is where the advantage of SMP lies where a single operating system manages all the cores. Since the OS has an idea about the tasks, it can allocate tasks to cores based on the behaviour patterns and thereby utilizing the cores effectively. Also the communication between tasks residing in different cores is simpler than AMP as heavy networking protocols are not needed.

2.3 VxWorks

VxWorks is a RTOS developed by Wind River for use in embedded systems. It is used in many sectors primarily safety critical systems where there is a requirement for deterministic performance. VxWorks has support for both single and multicore processors of various architectures like Intel, PowerPC and ARM. The OS is POSIX compliant.

The basic unit of execution in VxWorks is a task. The tasks can be run in kernel space or user space through Real Time Process (RTP). The scheduler in VxWorks schedules the tasks irrespective of whether they are running in user or kernel space.

The tasks executing in kernel space are called Kernel tasks. The kernel and the tasks run in the same address space in supervisor mode. Kernel tasks share the kernel address space and hence are not memory isolated. To execute the tasks in Kernel they are built as kernel modules which can be either statically linked to the OS or can be downloaded and dynamically linked.

The RTP provides the mechanism to execute the tasks in user mode. Each RTP has its own address space, stack for each task and the heap. Each RTP can start multiple tasks in the user space which are then scheduled by the scheduler. The tasks in RTP share the virtual address space of the RTP.

2.4 Real-time Linux

The PREEMPT_RT patch on top of Linux makes the opens source operating system as real-time. It converts Linux that handles the tasks in user space with no deadline guarantees into Linux that meets hard timing deadlines. As time passes by, it will be merged into the mainline Linux due to its popularity in industrial applications. The significant difference of real-time Linux [4] from the mainline version is that it turns Linux into a fully pre-emptible kernel. The spinlocks, that spins until a resource gets free, are replaced with mutexes in the kernel and many critical sections are now preemtible. Addition of priority inheritance protocol the semaphores, mutexes and spinlocks is another significant change from the mainline version. Along with the Ingo Molnar’s real-time patch, the Thomas Gleixner’s clock layer with high resolution support also adds up to the more predictable nature of the real-time patch.
Chapter 3
Related Works

A multitude of literature can be found about partitioning of tasks in a multi-core platform using an SMP operating system. Authors have differed in partitioning algorithms, scheduling policies and resource sharing protocols depending on the optimization criteria they want to achieve. Improving one feature might ameliorate another and hence the trade-offs between parameters should also be taken into account while partitioning task sets. Literatures have given insights into various partitioning strategies and what feature do they optimise which we summarise as given below:

1. Bin packing algorithms: The aim of these algorithms is to pack objects of different sizes into bins based on some condition: If an object does not fit into any of the available bins, a new bin is added.

- **Best fit** allocation tries to utilize the available space in the best way possible by trying to fit objects leaving the least amount of space in the processor. However, this type of task allocation is slower.
- **Worst fit** allocation is just the opposite of best fit because it tries to allocate objects to bins in such a way that after allocation it leaves the maximum amount of space in the core.

- **First fit** allocation
- **Next fit** allocation
- **First fit** allocation starts allocating from the first bin and tries to fit the objects in the available space. After an object is allocated to a bin, it tries to allocate the next object again from the first bin.

- **Next fit** allocation is similar to first fit except that it starts allocation of the next object at the bin where it allocated an object at the previous iteration.

In addition to these basic bin packing algorithms, there exists algorithms such as Best Fit Decreasing (BFD), First Fit Decreasing (FFD) etc. in which objects are arranged in decreasing order of size and bins are arranged in increasing order of space and they are allocated accordingly.

2. **Blocking-Aware Partitioning**: The Blocking Aware partitioning algorithm is detailed in [5].

   The paper assigns weight to each task based on the execution time and the blocking time of the task. The tasks are grouped into macro-tasks if the tasks share resources directly or indirectly. If a macro-task cannot fit in one processor it is marked as broken. The weight of each macro-task is the sum of weights of all the tasks it contains. It also defines attraction between task pairs based on the blocking time that one task can induce on the other. The algorithm starts assigning the tasks/unbroken macro-tasks to the processors. For task belonging to a broken macro-task two alternatives are proposed. First alternative tries to allocate as many as possible of the unallocated tasks in the macro-task which are maximally attracted to the selected task to a processor. The second alternative considers the tasks that are already allocated and tries to assign the selected task to the processor that has maximum attraction to it. If after the assignment the tasks become unschedulable the algorithm fails.

   The results in the paper show that even though the algorithms perform better than the blocking agnostic algorithm, there are some cases which can be scheduled only by the blocking agnostic algorithm.

3. **Synchronization-Aware Partitioning**: The Synchronization-Aware Partitioning algorithm detailed in [6] and [7] also groups tasks into macro-tasks. The tasks/macro-tasks are then sorted in decreasing order of utilization. The algorithm then tries to allocate them to the processors. The number of processors is fixed to the upper bound on sum of utilization of all tasks. The macro-tasks that cannot be allocated are ordered in the cost of breaking them, which is the blocking overhead introduced in the tasks because of transforming a local resource to global resource. The macro-task with minimum breaking cost is broken into two pieces such that one piece fits the processor with largest available utilization. If the fitting is not possible a new processor is added.

   The disadvantages of this algorithm are that no blocking times are considered and also no relationship between the tasks inside a macro-task is taken into account.

4. **Graph Partitioning**: The graph partitioning algorithms use the precedence task graph to partition tasks. The paper [8] details an algorithm to find an optimal partition to minimize either overall bandwidth consumption or the number of cores. Due to the exponential complexity of the optimal solution, it is not suitable for applications consisting of more than 20 tasks. Thus the paper proposes two heuristic partitioning algorithms which partition the graph based on critical paths. Both algorithms find the critical path and assign it to one core.
The first heuristic iteratively finds the critical path of the remaining graph and assigns it using best fit policy. The second heuristic after assigning the critical path iterates through the tasks in order of decreasing computation time and assigns the tasks using Best Fit.

Another graph partitioning algorithm is discussed in [9], where the algorithm randomly selects n nodes of the graph as starting point, where n is the number of processors. It then tries to minimize the communication cost and meet deadlines by trying to include the Heaviest Edge matching (HEM) a node and the Earliest Deadline First (EDF) in the same partition. If after the partitioning the response time is greater than the deadline the weights for EDF and HEM are changed and new partition is generated.

5. **Core shielding**: It is a technique used to achieve predictable behaviour of real-time systems by shielding real-time cores from non-real-time tasks. In the RedHawk kernel presented in [10], deterministic execution of real-time tasks as well as faster and guaranteed interrupt response is achieved when core shielding is implemented. Also immunity towards pre-emptions from operating system is also achieved. Experimental results show that there is significant improvement in interrupt response and minimization of jitter while using the RedHawk kernel over the vanilla Linux kernel. The contention among the cores due to hyper-threading is also highlighted in the paper. Another system known as Asymmetric Real Time System (ARTiS) is discussed in [11] which locks the set of highest priority real-time tasks (RT0) to certain cores and does not allow migration of those tasks to other cores. Another set of real time tasks (RT1+) is allowed to migrate to other cores, including non-real time cores. RT1+ tasks and non-real time tasks are migrated to non-real time cores when they try to disable pre-emption. ARTiS also allows non-real time tasks to execute on the shielded cores when it runs out of real-time tasks to execute, thus balancing the load efficiently.

6. **Cohesion and Coupling**: A partitioning strategy based on the cohesion and coupling properties of tasks is presented in [12]. Cohesion is used as a condition to group the tasks and coupling is used to measure the performance of the partitions generated. Since coupling and cohesion are closely related, the authors present a method to calculate coupling cost based on cohesion with the scheduling requirements considered. Then the Maximum Completion Time (MCT) of the tasks is calculated. The partitioning algorithm clusters the tasks according to functional cohesion and then uses sequential, communicational, temporal, logical and coincidental cohesion to partition the tasks such that the MCT in minimized.

7. **Cost based Partitioning**: A cost based partitioning strategy is discussed in [13], which needs weights of all tasks and a preference matrix for all task pairs as input. The algorithm then tries to find a partition such that the cost function which is the sum of costs for all task pairs in different cores is minimized and the task set is schedulable. The schedulability condition is checked before a task is assigned to a partition. The paper then proposes a weight scheme and preference matrix to reduce blocking time under Multiprocessor Priority Ceiling Protocol (MPCP). The weight of the task is based on the number of critical sections as well as the length of the largest critical section. The preference matrix contains the cost between pair of tasks which is a function of the length of critical sections and the largest critical sections of the tasks.
This partitioning strategy can be used to combine multiple optimization criteria and give weights to each partitioning strategy. The problem of implementing any partitioning strategy in this algorithm simplifies to finding appropriate weight and preference matrix functions for the tasks.
Chapter 4
Design

4.1 XML Model Definition

The first step in creating a framework for generating synthetic real-time loads is to specify the inputs to the framework. After careful consideration it was decided to use an XML to specify all the parameters for the real-time system. The benefits of using XML are:

1. Information is stored hierarchically so it is easy to read and understand.
2. Can be easily parsed. Various XML parsers available for free.
3. Extensibility: No fixed set of tags

To define the structure and the parameters of the input XML, a XML schema was created.

The requirements while creating this schema were:

1. (REQ1) The schema should define a real time system in a generic way which is invariant in any architecture or operating system.
2. (REQ2) The schema should be extensible to include the real time operating system and the architecture specific parameters.
3. (REQ3) The real time system defined should be easily scalable and reduce redundant definitions.
4. (REQ4) A task in the real time system should be assignable to a specific core, a shielded core group or a core calculated by the algorithm.
5. (REQ5) A task in the real time system can have different modules that are run to generate the desired runtime for the task.
6. (REQ6) It should be possible to enable/disable logging for each task.

The tree structure of the schema is shown in Figure 1.

The root node is the test setup whose two children are:

1. Process Group: The process group is a set of interdependent process, semaphores and message queues. Processes in different process group do not share resources or communicate to each other. The test case should have at least one process group. The process group defines the real time task structure and is hence invariant. (REQ1)
   The process group contains following children tags:
   a. Instances: The count in instances specifies the number of copies of all tasks in the process group that should be run. This provides easy scalability and removes redundancy. (REQ3)
b. Process: Each process is uniquely identified by its id. There should be at least one process in a process group. Each process contains the following parameters:
   i. Period: The periodicity of the process specified in microseconds.
   ii. Priority: The priority of the process.
   iii. Offset: The offset of the process in microseconds.
   iv. Utilization: The percentage utilization of the process. It can be in decimal also.
   v. Core: This parameter can have following values (REQ4):
      1. Positive value: It specifies the core number to which the process should be pinned to.
2. “-1”: The task is left unassigned and the scheduler allocates it to any core at runtime.
3. “-2”: The task is not assigned to any specific core but to the set of shielded cores. The scheduler assigns the task to any one of the shielded core.
4. “-3”: The task is allocated to the core which is the output of the partitioning algorithm.

vi. Run_module: This specifies the module that should be run to generate the utilization specified. (REQ5)

vii. Memory_size: This specifies the memory size in bytes that the task will use.

viii. Log_enable: This specifies the various logging modes for the task, which are as follows (REQ6):

   1. “0”: disable logging.
   2. “1”: log all events in the task with their time.
   3. “2”: log average time of all events in the task.
   4. “3”: log all events as well as their averages.

ix. Semaphores: This specifies semaphores the process uses. Each semaphore is identified by the id and contains the time in microseconds the semaphore is taken by the process.

c. IPC: This contains the Inter-process communications in the process group. For now only message queues are supported. Each message is uniquely identified by the id and contains the following parameters:

   i. Count: The number of messages in the message queue.
   ii. Sender: The id of the sender process.
   iii. Receiver: The id of the receiver process.
   iv. Size: The size in bytes of each message in the message queue.

d. Semaphores: This defines the semaphores in the process group. Each semaphore is uniquely identified by id. Each semaphore contains a type field for the semaphore.

2. Parameters: The parameters specify the operating system and architecture specific details for the test case. (REQ2)

   a. Test_name: The name of the test case specified in this XML.
   b. Shielded_cores: This contains the list of cores that should be shielded that is all the currently running tasks are moved to non-shielded cores as well as the spawned tasks are assigned to non-shielded cores. For this reason at least one core should be left unshielded.
   c. Partitioning_algo: This specifies the partitioning algorithm that should be run to assign cores to the tasks for which the core option is specified as “-3”.
   d. Path: This specifies the directory where the log files should be stored.
   e. Test_runtime: The duration the test should be run in seconds.
   f. Force_module_run: A value of 1 forces the modules to be run for benchmarking to calculate execution time of the module.
The parameters section of the XML schema for VxWorks is shown in Figure 2. The XML schema for VxWorks contains the following additional parameters:

a. **Number_of_cores**: The number of cores in the system. For VxWorks the tasks are cross compiles hence the total number of cores is not known.

b. **Target_path**: The location of the path directory when accessed from the target machine. This is needed as the path strings are different on the compilation machine and the target machine.

c. **Architecture**: The architecture for which the tasks are cross-compiled. The architecture string also tells whether the tasks are compiled as RTP (Real Time Processes) or Kernel tasks.
4.2 Task Structure

The requirements for the Task were:

1. (REQ1) The task structure should be generic which means that it should be independent of the real time operating system or architecture.
2. (REQ2) The task structure should be complete. It should be possible to represent all possible tasks with this task structure.
3. (REQ3) There should be a synchronization mechanism to start all the tasks simultaneously.
4. (REQ4) The task should log the events if the logging is enabled.
5. (REQ5) The task structure should be flexible to specify how the task generates its utilization.

The task structure is shown in Figure 3. The same task structure is used both for VxWorks and Linux running on ARM and x86 architectures. (REQ1). The task source code when compiled with different parameters results in different tasks’ executables. All the parameters in the task are hence defined as macros, the values of which are passed during compilation (REQ2).

The different functions which can be run to generate the utilization for the task are called modules. It is possible to specify the module for each task in the XML (REQ5). The task runs that module for so many iterations as to generate the specified utilization. The average runtime for each iteration of the module is calculated by benchmarking it before the task starts. The number of iterations of the module to be run to generate the utilization is calculated from this runtime.

The task is started by the “Task Launcher” of the Test Framework. The task once launched waits for a synchronization signal from the main program so that the starting time of all the tasks is the same (REQ3). Once the signal is received, the task performs the following operations sequentially:

- Iterate over all the receive message queues of the task and for each of them receive all the messages.
- Iterate over all the semaphores the task uses and for each semaphore take the semaphore, busy wait for the time the semaphore needs to be taken and then release the semaphore.
- Run the module for the task for so many iterations as to make up the utilization of the task.
- Iterate over all the send message queues of the task and for each of them send all the messages.
- If logging is enabled, it logs all the above events with the time they occurred. (REQ4)
- Run all the above steps until the end time of the test case.

Once the end time is reached, if the logging is enabled the task generates a log file with the events information and the time they occurred.
Figure 3: Task Flow diagram
4.3 Framework

The requirements when designing the framework were as follows:

1. (REQ1) The framework should be modular.
2. (REQ2) The framework should parse the XML and check for errors.
3. (REQ3) The framework should create the tasks, semaphores and message queues as per the specifications in the XML.
4. (REQ4) The framework should partition the tasks (assign cores to tasks) based on the partitioning algorithm.
5. (REQ5) The framework should benchmark the modules which are used by the tasks.
6. (REQ6) The framework should start the tasks at the same time.
7. (REQ7) The framework should perform clean-up after the tasks are finished.

The flow diagram for the framework for Linux is shown in Figure 4. The framework takes input the test case configuration XML file.

The framework consists of the following 6 components (REQ1):

1. XML Reader: The XML Reader parses the input XML and creates the data structures to be used. It also displays errors if the XML is not well formed. (REQ2)
2. Module Benchmarker: The modules to be used in various tasks should be benchmarked as the runtime of the module is platform dependent. The Module Benchmarker compiles these modules, runs them for a fixed number of iterations and measures the runtime. Each task then calculates the number of iterations of the module it needs to fulfil its runtime. (REQ5)
3. Partitioning Algorithm Manager: This component loads the Partitioning algorithm specified in the XML. It then runs the algorithm for the tasks whose core value was specified as -3 and then assigns them to cores. (REQ4)
4. Task Creator: It uses the data structure created above to create tasks, semaphores and message queues. Then the tasks are compiled with the attributes mentioned in XML by defining them as macros. (REQ3)
5. Task Launcher: The Task Launcher starts the tasks. It also takes care that the tasks start at the same time for worst case performance. (REQ6)
6. Terminator: The Terminator does the clean up after the tasks have finished running. It deletes the semaphores and message queues.(REQ7)

The framework outputs a Main log file which contains information about the log files for all the tasks and serves as input to Data Analyzer.
Figure 4: Flow diagram of Test Framework in Linux
The Framework design for VxWorks contains the same components. However it is divided into 2 applications. This is because VxWorks does not have C/C++ compiler to compile the tasks and modules. Hence the tasks need to be cross compiled on host machine (Windows/Linux) while the tasks are run on the target machine (VxWorks).

![Flow Diagram for Task Compiler in VxWorks](image)

Figure 5: Flow Diagram for Task compiler in VxWorks

The flow diagram for Task_Compiler is shown in Figure 5. The Task_Compiler application is run on Linux or Windows platform and it cross compiles the tasks (Real time processes (RTP) or kernel tasks) and generates the executables. It also creates a text file as output which contains information about the compiled tasks, semaphores and message queues.

This text file serves as input to Task_Launcher application whose flow diagram is shown in Figure 6. The Task_Launcher runs on VxWorks. It reads the text file, benchmarks modules and then starts the tasks. In the end, it outputs the Main log file after removing the message queues and semaphores.
Figure 6: Flow diagram for Task Launcher in VxWorks
4.4 Partitioning Algorithms

4.4.1 Class structure

The Class structure of partitioning algorithms is shown in Figure 7.

All the partitioning algorithms inherit the base class `PartitioningAlgorithm`. `PartitioningAlgorithm` class prepares the data for partitioning algorithms. It contains a pure virtual `runAlgorithm` function that all inheriting classes must implement.

The class structure of WorstFit, BestFit and CostBased partitioning algorithms is shown in Figure 7. Since all of them have chained constructor loading, it runs the `PartitioningAlgorithm` constructor that prepares the data for partitioning. Eventually, when the `runAlgorithm` is invoked the appropriate algorithm runs.

The `CostBasedPartitioningAlgorithm` can have one or multiple cost functions and each cost function has a weight assigned to it. The sum of weights for all cost functions should be 1. The cost functions for cost based partitioning inherit the base class `AbstractCostFunction`. The `AbstractCostFunction` contains the following pure virtual functions which should be implemented in all inheriting classes:

- `getTaskWeight`: Each task has a weight based on the cost function which represents the task’s importance in the cost function.
- `getTaskCorrelationWeight`: This returns the correlation cost between two tasks based on the cost function.

---

Figure 7: Class diagram for Partitioning Algorithms
- getEmptyCPUCost: The cost of assigning a task in a new empty CPU core. It can differ based on the cost functions.

The class structure of BlockingTimeCostFunction and ParallelizationCostFunction which inherit AbstractCostFunction is also shown in Figure 7.

- BlockingTimeCostFunction: It aims to minimize the global critical sections between the tasks when the tasks are partitioned.
- ParallelizationCostFunction: It aims to minimize the overall response time. This is achieved by assigning tasks in such a way that the tasks which can run in parallel are not assigned to the same core.

4.4.2 Partitioning Algorithm Design

A cost based partitioning strategy is discussed in [13]. The paper presents a cost based approach to partitioning algorithms.

Consider you have n tasks T₁… Tₙ to be assigned to m partitions P₁… Pₘ. The algorithm then performs the following steps:

1. Assign weights wᵢ to all the tasks based on the partitioning strategy.
2. Order the tasks T₁… Tₙ in the decreasing order of weights. Let the ordered set be τ₁… τₙ.
3. Create a preference matrix which contains the cost Cᵢᵢ between a pair of tasks Tᵢ and Tⱼ.
4. Iterate for each task τᵢ for k = 1… n
5. Iterate for each partition Pₗ for q = 1…m and calculate the cost Cₚᵢₗ of assigning the task τᵢ to partition Pₗ. The cost of assigning a task to a partition is

\[ Cₚᵢₗ = \sum_{τⱼ \text{ in } k \text{ partition}} Cₖᵢₗ \]  

(1)

6. Order the partitions P₁… Pₘ in the ascending order of the cost of assigning the task.
7. Iterate over partition Pᵢ in P₁… Pₘ and check the schedulability of the system after assigning τᵢ to partition Pᵢ.
8. If the system is schedulable after assigning τᵢ to partition Pᵢ, assign the task to the partition.
9. If all tasks are assigned to partitions, the system is schedulable.

The paper then presents a partitioning strategy to reduce blocking times under MPCP. A strategy is suggested to assign weights to tasks and also to create a preference matrix which denotes the cost between task pairs.

Thus the problem of implementing any partitioning strategy in this algorithm simplifies to finding appropriate weight and preference matrix functions for the tasks. In this thesis, the approach is furthered to the idea of combining multiple partitioning strategies. However the algorithm needs to be tweaked before it can be used. The following changes were proposed:

1. The weight of each task wᵢ should be less than 1 and the sum of weights of all the tasks should be 1.
2. Each element of the preference matrix should be less than 1.

The reason why these changes are needed is explained in the following paragraphs.

The task of combining multiple partitioning strategies is accomplished as follows. Each of the partitioning strategy to be used is assigned a weight such that sum of all weights is less than 1.
Let
τᵢ -> task where i = 1..n
Pᵢ -> partition where l = 1..m
PSₖ -> partitioning strategy where k = 1..q
Wᵢₗ -> weight of each partitioning strategy for k = 1..q partitioning strategies.
wᵢₗₖ -> weight of each task i for kᵗʰ partitioning strategy.
Cᵢⱼₖ -> The preference matrix cost between task τᵢ and τⱼ for PSₖ.

Then the weight for each task is calculated as:

\[ wᵢ = \sum_{k=1}^{q} Wₖ \cdot wᵢₗₖ \]  (2)

The wᵢₗₖ <= 1 and \( \sum_{i=1}^{n} wᵢₗₖ = 1 \) ensures that the weights are scaled properly for each partitioning strategy.

The preference matrix cost between two tasks is calculated as

\[ Cᵢⱼₖ = \sum_{k=1}^{q} Wₖ \cdot Cᵢⱼₖ \]  (3)

The Cᵢⱼₖ <= 1 again ensures that the preference matrices are scaled correctly. The above algorithm is then run with these wᵢ and Cᵢⱼ to assign tasks to the partitions.

Next the partitioning strategy based on reducing blocking time (Blocking time cost function) presented in [13] is discussed with the above mentioned modifications. Then a new partitioning strategy that is based on executing as many tasks as possible in parallel (Parallelization cost function) is presented and hence reducing the overall response time of the system.

### 4.4.2.1 The modified Blocking time Cost Function

A partitioning strategy to reduce blocking time is presented in [13]. In the partitioning strategy the authors assign weights to a task τᵢ based on the function which takes into account the number of critical sections and the length of the largest critical section:

\[ wᵢ = \frac{\sum_{q=1}^{[R]} (n \{ cᵢₗₚₗₖ \} \cdot m \{ cᵢₗₚₗₖ \})}{tᵢ} \]  (4)

Where \( n \{ cᵢₗₚₗₖ \} \) is the number of critical sections in which τᵢ requests a resource Rᵢ, \( m \{ cᵢₗₚₗₖ \} \) denotes the largest critical section of τᵢ requesting Rᵢ, and \( |R| \) is the total number of resources.

This was modified such that instead of dividing with the time period it is now divided by the sum of critical sections of all the tasks.

\[ wᵢ' = \sum_{q=1}^{[R]} (n \{ cᵢₗₚₗₖ \} \cdot m \{ cᵢₗₚₗₖ \}) \]  (5)

\[ wᵢ = \frac{wᵢ'}{\sum_{i=1}^{n} wᵢ'} \]  (6)

This ensures that the first condition in the algorithm modification is met and the weights are scaled properly.

The preference matrix in the blocking time partitioning strategy is denoted as vᵢⱼ for a task pair τᵢ and τⱼ is calculated as follows:
\[ v_{ij} = \sum_{R_q \in R} \frac{(-n \{ c_{l,p,q} \} \ast m \{ c_{l,p,q} \} \ast n \{ c_{j,p,q} \} \ast m \{ c_{j,p,q} \} + 1)}{\left(\sum_{i=1}^{n} w_i \right)^2} \]  \hspace{1cm} (7)

This preference matrix is modified by dividing the multiplication term in the above equation by \( (\sum_{i=1}^{n} w_i) \) and dividing the whole term by the number of resources as shown below:

\[ v_{ij} = \frac{\sum_{R_q \in R} \frac{(-n \{ c_{l,p,q} \} \ast m \{ c_{l,p,q} \} \ast n \{ c_{j,p,q} \} \ast m \{ c_{j,p,q} \} + 1)}{\left(\sum_{i=1}^{n} w_i \right)^2}}{\sum_{q=1}^{|R|} 1} \] \hspace{1cm} (8)

This ensures the second condition is met and the preference matrix values are less than 1.

4.4.2.2 A Novel Parallelization Cost function

A typical real-time application usually contains many event driven real time processes within an infinite loop. Typically one process may branch or signal and start multiple processes. A natural way to represent the system is by a task graph, where each node represents a process and the edges represent the flow of control. The Figure 8 shows a sample task graph consisting of 6 tasks.

Figure 8: Example task graph

The aim of this parallelization partitioning strategy is to reduce the overall response time of the real-time application. This is achieved by running as many tasks in parallel as possible. So the basic idea is that if two tasks can run at the same time, they should be assigned to different cores or, in other words, in different partitions. The goal is to minimize the overlap of tasks that can be run at the same time in all the partitions/cores.

This requires calculating the following parameters:

1. Sequential execution time \( C_s \): This is the minimum time needed to execute all the tasks in the application sequentially on a uniprocessor which is equal to:

\[ C_s = \sum_{i=1}^{n} c_{t_i} \] \hspace{1cm} (9)

Where \( c_{t_i} \) is the execution time of the task \( t_i \).

2. Parallel execution time \( C_p \): This is the minimum time required to execute all tasks in the application such that all tasks are run in parallel. To calculate this each task is assigned a starting time and a finishing time, where finishing time is the starting time plus the execution time. Following steps are preformed:
a. Start with all the tasks which do not need any events to start i.e. they are started at time 0.
b. Assign start time of 0 to the above tasks and end time = start time + execution time.
c. Continue to the immediate children. The start time of child task is the end time of parent and end time = start time + execution time.
d. If a child task already has a start time assigned, then update the value if the new start time (which is same as the end time of parent) is greater than the previously assigned value. Update the end time accordingly.
e. Repeat steps c, d until all tasks have been assigned a start time and end time.
f. Then $C_p$ is the highest end time in the task graph.

Consider the following example. Suppose the execution times of tasks $\tau_1$ to $\tau_6$ are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>5</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>2</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>3</td>
</tr>
<tr>
<td>$\tau_6$</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Example task and execution times

![Task graph with start and end times](image)

The numbers in blue show the start time of the task while the numbers in red show the end time.

So in this example $C_p = 11$ while $C_s = 20$.

3. Total overlap $C_o$: This is the time of overlap of the tasks occurring due to the parallelization. It is equal to

$$C_o = C_s - C_p$$

In the above example $C_o = 9$.

The strategy to assign weights to tasks and to create the preference matrix can then be defined as:

1. The weight ($w_i$) of the task $\tau_i$ should depend on duration of its execution time it overlaps with all the other tasks. Since we need to scale the weights, the weight of the task $\tau_i$ is the overlap time divided by sum of the overlap times of all tasks.

2. The preference matrix ($v_{ij}$) value for two tasks $\tau_i$ and $\tau_j$ should depend on the duration in time their execution overlaps. And the overlap time is scaled by $2*C_o$ to meet the requirements.
Hence for the above example the weights and the preference matrix are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>Weight ($w_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>4/18</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>1/18</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>3/18</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>2/18</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>3/18</td>
</tr>
<tr>
<td>$\tau_6$</td>
<td>5/18</td>
</tr>
</tbody>
</table>

Table 2: Task weights based on parallelization cost function

<table>
<thead>
<tr>
<th>Preference Matrix ($v_{ij}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$\tau_1$</td>
</tr>
<tr>
<td>$\tau_2$</td>
</tr>
<tr>
<td>$\tau_3$</td>
</tr>
<tr>
<td>$\tau_4$</td>
</tr>
<tr>
<td>$\tau_5$</td>
</tr>
<tr>
<td>$\tau_6$</td>
</tr>
</tbody>
</table>

Table 3: Preference matrix of tasks for parallelization cost function

Now the cost based partitioning algorithm can be run using the weights ($w_i$) and the preference matrix ($v_{ij}$).
4.5 Data Analyzer

The Data Analyzer unit is used to analyse the performance of the system that uses the framework. The user should be able to decide upon which algorithm to choose for a particular setup based on the results from the Data Analyser.

Few tools have been tried before the authors came up with the idea of an analysing unit. *Kernelshark* [14], the front end tool of *ftrace* command, runs at the kernel level and logs the information like when a task has been started, when it goes to sleep and wakes up again, when a context switch happens etc. However, it logs this information into a file and this process takes up a significant amount of CPU thus affecting the performance of the user's processes. Another tool called LTTng (Linux Trace Toolkit Next Generation) [15] has also been tested which had a similar effect as the kernelshark on the performance. Then the authors came up with the design idea of indigenous logging and data analyzing along with the framework.

The requirements put forward for creating the data analyzer unit are:

1. It should get raw timestamps from the framework and output a file that has the summary of the latencies and jitter of interested factors.
2. From the results of the data analyzer, it should be possible to compare and contrast the performance of various partitioning algorithms
3. It should be extensible in the future as more parameters are being added.
4. The unit should be platform independent

All the timestamps for the required events are logged into a data structure while the tasks are being executed thus avoiding file operations during the execution time. After all the tasks have finished executing, the logs are written into a log file for analysis by the Data Analyzer. The events read from the log files should be processed to determine the latencies and jitter of various logged events. The following events are logged to obtain the latencies and jitter for each task:

1. Time at which task gets activated
2. Time at which task starts its execution
3. Time at which task finishes its execution
4. Time at which semaphore is requested
5. Time at which semaphore is locked
6. Time at which a write request is given to a message queue
7. Time at which a message is written into a message queue
8. Time at which a read request is given to a message queue
9. Time at which a message is read from a message queue

The difference between the respective events gives the latencies and variance from the mean value gives the jitter as shown in the block diagram below.

The calculated values should be printed on to a file for the user to monitor the performance of the system. So in short, the design flow of Data Analyzer is

Read task log files -> Process the data -> Write to output log file
Figure 10: Data Analyzer Flow diagram
Figure 11: Calculation of values in Data Analyzer
Chapter 5
Implementation

5.1 Task Implementation

As stated in the design requirements, the task needs to be generic and it should be possible to generate all possible tasks using this design by specifying the required parameters. To achieve this aim, the task is implemented with the parameters defined as macros. Different task binaries can be generated from the same source code implementation by compiling the task code by defining the appropriate macros. Following macros are used in the task implementation to provide this versatility:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Required</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK_GROUP_ID</td>
<td>Yes</td>
<td>Defines group id to which the task belongs</td>
</tr>
<tr>
<td>INSTANCE_NUM</td>
<td>Yes</td>
<td>Defines the instance number of the task.</td>
</tr>
<tr>
<td>TASK_ID</td>
<td>Yes</td>
<td>Defines the id number of the task.</td>
</tr>
<tr>
<td>RUN_TIME</td>
<td>Yes</td>
<td>Defines the time the test should be run in seconds.</td>
</tr>
<tr>
<td>PERIOD</td>
<td>Yes</td>
<td>Defines the period of the task in microseconds.</td>
</tr>
<tr>
<td>CORE_NUM</td>
<td>Yes</td>
<td>Defines the core the task is assigned to.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>No</td>
<td>The offset in microseconds of the task from the start time.</td>
</tr>
<tr>
<td>EXEC_PARAM</td>
<td>Yes (Not present in VxWorks)</td>
<td>The execution parameter which is passed to the module. Tells how many times the module should run.</td>
</tr>
<tr>
<td>NUM_OF_MSGQ_R</td>
<td>No</td>
<td>The number of message queues from which the task will be receiving messages.</td>
</tr>
<tr>
<td>MSG_R_MAX_SIZE</td>
<td>Yes, if NUM_OF_MSGQ_R&gt;0</td>
<td>The maximum size message that the task will be receiving from all the message queues. Needed to specify the size of the arrays receiving the message.</td>
</tr>
<tr>
<td>MSG_R_NAME</td>
<td>No</td>
<td>An array of the names of all the message queues from which the task will be receiving messages. The array is of size NUM_OF_MSGQ_R.</td>
</tr>
<tr>
<td>MSG_R_SIZE</td>
<td>No</td>
<td>An array of the size of the messages that the task will receive from each of the message queue in MSG_R_NAME. The array is of size NUM_OF_MSGQ_R.</td>
</tr>
<tr>
<td>MSG_R_COUNT</td>
<td>No</td>
<td>An array of the number of the messages that the task will receive from each of the message queue in MSG_R_NAME. The array is of size NUM_OF_MSGQ_R.</td>
</tr>
<tr>
<td>NUM_OF_SEM</td>
<td>No</td>
<td>The number of semaphores used by the task.</td>
</tr>
<tr>
<td>SEM_NAME</td>
<td>No</td>
<td>An array of the names of all the semaphores the task will use. The array is of size NUM_OF_SEM.</td>
</tr>
<tr>
<td>SEM_TIME</td>
<td>No</td>
<td>An array of the time in microseconds the task will take each of the semaphore in the SEM_NAME. The array is of size NUM_OF_SEM.</td>
</tr>
<tr>
<td>NUM_OF_MSGQ_S</td>
<td>No</td>
<td>The number of message queues to which the task will send messages.</td>
</tr>
</tbody>
</table>
### Table 4: Macros in the TaskImplementation

Apart from the above mentioned macros the following macros need to be defined for logging purpose in the task:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Required</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_ENABLE</td>
<td>No</td>
<td>Defines if the logging is enabled. It can have following values: 1. “0”: disable logging. 2. “1”: log all events in the task with their time. 3. “2”: log average time of all events in the task. 4. “3”: log all events as well as their averages.</td>
</tr>
<tr>
<td>LOG_NAME</td>
<td>No</td>
<td>The complete path and the name of the log file to be generated.</td>
</tr>
<tr>
<td>MAX_LOG_EVENTS</td>
<td>No</td>
<td>The maximum number of events that will be generated during the run time of the test. It is the maximum number of events that will be logged.</td>
</tr>
<tr>
<td>UNIQUE_EVENTS</td>
<td>No</td>
<td>The number of logging events in the task in one iteration. Needed when logging the averages of times of events.</td>
</tr>
</tbody>
</table>

### Table 5: Macros in Task for Logging framework

The task implementation is divided into following parts:

#### 5.1.1 Task Entry

In Linux and VxWorks RTPs the entry point of the task is a main function. This is because in Linux the task is launched as a different process. Also, in VxWorks the RTPs are started as separate processes. This task source with main as entry point is called “task.cpp” both for VxWorks and Linux.

But for VxWorks kernel tasks, the entry point is a function pointer as it is not a separate process. The task source for kernel tasks is called “task_k.cpp” and has a function `task_start` as the entry point.

In VxWorks the task needs the results of the Module Benchmarking as arguments. This is because the benchmarking is run in VxWorks but the tasks have been cross-compiled before in Linux/Windows. So the following two values are passed as arguments:

- **exec_param**: This is the same as EXEC_PARAM macro defined in Linux. It uses the result of the benchmarking of the module the task is using to calculate the number of times the module should be run to generate the required utilization for the task.

---

**MSG_S_MAX_SIZE**
- Yes, if `NUM_OF_MSGQ_S > 0`
  - The maximum size message that the task will be sending to all the message queues. Needed to specify the size of the arrays sending the message.

**MSG_S_NAME**
- No
  - An array of the names of all the message queues to which the task will be sending messages. The array is of size `NUM_OF_MSGQ_S`.

**MSG_S_SIZE**
- No
  - An array of the size of the messages that the task will send to each of the message queue in `MSG_S_NAME`. The array is of size `NUM_OF_MSGQ_S`.

**MSG_S_COUNT**
- No
  - An array of the number of the messages that the task will send to each of the message queue in `MSG_S_NAME`. The array is of size `NUM_OF_MSGQ_S`. 

---

**MSG_S_MAX_SIZE**
- Yes, if `NUM_OF_MSGQ_S > 0`
  - The maximum size message that the task will be sending to all the message queues. Needed to specify the size of the arrays sending the message.

**MSG_S_NAME**
- No
  - An array of the names of all the message queues to which the task will be sending messages. The array is of size `NUM_OF_MSGQ_S`.

**MSG_S_SIZE**
- No
  - An array of the size of the messages that the task will send to each of the message queue in `MSG_S_NAME`. The array is of size `NUM_OF_MSGQ_S`.

**MSG_S_COUNT**
- No
  - An array of the number of the messages that the task will send to each of the message queue in `MSG_S_NAME`. The array is of size `NUM_OF_MSGQ_S`.

---

**Table 4: Macros in the Task Implementation**

Apart from the above mentioned macros the following macros need to be defined for logging purpose in the task:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Required</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_ENABLE</td>
<td>No</td>
<td>Defines if the logging is enabled. It can have following values: 1. “0”: disable logging. 2. “1”: log all events in the task with their time. 3. “2”: log average time of all events in the task. 4. “3”: log all events as well as their averages.</td>
</tr>
<tr>
<td>LOG_NAME</td>
<td>No</td>
<td>The complete path and the name of the log file to be generated.</td>
</tr>
<tr>
<td>MAX_LOG_EVENTS</td>
<td>No</td>
<td>The maximum number of events that will be generated during the run time of the test. It is the maximum number of events that will be logged.</td>
</tr>
<tr>
<td>UNIQUE_EVENTS</td>
<td>No</td>
<td>The number of logging events in the task in one iteration. Needed when logging the averages of times of events.</td>
</tr>
</tbody>
</table>

**Table 5: Macros in Task for Logging framework**

The task implementation is divided into following parts:

#### 5.1.1 Task Entry

In Linux and VxWorks RTPs the entry point of the task is a main function. This is because in Linux the task is launched as a different process. Also, in VxWorks the RTPs are started as separate processes. This task source with main as entry point is called “task.cpp” both for VxWorks and Linux.

But for VxWorks kernel tasks, the entry point is a function pointer as it is not a separate process. The task source for kernel tasks is called “task_k.cpp” and has a function `task_start` as the entry point.

In VxWorks the task needs the results of the Module Benchmarking as arguments. This is because the benchmarking is run in VxWorks but the tasks have been cross-compiled before in Linux/Windows. So the following two values are passed as arguments:

- **exec_param**: This is the same as EXEC_PARAM macro defined in Linux. It uses the result of the benchmarking of the module the task is using to calculate the number of times the module should be run to generate the required utilization for the task.
• busy_iterations: This tells the number of iterations of the busy wait loop that are run in one microsecond. This is used for the busy wait when the task has acquired the semaphore. This value is generated from the benchmarking results of the busy_wait_module. This value is needed only in VxWorks because it lacks a process specific clock (CLOCK_PROCESS_CPU_TIME_ID in Linux).

5.1.2 Task Synchronization
As described in the design requirements, all the tasks should start simultaneously to get the worst case behaviour. To achieve this, after the task is started it waits for a synchronization signal from the Task_Launcher which starts the tasks. The Task_Launcher sends this signal after all the tasks are started and are waiting for the synchronization signal. Different synchronization mechanisms are used for Linux and VxWorks which are detailed below:

• Linux: Once the task is started it sends a SIGSTOP signal to halt itself using the kill command. This halts the process until a continue signal, SIGCONT is received. The Task_Launcher sends this when all tasks have sent a SIGSTOP and are halted.

• VxWorks: VxWorks does not have the kill API as Linux. Hence semaphore is used for synchronization. After the task starts the task tries to lock a semaphore called “/syn” which has been locked by the Task_Launcher. Hence the task gets blocked. This semaphore is released by the Task_Launcher when all tasks are blocked waiting for the semaphore. After the signal is received the task then sleeps to the absolute time which is current time in seconds plus one while the nanoseconds value is 0. This is done to remove the delay between the signal propagation to different tasks. The task start time is set equal to this time.

5.1.3 Task Start
The following operations are done to start the task and run it periodically:

• If the OFFSET is defined sleep for that time.
• Calculate the end time of the task based on the start time and the RUN_TIME of the task.
• Calculate the end time in real-time clock value (CLOCK_REALTIME) which is passed to the message queues as a timeout value.
• Start an infinite while loop. Call the task function within this loop. This is the set of operations that the task should do in each period.
• Exit the while loop when the current time plus the period of the task exceeds the end time. The period value is added to the current time to check if it is possible to accommodate another run of the task within the task run time.
• If the end time is not reached, sleep till the next activation time which is the previous activation time plus the period.

5.1.4 Task Operations
• The function task is called in each period and performs a fixed set of tasks. It does the following steps:
  1. Get the time at the start which is used to measure the runtime of the task.
  2. If there are any message queues from which the task should receive messages i.e. NUM_OF_MSGQ_R>0, then for each message queue:
     a. Open the message queue whose name is defined in MSG_R_NAME.
     b. If successful, iterate for the number of messages to receive from this message queue defined in MSG_R_COUNT. Try to receive the message of size defined in MSG_R_SIZE. The timed receive is used to receive messages with the timeout specified as the end time of the task. If the message queue is being used by some other task, this task gets blocked till the time in timeout is reached.
c. Close the message queue.

3. If there are any semaphores used by the task i.e. NUM_OF_SEM>0, then iterate over them and for each semaphore:
   a. Open the semaphore whose name is defined in SEM_NAME.
   b. If successful, try to lock the semaphore. If the semaphore is already taken, the task goes into blocked state.
   c. When the task locks the semaphore, busy wait for the duration the task is supposed to lock the semaphore which is defined in SEM_TIME.
   d. Release the semaphore.
   e. Close the semaphore.

- Run the module by calling the module_run function of the module with the EXEC_PARAM as argument.

4. If there are any message queues to which the task should send messages i.e. NUM_OF_MSGQ_S>0, then for each message queue:
   a. Open the message queue whose name is defined in MSG_S_NAME.
   b. If successful, iterate for the number of messages to send to this message queue defined in MSG_S_COUNT. Try to send the message of size defined in MSG_S_SIZE. The timed send is used to send messages with the timeout specified as the end time of the task. If the message queue is being used by some other task, this task gets blocked till the time in timeout is reached.
   c. Close the message queue.

5. Get the time at the end. Return.

The task function implementation is same in Linux and VxWorks apart from the fact that the API’s used in Linux and VxWorks are different. The table below summarizes the API’s used in Linux and VxWorks:

<table>
<thead>
<tr>
<th>API function</th>
<th>Linux</th>
<th>VxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open the message queue</td>
<td>mq_open</td>
<td>msgQOpen</td>
</tr>
<tr>
<td>Receive messages timed</td>
<td>mq_timedreceive</td>
<td>msgQReceive</td>
</tr>
<tr>
<td>Send messages timed</td>
<td>mq_timedsend</td>
<td>msgQSend</td>
</tr>
<tr>
<td>Close message queue</td>
<td>mq_close</td>
<td>msgQClose</td>
</tr>
<tr>
<td>Open semaphore</td>
<td>sem_open</td>
<td>semOpen</td>
</tr>
<tr>
<td>Take semaphore</td>
<td>sem_wait</td>
<td>semTake</td>
</tr>
<tr>
<td>Release Semaphore</td>
<td>sem_post</td>
<td>semGive</td>
</tr>
<tr>
<td>Close Semaphore</td>
<td>sem_close</td>
<td>semClose</td>
</tr>
</tbody>
</table>

*Table 6: API comparison between Linux and VxWorks*

### 5.1.5 Logging Framework

In order to measure the performance of the partitioning algorithms and also to debug the problems, logging is needed. The available logging techniques have significant overhead on the performance. Hence a logging framework is implemented embedded into the task to provide minimal overhead. The logging data is stored in the statically allocated data structures and is written out to a file after the task is finished executing. This avoids the resource intensive operations of dynamic memory allocation and writing to disk and hence providing minimal overhead.
Each action that needs to be logged is referred to as an event in the framework. The enum `log_event_type` lists down all the possible events that can occur in the task. The table below describes those events with the enum value:

<table>
<thead>
<tr>
<th>Log_event_type enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_event_task_arrival</td>
<td>This event logs the time of arrival of the task i.e. the time when the task becomes available to run (moves to a READY state).</td>
</tr>
<tr>
<td>log_event_task_start</td>
<td>This event logs the time the task starts to execute. This can be different than the arrival time as the task might get blocked.</td>
</tr>
<tr>
<td>log_event_task_end</td>
<td>This event logs the time the task is done executing for the period and goes to sleep.</td>
</tr>
<tr>
<td>log_event_task_runtime</td>
<td>This event logs the time the task is executing. This can be different than the difference between start and end time as this does not include the time the task was blocked.</td>
</tr>
<tr>
<td>log_event_semaphore_requested</td>
<td>This event logs the time the task requests a semaphore. The task may lock the semaphore immediately or get blocked if the semaphore is taken.</td>
</tr>
<tr>
<td>log_event_semaphore_acquired</td>
<td>This event logs the time the task actually locks a semaphore.</td>
</tr>
<tr>
<td>log_event_semaphore_released</td>
<td>This event logs the time the task releases a semaphore after it has held it for the required time.</td>
</tr>
<tr>
<td>log_event_msg_send_request</td>
<td>This event logs the time the task requests to send a message to a message queue.</td>
</tr>
<tr>
<td>log_event_msg_sent</td>
<td>This event logs the time the message is sent to the message queue. This is different than the send request as the task may get blocked if the message queue is full.</td>
</tr>
<tr>
<td>log_event_msg_receive_request</td>
<td>This event logs the time the task requests to read a message from a message queue.</td>
</tr>
<tr>
<td>log_event_msg_received</td>
<td>This event logs the time the message is received from the message queue. This can be different that the receive request as the task will get blocked if the queue is empty.</td>
</tr>
<tr>
<td>log_event_main_tasks_start</td>
<td>This event logs the time the test case starts to run.</td>
</tr>
<tr>
<td>log_event_main_tasks_end</td>
<td>This event logs the time the test case finishes.</td>
</tr>
</tbody>
</table>

*Table 7: List of events that are logged*

Each event in the task is stored as an “event_log” data structure which contains the following variables:
- `log_event_type`: The enum telling what type of event it is.
- `time`: The time the event took place.
- `ID`: This tells the id of the object for which the event took place. For task it is task ID, for message queue it is message queue ID while for semaphores its semaphore ID.
- `value`: This is an optional field and is set to 0 when not in use. It is primarily used to record the message number that the task is sending/receiving to/from a message queue.

The “task_log” data structure stores the complete logging information for the task. It contains the following variables:
- `group_id`: The group id of the task.
- `instance`: The instance number of the task.
- `task_id`: The id number of the task.
- `core`: The core number the task is running on.
- **run_count:** This keeps a track of the number of times the task is run during the complete run time. It is incremented in each period.

- **count:** If the LOG_ENABLE is 1 or 3, this counts the number of events that are logged during the task’s run time.

- **events_log:** If the LOG_ENABLE is 1 or 3, this is the array of “event_log” data structure that stores all the events that occur during the task’s run time. The size of the array is equal to MAX_LOG_EVENTS macro which is defined during task compilation.

- **avg_count:** If the LOG_ENABLE is 2 or 3, this counts the number of average log events recorded. This should be equal to the UNIQUE_EVENTS macro and serves as a security check.

- **avg_events:** If the LOG_ENABLE is 1 or 3, this is the array of “event_log” data structure that stores the average time of the events. It keeps a running average which means a new average value is calculated each time. The size of the array is equal to UNIQUE_EVENTS macro which is defined during task compilation.

Depending on the mode of the logging defined in the LOG_ENABLE macro, either log_event or log_avg_event or both of these functions are called at appropriate points in the task. These functions access the “events_log” and “avg_events” arrays in the “task_log” data structure respectively and store the event information.

If logging is enabled, the task writes out the data in the “task_log” data structure to a log file after the task finishes execution (i.e. the end time of the task is reached). The name and path of the log file is defined in the LOG_NAME macro.
5.2 Test Framework Implementation

The test framework starts by running a set of pre-run units that pave the way for the tasks to start. This is followed by the task execution and then analysing the execution results. Proper care is taken while designing the framework to ensure that nothing is hampering the user tasks’ execution as they are real-time in nature. So to make sure that all the memory used up for pre-processing is cleared before the task execution, the pre-run functionalities from parsing the XML up to launching the tasks are called as separate function. When the function exits, a destructor is called that clean up the memory used by the data structures within the function. Only the data that should persist even after the task execution are stored separately. Hence, the “persist_data” structure is created before the pre-run function is called and a pointer to it is passed on to the function as argument. The structure is replete while the function passes through various units and is available even after the function exits.

5.2.1 Data structures used in the framework

The test data structure is at the top of the hierarchy that stores the overview information about the test setup. The test structure contains a task_group_id_map which links it to the structure of task_group. Each task_group contains the list of tasks and hence connected to the task structure. The contents of each structure are detailed below.

The test structure consists of the following:

1. Testname – A string that stores the name assigned for the test case.
2. Total_cores – In Linux, a system call is used to get the number of cores the operating system handles. This is later used to find if the user inputted a valid set of shielded cores.
3. Path – The folder location where the log files are generated when the tasks finish their execution.
4. Test_runtime – The time for which the tasks should run from the instant at which they are launched simultaneously.
5. Shielded_cores – It is an unordered set of integers that stores the cores numbers which are solely dedicated for the tasks to run.
6. Partitioning_algorithm – the partitioning algorithm that the user plans to run on the test setup so that the tasks are wisely allocated to different cores. It can be either one of BestFit, WorstFit or CostBased algorithm which is detailed in section 4.4.
7. Cost_function – If the user intends to use a cost based algorithm, then it is passed on to this vector of type partitioning_function. The partitioning_function consists of two elements:
   o The name of the algorithm - Currently, it supports optimizations based on blocking time and parallelization.
   o The weightage of the algorithm from 0 to 1 (floating point).
8. Module_set – The benchmarking modules that are used in the framework defined as an unordered_set of module structure which is described in Section 5.2.3.
9. Force_module_run – A Boolean variable that lets the user to turn on and off the benchmarking process every time the framework is invoked. Once the benchmarking is done on a platform, the binary files are available for later use by the tasks.
10. Task_group_id_map – an unordered map where the key is the task_group ID and the value is the structure task_group.
Additional variables in VxWorks

11. Arch – a string that stores the architecture of the target platform.
12. Target path – The path where an output text file is generated after task compilation. Since the task compilation and launching is done separately, the information from one to the other is passed as a text file.
13. IsSMP – Boolean variable to show if the target is a Symmetric Multi-Processing unit or not.
14. IsRTP – Boolean variable that shows if the tasks are realized as real-time processes or not.
15. Msgcount – number of message queues used in the test setup
16. semcount – number of semaphores used in the test setup.

The task_group structure consists of:

1. Id – The identifier for the task group.
2. Task_id_map – an unordered mapping of task structure with an integer as the key.
3. Semaphore_map – an unordered map of semaphore structure with an integer as key. The semaphore structure contains two elements:
   a. ID – a unique identification number for semaphore
   b. Type – the type of semaphore. Currently, the framework supports binary and counting semaphores.
4. Msgqs – a vector of msgq structure. The structure for message queue comprises of:
   a. ID – unique identification for the message queue in the task group.
   b. Task ID – the ID of the task associated with the message queue.
   c. Size – the size of each message in the message queue.
   d. Count – the number of messages in the message queue.
5. Number_of_instances – the number of times the task_group has to be replicated to enable the scaling the number of tasks in the system.

The task structure (mentioned in #2) is the vital data structure for a task’s execution. The elements of the task structure are as follows:

1. Offset – the shift in task launching as compared to other tasks in microseconds.
2. Period – the time interval in which the tasks get activated periodically.
3. Utilization – a floating point value that specifies the CPU utilization in percentage.
4. Priority – The accepted priority range is from 1-139 according to the definitions from Linux [16]. 1-99 is considered as real-time priorities and the range 100-139 is considered as non-real time priorities where the tasks won’t be given any real time privileges.
5. Core – As mentioned in the Design, the core value can be any value from 0 to (maximum number of cores -1) or any one of -1, -2 or -3 if the user doesn’t wish to pin the task to a core.
6. Memory size – the size in bytes the task requires in the memory.
7. Run module – the module that should be run in order to mimic the execution time. It can be either of busy_wait_module that just loops itself until the execution time is reached or matrix_scan_module that scans a matrix which depends on cache behaviour.
8. Log enable – this feature lets the user to log the timestamps of only the interested tasks.
9. Semaphore – the semaphore that the task is planning to lock for some amount of time is read. The ID as well as the duration in microseconds for which it has to be locked is read. Nested semaphores are not supported in this framework.
10. Message send – If the task is a sender, then the message queue is added to the msg_send vector of type msgq (described above).

11. Message receive – Similar to message send, if the task is supposed to receive a message, that message queue will be added here as a vector of type msgq.

As mentioned before, all these data are not required to be kept until the end of the system execution. Only some of them are required for data logging and termination purposes. The data that should persist are defined in persist_data structure which are:

- testname - The test name that has to be printed in the master log file.
- semaphores - The unordered set of semaphores that have to be unlinked when the tasks finish execution. Only the names of semaphores have to be preserved for unlinking.
- msgq - The unordered set of message queue names that also have to be unlinked.
- task_logs - The task log names for printing in the master log file.
- path - The path where the log files are stored to tell the data analyzer where to look for the task logs.
- total_cores - The total number of cores in the system for the data analyzer to print the events sorted in ascending order of timestamps on each core.
- core_shielding_enabled - If core shielding is enabled, then it has to be disabled after the tasks finish its execution so that the OS tasks can be distributed to all the cores rather than stuffing themselves in one.

For each unit, the Linux implementation details are explained first followed by the implementation details in VxWorks. The implementation is described based on these data structures.
5.2.2 XML reader
At first, the framework accepts an XML file and checks if it is a valid one. If it is so, the XML parser is invoked. An open source parser called Rapid XML [17] is used to parse the XML file as it is fast, stable and targeted for embedded platforms where memory is critical. It parses tag by tag from the root up to the last unit and fills up the above described data structure for framing up the tasks and its dependent functionalities. Once an erroneous data is spotted in the XML, it is printed into the terminal and the code exits right there.

The XML reader starts off by reading the first node in the system doc.first_node and assigns it as the root node where doc is the XML file of type xml_document<>. Within the root node, there are two main nodes – parameters and process_group – which are accessed by calling the first node in the root node like root_node->first_node. Using these pointers, the values are read from XML and assigned to the respective structure by converting the string into the required data type. For parsing the repeated ones, like in shielded cores where different core number have to be specified, the next_sibling API is used. The first_attribute API is used to read the attribute, if any, of a tag.

The tags that require error handling while parsing the XML are cited below:

**In Parameters tag**
- Path – When a path is read as a string from the XML, it is validated against the file system to confirm that it exists or not. If there is no such path in the system, the XML reader prints an error in the console and exits there.
- Shielded cores – While parsing the XML it is checked if all cores are used as shielded cores. If so, it returns an error and exits the code. It is because at least one core should be left for the other operating system tasks to run. Also any value greater than the maximum number of cores will also result in an error.
- Partitioning algorithm – Any name other than Best Fit, Worst Fit or Cost Based results in an error from the parsing unit.
- Force_module_run expects either a zero or one and any other value is an error.
- In VxWorks, the supported architectures are:
  - SIMLINUXdiab
  - SIMLINUXdiab_SMP
  - SIMPENTIUMdiab_RTP
  - NEHALEMdiab
  - NEHALEMdiab_RTP
  - NEHALEMdiab_SMP
  - ARMARCH7diab
  - ARMARCH7diab_SMP
  - ARMARCH7diab_RTP
SIMLINUXdiab(_SMP) is used for simulation/testing purposes in the eclipse environment. NEHALEMdiab(_RTP)/(_SMP) is the Intel target hardware used for testing the framework. ARMARCH7diab(_RTP)/(_SMP) is the architecture for Xilinx Zynq which was another testing platform. Any other value passed to the architecture tag returns an error.

**In Process group tag**
- Utilization – Since the CPU utilization is accepted as percentage, values greater than 100% are not acceptable.
• Core – It is required that the core value should be [0, Max. number of cores), or [-3, -1]. Other values results in an error from the system.
• Run module – module names other than `busy_wait_module` and `matrix_scan_module` are not permitted.
• Log enable – the values outside the range of [1, 3] causes the parser to return an error.
5.2.3 *Module Benchmarker*

These modules provide the functionality to specify how the task will generate its utilization. The simplest way to do this is to run a busy wait loop for the runtime duration of the task. However the problem with this simple approach is that it does not take into consideration the effects of cache access or the memory read/write time. For this reason the user is given the flexibility to write his own functions which he wants to run for the tasks.

5.2.3.1 Implementation Concept

To provide the above mentioned functionality, the task always includes a header file “run_module.h” which contains the definition of a `module_run` function. The task uses this function to generate its utilization. All the modules which the user wants to write should include this header file and implement the `module_run` function. When the task is compiled the C++ file is specified which provides the implementation of this function. So the same task source code when compiled with different modules’ source code will generate binaries with different functions that generate the task’s utilization.

To benchmark tasks, the concept of dynamic loading is used. Dynamic loading is a technique where a library or binary can be loaded at runtime. It is then possible to extract the function and variable addresses from the library, execute those functions and then unload the library/binary from the memory. C++ provides a header file `<dlfcn.h>` which provides the following POSIX compliant API’s [18):

- `dlopen`: To load the library.
- `dlsym`: To extract the contents of the library.
- `dlclose`: Unload the library from the memory.

The module benchmarker implementation can be divided into these sub sections:

1. The modules.
2. Compiling the modules.
3. Loading and running the modules.
4. Reading and storing the results.

5.2.3.2 The modules

As stated above each module should implement the `module_run` function. The function takes input the parameter which tells how many iterations to run. The name of the module C++ file is given as input into the XML in the “run_module” parameter inside the Process tag. The C++ file for the module should be stored in the TASK_SRC_DIR which is defined in “definitions.h”.

Sometimes the module might need some more parameters which are read from the XML. These parameters can be passed into the modules by defining them as macros, which are named `ARG<argument number>`, where the argument number ranges from 0 to the maximum arguments for the module.

A module structure is defined in “Module_Benchmarker.h” which contains following variables:

1. name: the name of the module.
2. argc: the number of arguments for the module.
3. args: an integer array which holds the arguments.
4. run_time: the run time of 1000000 iterations of the module.

When the XML Reader reads the module name it calls `get_module` in “Module_Benchmarker.h” which creates the module structure for the specified module name with appropriate number of
arguments and returns the structure. The module is then put in a set called “module_set” in the test structure. The set makes sure that there are no duplicates. It is to be noted that the modules with same name but different arguments are considered unique. Hence the equal to criteria for the set not only checks if the name of the module is same but also the value of each of the arguments.

The following 2 modules have been implemented:

1. **busy_wait_module**: This module has no arguments hence the argc variable is 0. The module_run function of this module takes input the time in microseconds. It then adds these microseconds to the current time and then busy waits till the current time is equal to the sum.

2. **matrix_scan_module**: This module scans an array in order to get some effects of cache hit and miss. This module has one argument ARG0. This argument is tells the module how much memory the task is using and is specified in “memory_size” parameter in XML. The module then creates an array of that memory size and iterates through it. The argument in module_run of this module tells how many times to iterate thorough the array.

### 5.2.3.3 Compiling the modules

Each of the unique modules in the “module_set” set is compiled by generating a compilation string which is then executed using the system command. A sample compilation string is shown below:

```
g++ -shared -DARG0=1000 -o ../Debug/matrix_scan_module_1000.so -fPIC ../src/task_src/matrix_scan_module.cpp
```

The module is compiled with -shared flag to generate a shared object. The arguments needed by the module are defined as macros using the –D flag. It also includes –fPIC flag which tells the compiler to generate Position Independent Code suitable for dynamic linking.

After compilation a .so file is generated. To guarantee the uniqueness of the generated binary, the name of the .so file is the module name followed by the arguments separated by “_”. The compiled binary is stored in the BUILD_DIR specified in “definitions.h”.

### 5.2.3.4 Loading and running the modules

As described above the modules are loaded using the dynamic loading functionality of C++. For each compiled binary the dlopen function is used to open the binary. The dlsym function is then used to extract the module_run function from the opened binary which returns the function pointer to the module_run function. The function is then run with 1000000 passed as the parameter and the time is measured using the POSIX clock function clock_gettime. In the end the binary is closed using the dlclose. The measured time is stored in the run_time of module’s structure.

### 5.2.3.5 Reading and storing the result

There was a requirement not to run the module benchmarker every time because the run time of the modules will be the same on the same platform. It should also be possible to force the module benchmarker to run which is specified in the “force_module_run” parameter in the XML. These requirements are incorporated as follows:

1. The file “module_benchmarker.bin” stored in the BUILD_DIR is a binary file which contains the module data structure from the last time the Module Benchmark was run.
2. If the force_module_run is true, then the Module Benchmarkers benchmarks all the modules and overwrites the file with the new data.
3. If the force_module_run is false, then the Module Benchmarkers first reads the file. If it is found that some modules required by this test case have not been benchmarked before
then the benchmarking iterations are run for that module and the result appended to the file.

The implementation of Module Benchmarker in VxWorks is similar to Linux described above but it is divided into 2 projects. The first project “test_suite” which is run on Windows/Linux which cross compiles the tasks and the second project called “test_launcher” on VxWorks which runs the tasks on VxWorks. So the modules are compiled in “test_suite” project while the benchmarking is done in the “test_launcher” project. The differences in the VxWorks implementation are described below.

5.2.3.6 Compiling the modules in VxWorks
The compilation process is similar to the Linux. The compilation string generated is shown below:

-DCPU=_VX_SIMLINUX -DTOOL_FAMILY=diab -DTOOL=diab -D_VX_WORKS_KERNEL - D_VSB_CONFIG_FILE="$WIND_BASE/target/lib/h/config/vsbConfig.h" - DIP_PORT_VXWORKS=69 -DARG0=1000 - ISWIND_BASE/target/h - o " /home/rizwin/workspace/test_suite/run/matrix_scan_module_1000.out" - c " ../src/task_src/matrix_scan_module.cpp"

As shown in the compilation string dcc compiler is used which cross compiles for VxWorks. The architecture is specified in –DCPU parameter as NEHALEM and “diab” toolchain is used in TOOL_FAMILY for compilation. It also tells that the VX_WORKS version in use is 6.9. Finally the module is compiled with its arguments to generate an .out file. This file is stored in the path provided in the XML file.

5.2.3.7 Benchmarking the modules in VxWorks
The modules are benchmarked in the “test_launcher” project that runs on VxWorks. The modules are loaded using dynamic loading and run in a similar manner as Linux. As mentioned before, we can compile tasks as RTP or Kernel tasks in VxWorks which is specified in the architecture tag in XML. It is to be noted that for both RTPs and kernel tasks the module benchmarking is done in the “test_launcher” which is run as a kernel task.
5.2.4 Partitioning Algorithms

The class diagram of the partitioning algorithms is shown in Figure 7. All the partitioning algorithms must inherit the “PartitioningAlgorithm” class and implement its pure virtual functions like runAlgorithm. The PartitioningAlgorithm class contains the following two variables which are used in partitioning algorithms:

1. task_set: It is a vector which contains all the tasks that are to be assigned the core by the partitioning algorithm, i.e. they have the core value as -3 in XML. The vector has “task_id” as its element which is a data structure that contains the group_id, id, instance of the task.

2. core_util: It is a unordered_map data structure with the key as the core number which is an integer and the value as a data structure “core_param”. The “core_param” structure contains “util” which is a floating value representing the utilization of the core and “taskAssigned” which is a vector of “task_id” assigned to this core.

5.2.4.1 Preparing the data

The prepareData function in Partitioning algorithms fills up the data in these two data structures. It first creates the core map and sets the utilizations to zero. It then iterates through the tasks and if the task has core value as -3 it is assigned to task_set vector otherwise if it has a core value of greater than 0 it is assigned to that core’s “taskAssigned” vector and the utilization of the core is updated. If the utilization of any core exceeds 100% an error is generated and the system exits.

The prepareData function is called in the constructor of the PartitioningAlgorithm class. For all the algorithms inheriting the PartitioningAlgorithm class, chained constructor loading is used which call the PartitioningAlgorithm’s constructor.

5.2.4.2 Running the algorithm

The algorithms are run by calling the runAlgorithm function. As shown in the design, worst-fit, Best-fit and Cost based partitioning have been implemented and are described below:

5.2.4.2.1 Worst Fit Partitioning Algorithm

The Worst Fit partitioning algorithm is described in Chapter 3. It is implemented as “WorstFitAlgorithm” class which inherits “PartitioningAlgorithm” class. The idea is to assign the task to the core which has the least utilization. The runAlgorithm function in WorstFitAlgorithm class iterates over all tasks in “task_set”. For each task, it iterates over all cores in the “core_util” map to find the core which can accommodate a task (i.e. has utilization less than 100 after assigning the task) and has maximum space remaining after assigning the task to that core or has the minimum utilization. If no core can accommodate the task an error is displayed and the code exits. Else the task is assigned to the core’s “assigned_task” vector and the utilization of the core is updated.

In the end it returns a unordered_map with the task_id as key and the core it is assigned to as value.

5.2.4.2.2 Best Fit Partitioning Algorithm

The Best Fit partitioning algorithm is described in Chapter 3. It is implemented as “BestFitAlgorithm” class which inherits “PartitioningAlgorithm” class. The idea is to assign the task to the core which has the least free utilization after assigning the task. The runAlgorithm function in BestFitAlgorithm class iterates over all tasks in “task_set”. For each task it iterates over all cores in the “core_util” map to find the core which can accommodate a task (i.e. has utilization less than 100 after assigning the task) and has minimum space remaining after assigning the task to that core or has the maximum utilization. If no core can accommodate the task, an error is displayed and the code exits.
Else the task is assigned to the core’s “assigned_task” vector and the utilization of the core is updated.
In the end it returns a unordered_map with the task_id as key and the core it is assigned to as value.

5.2.4.2.3 Cost Based Partitioning Algorithm
The modified Cost based partitioning algorithm is explained in 4.4.2. It is implemented as “CostBasedPartitioningAlgorithm” class which inherits “PartitioningAlgorithm” class. In cost based partitioning algorithm each partitioning strategy is represented as a cost function. The partitioning algorithm then tries to minimize the costs of the resulting partitions. It also explains a way to combine multiple cost functions to produce the optimal result. Hence cost functions are central to cost based partitioning algorithm.

The base class for cost function is “AbstractCostFunction”. As shown in the class diagram (Figure 7: Class diagram for Partitioning Algorithms), all the cost functions must inherit this base class. The “AbstractCostFunction” class constructor takes input the pointer to “test” object and the weight that is assigned to this cost function as floating value. It defines following purely virtual functions which all the inheriting classes should implement:

- getEmptyCPUCost: The cost to assign the task to an empty partition or core. It should return a value between 0 and 1.
- getTaskWeight: This returns the weight of the task whose id is passed as argument. As defined in section 4.4.1, the task weight should be between 0 and 1.
- getTaskCorrelationWeight: This returns the value of the preference matrix for the two tasks whose ids are passed as arguments. This value should also be between 0 and 1.

Sorting the tasks
The CostBasedPartitioningAlgorithm class constructor takes as argument a vector of “partition_fn” data structures which contain the name of the cost function and its weight. It then creates the objects for those partition functions and stores them in “fn_list” which is a vector of the pointers to “AbstractCostFunction” class. A sort structure is declared which uses the “getTaskWeight” functions of the cost functions to calculate the weight of the tasks according to Eq(2) and compare them. The tasks are then sorted in decreasing order of their weights using this sort structure.

Assigning the tasks
The runAlgorithm function in the Cost based partitioning algorithm tries to assign the task to the partition which has the least cost. It iterates over all the tasks in the “task_set”. For each task ti it then iterates over all the partitions/cores. If the utilization of the core/partition Pk is less than 100 after assigning the task, the function getCostofTaskonCPU is called which returns the cost of assigning the task to Pk.

The getCostofTaskonCPU function iterates over all tasks already in the partition Pk. For each task tj in the partition it calculates the correlation cost vij between ti and tj by combining the getTaskCorrelationWeight from all the cost functions using the Eq(8). The total cost of assigning task on the core/partition Pk is obtained by adding vij for all ti.

The runAlgorithm function tries to find for each task ti the partition where the cost of assigning the task is the least. It returns an unordered_map with the task_id as key and the core it is assigned to as value.

The cost functions
The following cost functions have been implemented:

1. **Blocking Time Cost Function**
The Blocking time cost function is described in section 4.4.2.1. It is implemented as “BlockingTimeCostFunction” class which inherits the “AbstractCostFunction” class and implements the above mentioned functions in the following way:

   - Constructor: In the constructor the dividing factor it calculated to make sure the sum of all task weights is less than 1. It iterates over all tasks in test case. For each task it iterates over all the semaphores the task uses and adds the time the semaphore is used to obtain the dividing factor “max_cost”.
   - getEmptyCPUCost: The cost to assign the task to an empty CPU for the blocking time cost function is 1. This can be calculated from the Eq(8), where since there are no tasks the multiplication term is 0.
   - getTaskWeight: The task weight of each task is calculated by adding up the times that the task locks a semaphore. This sum is divided by the “max_cost” to obtain the weight as a floating value.
   - getTaskCorrelationWeight: This is calculated by finding the semaphores that are used by both the tasks passed as arguments. It iterates over the semaphores contained in first task. If the semaphore is contained in the other task’s list of semaphores, then the time they are locked by the two tasks is multiplied and added to the total cost. In the end the total cost is divided by the square of the “max_cost” and subtracted from 1 according to Eq(8) and the result is returned.

2. **Parallelization Cost Function**
The Parallelization cost function is explained in section 4.4.2.2. It is implemented as “ParallelizationCostFunction” class which inherits the “AbstractCostFunction” class and implements the above mentioned functions:

   - Constructor: In the constructor the Sequential execution time $C_s$ and the Parallel execution time $C_p$ is calculated. $C_s$ is calculated by iterating over all tasks in the test case and adding their execution times. To calculate $C_p$ first all the tasks are found which do not receive any messages from message queues. These tasks are the starting point. Then for each starting task a recursive Depth First Search algorithm (runDFS) is run which populates the “taskset” map containing the task id as the key and the start time of the task as the value. The end time of the task is the start time plus the execution time. The Depth First Search keeps track of the maximum value of end time of all tasks which is the $C_p$ value. The end time and start time of tasks are calculated according to the rules defined in 4.4.2.2. The “divisor” is then calculated by subtracting $C_p$ from $C_s$ and multiplying the result by 2.
   - getTaskWeight: The task weight is the sum of overlap times of this task with other tasks. This is calculated by iterating over all the other tasks in “taskset” map. For each task it is calculates the overlap time using the start time and end time of the tasks. The sum of all overlap times divided by the “divisor” is the task weight.
   - getTaskCorrelationWeight: The task correlation weight is calculated by calculating the overlap time between the two tasks using their start time and end time. The overlap time is divided by the “divisor” to get the task correlation weight.
5.2.5 Task Compilation

The responsibility of this unit is to create a compilation string using the information from the data structures filled while parsing the XML. This unit iterates over the process groups and over each instance of it to generate such a string. It runs after the partitioning algorithm has decided upon the task allocation to cores.

There is only one standard file called task.cpp that has the task structure incorporated in it along with semaphores, message queues and module benchmarking. Depending on the user requirements, the values are passed as arguments to the task file with -D flag and processed during runtime. Different executables are generated for various tasks and each executable is identified with a unique task name.

A sample compilation string for Linux platform is shown below:

```
g++ -std=c++0x -DTASK_GROUP_ID=1 -DINSTANCE_NUM=1 -DTASK_ID=2 -DRUN_TIME=10 -DCORE_NUM=0 -DPERIOD=200000 -DEXEC_PARAM=54000 -DLOG_ENABLE=3 -DLOG_NAME="/home/rizwin/workspace/test_suite/run/graphsample_task_1_1_2.log" -DMAX_LOG_EVENTS=900 -DUNIQUE_EVENTS=18 -DNUM_OF_SEM=2 -DSEM_NAME="/sem_1_1_2","/sem_1_1_3" -DSEM_TIME='{200,100}' -DNUM_OF_MSGQ_R=2 -DMSG_R_MAX_SIZE=1000 -DMSG_R_NAME="/mq_1_1_1","/mq_1_1_6" -DMSG_R_SIZE='{1000,1000}' -DMSG_R_COUNT='{1,1}' -DMSG_S_MAX_SIZE=1000 -DMSG_S_NAME="/mq_1_1_2","/mq_1_1_3" -DMSG_S_SIZE='{1000,1000}' -DMSG_S_COUNT='{1,1}' ../src/task_src/task.cpp ../src/task_src/busy_wait_module.cpp ../src/utils.cpp ../src/task_src/Logger.cpp -o ../Debug/task_1_1_2 -pthread -lrt
```

5.2.5.1 Parameters in the compilation string

The g++ compiler is used to compile the task’s C++ code. g++ version 4.6 is used to compile the framework as well as tasks and -std=c++0x is added to support the experimental features used in the codes.

The TASK_GROUP_ID, INSTANCE_NUM and TASK_ID are assigned while the task creator iterates over each instance of various tasks in each process group. These IDs are used to define a task name. A task is always named as task_X_Y_Z where X, Y and Z are group ID, instance number and task ID respectively. If such an executable already exists in the folder location, the file is deleted beforehand. All the generated strings are run using system function call and the corresponding compiled binary file is generated.

The parameters like RUN_TIME, PERIOD and OFFSET are read from the data structures and assigned directly to the respective arguments.

As written in the Section 4.1, the core value that is passed in the XML can be any value in the range [0, Maximum number of cores-1] or any of -1, -2 and -3. If the core value is in [0, Maximum number of cores-1], then it is assigned directly to CORE_NUM without further processing. If it is -1, then the task creator lets the operating system do the core assignment job and it can go to any of the available core. However, a value of -2 restricts the operating system to assign the task to any of the available cores. If the core value is -3, then the operating system assigns the task to a busy wait module which is a generic module that blocks any thread from running.
the shielded cores. Finally, if the value is -3, the partitioning algorithm takes care of it and assigns a value to `CORE_NUM`.

The `EXEC_PARAM` is a factor of the run time of the module used for benchmarking that ran before the tasks creator. The period and utilization of the task is known in advance and it is known that, utilization (U) in percentage is calculated by the equation:

\[ U(\%) = \frac{C}{T} \times 100 \]  

Where C is the execution time and T is the time period.

The benchmarking module is run for a very large number of times; say one million, and the total time taken for execution is calculated and stored. So for creating a delay of C time units,

\[ \text{Number of iterations required} = \frac{C \times \text{Benchmark iterations}}{\text{Runtime taken for the module}} \]  

This result is stored in the `EXEC_PARAM` field.

The `LOG_ENABLE` tag that defines the mode of logging is also read from the task structure and assigned directly. The log file for storing the results is obtained by concatenating the test name with the task name which looks like `<testname>_task_X_Y_Z` where X, Y and Z are group ID, instance number and task ID respectively. The output files are generated at the folder location specified within the path tag and so the path along with the log file name is assigned to `LOG_NAME`.

`MAX_LOG_EVENTS` is used to find the total number of events to be logged for the task. This is useful in defining the array size in task implementation. Also, `UNIQUE_EVENTS` is the number of unique events that happen in a single cycle time or period of a task.

\[ \text{Maxnr. of log events} = \frac{\text{Total runtime}}{\text{Period}} \times \text{Unique events} \]  

\[ \text{Unique events} = \text{Count}_\text{sem} \times \text{Events}_\text{sem} + \text{Count}_\text{mq} \times \text{Events}_\text{mq} + \text{Events}_\text{task} \]

where  
\[ \text{Count}_\text{sem} \] is the number of semaphores used by the task  
\[ \text{Events}_\text{sem} \] is the number of events recorded for each semaphore  
\[ \text{Count}_\text{mq} \] is the number of messages sent and received by the task in a single period  
\[ \text{Events}_\text{mq} \] is the number of events recorded for each message queue  
\[ \text{Events}_\text{task} \] is the number of events recorded for a task in each period

When the parameters for message queues are passed to the task, both senders and receivers have to be distinguished from each other. In order to facilitate that, the parameters are prefixed with `MSG_S_` for senders and `MSG_R_` for receivers.

Along with the C++ file for task, few other files also have to be linked to generate the final task binary. They are:

- The module that is used to benchmark the timing of the task execution.
- The file that supports the APIs for logging, Logger.cpp
- The utilities used in the task file for various calculations like conversion from struct timestamps to other units, basic arithmetic operations using the struct timestamps etc.

Two linkers `-pthread` and `-lrt` are passed to the task for the IPC support and other real-time features.
5.2.5.2 Amendments in VxWorks

Apart from the parameters that are explained above for Linux, VxWorks need more information to generate the binary file. Wind River uses its own dcc compiler on VxWorks for compiling the C++ code.

A sample compilation string for a Kernel task looks like as follows:

dcc -g -r -tNEHALEM:vxworks69 -W:c:,-Xcib-optim-off -Xansi -Xlocal-data-area-static-only -W:c++:CPP -ei1518,4177,4223,4301,4550,5409,1606 -ei4193,4826,4381,4237,1573,4007,4082,4177,4223,4260,4550,5361,5828,2273,5387,5388 -ei1512,4092,4111,4144,4152,4167,4171,4174,4186,4188,4191,4192,4223,4231,4236,4284,4375,4494,4513,5152,5157 -Xforce-declarations -Xmake-dependency=0xd -DCPU=_VX_NEHALEM -D_TOOL_FAMILY=diab -D_TOOL=diab -D_VRS_KERNEL -D_VRS_VX_SMP -D_VRS_CONFIG_SMP -D_VSB_CONFIG_FILE="$WIND_BASE/target/lib_smp/h/config/vsbConfig.h" -DIP_PORT_VXWORKS=69 -DTASK_GROUP_ID=1 -DINSTANCE_NUM=1 -DTASK_ID_NUM=2 -DRUN_TIME=10 -DCORE_NUM=0 -DPERIOD=200000 -DLOG_ENABLE=3 -DLOG_NAME='"/home/rizwin/workspace/test_suite/run1/kSMP_task_1_1_2.log"' -D_MAX_LOG_EVENTS=800 -DUNIQUE_EVENTS=16 -DNUM_OF_SEM=2 -DSYNC_TIME='(200,100)' -DNUM_OF_MSGQ_R=2 -DMMSG_R_MAX_SIZE=1000 -DMMSG_R_NAME='"/mq_1_1_1","/mq_1_1_6"' -DMMSG_R_SIZE='(1000,1000)' -DMMSG_R_COUNT='(1,1)' -DNUM_OF_MSGQ_S=1 -DMMSG_S_MAX_SIZE=1000 -DMMSG_S_NAME='"/mq_1_1_2"' -DMMSG_S_SIZE='(1000)' -DMMSG_S_COUNT='(1)' -I$WIND_BASE/target/h -o "$WIND_BASE/target/usr/h ../src/task_src/task.cpp ../src/task_src/busy_wait_module.cpp ../src/task_src/utils.cpp ../src/task_src/Logger.cpp

The compilation string for a RTP is shown below:

dplus -g -tNEHALEML:rtxp -Xansi -WDVSB_DIR="$WIND_BASE/target/lib" -W:c++:CPP -Xforce-declarations -ei14167,4092,4144,4152,4167,4171,4174,4186,4188,4191,4192,4223,4231,4236,4284,4375,4494,4513,5152,5457,2273,5387,5388 -Xmake-dependency=0xd -D_VX_CPU=_VX_NEHALEM -D_VX_TOOL_FAMILY=diab -D_VX_TOOL=diab -DTASK_GROUP_ID=1 -DINSTANCE_NUM=1 -DTASK_ID_NUM=2 -DRUN_TIME=10 -DCORE_NUM=0 -DPERIOD=200000 -DLOG_ENABLE=3 -DLOG_NAME='"/home/rizwin/workspace/test_suite/run1/RTPSMP_task_1_1_2.log"' -D_MAX_LOG_EVENTS=800 -DUNIQUE_EVENTS=16 -DNUM_OF_SEM=2 -DSYNC_TIME='(200,100)' -DNUM_OF_MSGQ_R=2 -DMMSG_R_MAX_SIZE=1000 -DMMSG_R_NAME='"/mq_1_1_1","/mq_1_1_6"' -DMMSG_R_SIZE='(1000,1000)' -DMMSG_R_COUNT='(1,1)' -DNUM_OF_MSGQ_S=1 -DMMSG_S_MAX_SIZE=1000 -DMMSG_S_NAME='"/mq_1_1_2"' -DMMSG_S_SIZE='(1000)' -DMMSG_S_COUNT='(1)' -I$WIND_BASE/target/usr/h -o "$WIND_BASE/target/usr/h ../src/task_src/task.cpp ../src/task_src/busy_wait_module.cpp ../src/task_src/utils.cpp ../src/task_src/Logger.cpp

"/home/rizwin/workspace/test_suite/run1/RTPSMP_task_1_1_2.vxe" -lstdstd
In VxWorks, the form of executable decides how the compilation string should look like. The tasks can run either as a real time process (RTP) or as a kernel task on the target platform hosting VxWorks. This can be identified from the extension of the output file to be generated in the compilation string -.out (binary) for kernel tasks and .vxe (library) for RTPs.

Like Linux, VxWorks supports a plethora of architectures and hence it is also requirement to pass the architecture as argument to generate the binary. The parameter \texttt{-DCPU=_VX\_NEHALEM} gives an idea to the linker that the binaries are generated for \texttt{NEHALEM architecture}.

There are different compilers available for VxWorks such as Diab, GNU and Intel C++ Compiler (ICC) [19]. Currently in this system, the \texttt{Diab} compiler is used which is shown in the string as \texttt{--DTOOL\_FAMILY=diab --DTOOL=diab}.

The kernel is configured as either unicore or SMP. Hence kernel tasks and RTPs can run on both kinds of platforms. The architecture tag in the XML has to be changed according to the platform it runs.
5.2.6 Creation of message queues and semaphores

After the tasks are compiled, the list of message queues and semaphores obtained from the process group are created. This ensures that by the time the created tasks start executing all the semaphores and message queues are available on the system and the execution time of tasks will not be used for creating them. Since it is a POSIX requirement that the named semaphores and message queues should start with a slash(/), they are name as /sem_<group ID>_<instance nr.>_<semaphore ID> and /mq_<group ID>_<instance nr.>_<message queue ID> respectively.

A structure, struct mq_attr, is used for defining the initialization parameters of message queues. The size the message queue and the number of messages are assigned to the attribute. The message queue is then initialized using mq_open with the name, flags that control the message queue creation, file modes and the attributes mentioned above as parameters. The flags include O_CREAT which creates the message queue if it doesn’t exist and O_RDWR which gives tasks the permission to both send and receive messages. The modes enable the read/write permission for the owner of the file, group owner of the file and other users.

The creation of semaphores is simpler than message queues as no initialization structure has to be passed. The semaphore name, initialization flags, file mode creation mask and the initial value of the semaphore are passed as arguments to the sem_open function. The flags and modes specified are same as for the message queues.

VxWorks

In VxWorks, the function used to create message queue is a bit different from that in Linux in terms of the parameters. The API used for message queue creation is msgQOpen which takes in the name, size, count of messages, options, creation mode and context value. The option specified is MSG_Q_FIFO which facilitates tasks to be queued in FIFO order waiting for messages. The creation is mode is OM_CREATE which tells the compiler to create the message queue if it is not found. The name is passed starting with a forward slash for the OS to search the message queue name in public namespace. If the queue doesn’t exist, a new one is created. If any errors occur, the error message is printed and the message queue is closed.

Similarly, semaphores are created using semOpen API that requires name, type, initial state, options, mode and context value as arguments. As mentioned before, a new semaphore is created or an existing one is opened depending on the mode. Currently, the type supported in the framework is binary and the initial state is SEM_EMPTY. The option is SEM_Q_FIFO as in the case of message queues. If the creation fails, an error is printed and the semaphore is closed.
5.2.7 Data transfer between Task Compilation and Launching units (VxWorks)

After the tasks are compiled in the VxWorks workbench, it has to be launched in the target running the VxWorks. Since the workbench resides in a different machine than the target, the task launcher needs some information in order to start running them in the target machine. The necessary information to start the tasks are:

1. The test name for the setup.
2. The target path where the output log files have to be saved.
3. The test run_time in seconds and nanoseconds (from timespec structure).
4. Force module benchmarking option is enabled or not.
5. The task set should be launched as RTP or not.
6. The set of shielded cores
7. Number of message queues
8. The list of message queues – the name, size and count of each.
9. Number of semaphores
10. The list of semaphores – the names and their type.
11. The tasks and its relevant information such as:
   a. Group ID
   b. Instance number
   c. Task ID
   d. Core assigned to the task
   e. Task priority
   f. Utilization
   g. Module name used for benchmarking

At the launcher’s side, the file is passed as argument and hence it is first checked whether it is a valid one or not. Then it reads the first line which contains the information #1-#5 mentioned above.

In order to process the list of shielded cores (#6), the launching unit should know the available cores in the system. vxCpuEnabledGet is used to get the cores in the system as a cpuset and the list of shielded cores is parsed and checked whether it belongs to list. If not, an error is displayed and launching halts.

The count of message queues (#7) that are supposed to be created are read and that many number of lines are then read to get the details of each message queue. From each line that follows (#8), the message queue name, the size of each message and the count are read and assigned to a structure msgq. Each structure is then pushed to a vector of type struct msgq using push_back function. Likewise, the semaphore details are read and the vector of struct sem is filled up.

Finally, the task details (#11a – #11g) are read till the end of file is reached and assigned to struct task. Each structure is then pushed to the vector of struct task after all the parameters are read.
5.2.8 Core shielding

A set of CPUs and memory nodes is called as a CPUSET. The cpuset file system is a pseudo-file system that is used to control the deployment of CPUs and memories to tasks. It is used in the project to group tasks in a core or a set of cores and migrating to other cores is forbidden. This helps the system to run clutter free without the interference from other operating system tasks. The user tasks run in a 'shielded' cpuset while the OS tasks run in 'unshielded' cpuset and both the sets are mutually exclusive. Python scripts are used to facilitate such functionalities in a system and it is currently available as a library for C/C++ and hence by including those libraries such functionalities are available in the framework. The code is linked with -lbitmask and -lcputset in order to use the APIs.

Firstly, the APIs cpuset_cpus_nbits and cpuset_mems_nbits are called to get the number of bits available in a CPU bitmask and a memory node bitmask respectively on the current system. These bitmask values are then allocated to the shielded and unshielded CPUs as well as memory nodes followed by clearing all the bit fields.

By default, all the tasks belong to the root cpuset file system mounted at ‘/dev/cpuset’. So the ‘/dev/cpuset’ directory is created and the cpusets are mounted there. Then two cpusets namely ‘/dev/shielded’ and ‘/dev/unshielded’ are created. If such folder location already exists, then all the tasks in those cpusets are moved back to the root cpuset and the cpusets are re-created. All the OS tasks will be relocated to the unshielded cpuset and the user tasks will be assigned to the shielded cpuset so that the user tasks can run uninterrupted by the OS tasks.

The total number of cores in the system can be obtained by using a system call get_nprocs_conf. By using this, the unshielded cores can be obtained by excluding all the shielded cores from the total number of cores and hence the cpusets ‘shielded’ and ‘unshielded’ are assigned with the respective cores. The CPUSET creator then iterates from 0 to the number of cores-1 and checks if a core belongs to the set of shielded cores. If so, the corresponding bit number as the core number is set in the shielded cpuset and if not, the bit in the unshielded cpuset is set. The bitmask_setbit API is used to set the bits in the bitmask structures for shielded and unshielded CPUs. Using these bitmasks values, the shielded and unshielded cpusets are created using cpuset_setcpus API.

Once the cpusets are created, the next job is to relocate the OS tasks from the shielded cores into the unshielded cores. The list of process IDs (PID) are obtained from the root cpuset using cpuset_init_pidlist and then migrated over to the unshielded cpuset using cpuset_move_all API. The unshielded cpuset is passed as argument to the API. However, few system tasks cannot be migrated to unshielded cores as the operating system doesn’t let them to do so. After the migration is done, the bitmasks are cleared to save memory. If everything goes well, the CPUSET creator returns a positive result and negative otherwise.

Some screenshots of what happens before and after core shielding are depicted in Figure 12: Core Utilizations without core shielding and Figure 13 : Core Utilizations with core shielding. Note that the CPU utilization of cores 2, 3 and 4 are 0% when they are shielded.
VxWorks

The Core shielding unit in VxWorks is relatively simpler as compared to that in Linux. The \textit{vxCpuReserve} is used to reserve the cores for a set of tasks. A cpuset is created for the set of shielded cores is passed as an argument to the \textit{vxCpuReserve} function which reserves those CPUs.
5.2.9 Task launcher

5.2.9.1 Linux:
The final unit before the tasks start running is the task launcher. At first, it kills all the tasks with names starting with ‘task_’ to ensure that no zombie/defunct processes [20] are running. The zombie processes may hamper the execution of newly created ones if they have the same name.

This unit iterates over all the instances of each task in every process group and creates a new process with the name task_<group ID>_<instance nr.>_<semaphore ID>. Using this unique ID, the core to which the task is assigned is obtained either from the partitioning algorithm or from the XML if the user wants the task specifically pinned to a core.

The function fork is called to create a new child process and here it branches into two – one child process and the parent process.

Launching the Child Process
If the user lets the tasks partitioning to the OS by specifying -1 in the core tag in the XML, then the newly created task is assigned to the root cpuset. Otherwise, the task is moved to the shielded cpuset that was created before (mentioned in Section 5.2.8).

cpuset_move is used to allocate the task to the desired cpuset. If any error arises due to cases like the cpuset doesn’t exist, then the Task Launcher prints the error and exits.

Next, the task is pinned to the required core using the multi-core CPU affinity APIs. Like the cpuset library, these are also manipulated using bit masking. Once the corresponding core number is set using bit masking, the sched_setaffinity is called to set the core affinity for the child process. Once the core is set, the task should be assigned to one of the two real-time scheduling protocols [21] in Linux for which the sched_setscheduler API is utilized. SCHED_FIFO, the First In First Out scheduler, is used for the framework tasks. This scheduler processes the task in the order of priority and for the tasks having the same priority it adopts the FIFO policy. The function execvp is then called which overwrites the current program’s address space with the program file passed as argument to the function. Hence, the compiled task name is passed as argument and the task execution starts.

Within Parent Process
While the child process launches, the parent process enters a waiting state where it delays itself until all the child processes start and get paused by issuing a stop signal. The waitid API is used to wait to ensure the child processes have stopped. The process group ID is passed as argument so that the parent waits for all child processes it forked. Also, it should be mentioned which signal the parent waits for from the child process and hence the WSTOPPED is passed as argument to notify that the parent waits for a stop signal from the child process.

All task should have a reference zero time to start their execution. Some tasks might have offsets and these deviations should be from the reference zero time. Signalling is one of the most convenient ways in Linux to notify the tasks about an activity without much overhead. The task execution is halted using SIGSTOP signal and resumed using SIGCONT signal. However, the next issue to address was to send the SIGCONT signal to all tasks at the same time because sending it one after the other might not serve the purpose of simultaneous starting when the number of tasks is huge. Hence, the concept of process group is utilized to achieve simultaneous signalling to all the newly created tasks. Since all the tasks are forked from the same parent process, all of them will have the
same process group ID as the parent and hence giving a continue signal to the process group will serve the purpose of simultaneous starting. A redundant measure of synchronization is also done by letting all the tasks sleep up to two seconds and zero nanoseconds from the time of issuing the continue signal using the absolute clock, CLOCK_MONOTONIC. For instance, if the time when the signal is issued is x seconds and y nanoseconds, then the reference zero will be (x+2) seconds and 0 nanoseconds. (x+2) seconds is taken because there is a possibility that tasks get the continue signal very close to (x+1) seconds and so by considering the overhead in sleeping and activating, the tasks can get unpunctual in starting at the specified time. The time stamp of this instant i.e. (x+2) seconds is recorded as the reference zero for rest of the periodic timing and logging purposes. The master process then sleeps until run time has elapsed in order to start the termination functions.

5.2.9.2 VxWorks
In VxWorks, all the tasks in the test case can be run either as RTP or Kernel tasks. The task compiler generates a .vxe file for RTP and a binary file for Kernel tasks. The RTPs execute as new process in VxWorks with separate memory space. Hence RTP task implementation has a main function as entry point. However kernel tasks are run within the same process and same memory space. Hence it needs a function pointer as entry point.

In VxWorks a semaphore “/syn” is used for synchronization purposes. This is needed to guarantee that all tasks start at the same time for worst case performance. Before starting the RTPs or kernel tasks the semaphore is locked. When the tasks try to lock the semaphore they get blocked. After starting all the tasks, a check is performed for all the tasks to see that they are in blocked state using the API, “tasksIsPended” with the task ID. If some task is not blocked, sleep function is called for 1 second. This is repeated until all tasks are in blocked state. Then the semFlush function is called with the semaphore ID. This unblocks all the tasks that were blocked on the semaphore.

The function to start RTP or kernel tasks iterates over all the tasks in the test case. It then calculates the “exec_param” which along with “busy_iter” needs to be passed to the RTP or kernel task. As explained previously these parameters need to be passed in VxWorks as the module benchmarking is performed after the compilation of the tasks. “exec_param” is the parameter passed to the module run function to indicate the number of iterations while “busy_iter” tells the number of busy wait iterations in 1 second for the busy wait loop when the semaphore is taken. RTP and kernel tasks are then started as follows:

Starting RTP
VxWorks provides the API rtpSpawn to launch the RTPs. This function takes in the following arguments:

- The name of the RTP with full path as a character pointer.
- The arguments which needs to be passed to main as a null terminated character pointer array.
- A null terminated array of environment variables which is NULL in this case.
- Priority of the task as integer.
- Initial user’s side stack size for the task hardcoded to 0x100000.
- The RTP option which is set to “RTP_LOADED_WAIT”. This options instructs the rtpSpawn function to return only after the RTP is completely initialized and the execution is about to transfer to user mode.
- The task options are specified as 0.

If unsuccessful, rtpSpawn returns error and the program exits. Otherwise it returns the RTP ID. This ID is used to get the initial task ID for the RTP using the rtpInfoGet function. The task ids are
stored in the task_ids vector which is used to check if the task has reached in blocked state. Also this task id is used to set the CPU affinity of the RTP.

**Starting Kernel Tasks**

VxWorks provides the API `taskSpawn` to launch kernel tasks. However the function requires the pointer to the function which starts the task. This is achieved by the concept of dynamic loading discussed in section 5.2.3. VxWorks provides the library `loadLib` and `unloadLib` to load and unload the libraries respectively. Following functions are used to get the pointer to the `task_start` function from the binary:

- **loadModule**: This function loads the object module from the file specified to the memory. It adds all the external definitions to the system symbol table.
- **symFindByName**: This function scans through the system symbol table to find a symbol matching the name specified which in this case is `task_start`. It returns OK if the symbol is found and it assigns the pointer of the symbol to the `pValue` argument.
- **unload**: This function is used to unload the modules. The modules are unloaded after the test has finished running because the definitions of `task_start` functions need to remain in the memory in order to run the tasks.

Once the pointer to the `task_start` function is obtained for a particular task, kernel task can be started using the `taskSpawn` API which takes as input the following parameters:

- Name of the task to start.
- The priority of the task.
- The task’s stack size which is specified as 0x100000.
- The function pointer to the entry point of the task.
- Number of arguments to be passed which is 3 including the task name.
- `exec_param` and `busy_iter` as the arguments to be passed to the task.

The `taskSpawn` returns the task id if the task is created successfully. This task id is stored in the task_ids vector which is used to check if the task has reached in blocked state. On error the program exits. Also this task id is used to set the CPU affinity of the kernel task.
5.2.10 Data Analyzer

The Data Analyzer unit takes in a master log file generated by the framework as input. The master log file has the same name as the test case name. This log file contains the necessary information about the task logs such as:

- Folder location where the log files are stored.
- File names of the task logs to be processed.
- The test name of the setup.
- The number of cores in which the tasks ran.

The data analyzer then iterates over all the log files and fills its data structures with the corresponding data. The first line in a log file gives an idea about the task to the data analyzer and it consists of the following:

- Group ID – It defines the process group to which the task belongs
- Task ID – It is the identifier of a task within a process group
- Instance ID – It is used to distinguish among the identical instances of the same task
- Core in which the task executed
- Logging mode – The various modes can be auto mode that logs timestamp of every event, average mode that logs the average time of each event and a mode that saves both of these data simultaneously.

Once a task identification is done, the core in which the task ran and the logging mode is read to have a picture of further data storage of events and its processing. This information along with the events is stored in a data structure.

Consecutive lines are then read from the log files which are the timestamps of the recorded events. Similar to the task identification, each event is stored as a data structure with its identifier. The ID varies according to context – task ID for task related information, semaphore ID for semaphores and message ID for message queues. The event types that are described in Section 5.1.5 are stored for the respective event. Also, the timestamp at which the event occurred is recorded as seconds and nanoseconds.

Once all the events are logged, the data is further processed to calculate mean, maximum, minimum and standard deviation of the events. Averaging at each stage while iterating over the events was tried which shows a deviation from the actual mean at the end. Hence, the alternative was adopted that sums up the timestamps of each event and divided by the total number of occurrence of that event to get the average time of an event.

The difference between timestamps of the interested events are recorded on the go and saved as a new event. Later on, with these difference the maximum, minimum and the standard deviation of task starting latency, response time, execution time, semaphore lock latency, message send and receive latency are calculated by iterating over a task’s events list.

Standard deviation of $X$ is calculated by the following equation

$$\sigma = \sqrt{E[X^2] - (E[X])^2}$$

(15)

Where $E[X^2] - (E[X])^2$ or $\sigma^2$ is the Variance of $X$. 

58
Hence to make it simpler, the variance is calculated first and then the square root of it is taken just before printing into the file. Consider ‘a’ is the timestamp when the request for an event is given and ‘b’ is the timestamp when the request is granted. (b-a)^2 is calculated while these timestamps are read from the log file. After all the logging is done, Σ (b-a)^2 is calculated along with other events’ sum of timestamps. Σb and Σa are ready at this point and so (Σb-Σa)^2 is also estimated. The difference Σ(b-a)^2- (Σb-Σa)^2 is calculated and followed by their square root to get the standard deviation.

The calculated values are printed onto a file ending with <test_name>_latencies.log. The process is repeated for all tasks in the test group and appended to the output log file.

The total delivery time of a message in a message queue i.e. the time taken to send a message by one task and receive in another, is also averaged and recorded. When a message send request event is encountered in a task’s list of events, the group ID and instance ID is saved along with the message ID. Similarly, when a message is received, the details are recorded. After all the log files have been read, this message keys are used to compare the identifiers and if they are matching, the difference between the timestamps are calculated and printed to the output file.

Finally, all the events that occurred in the same core are sorted in ascending order of timestamps and printed in separate files. <test_name>_events_coreX.log is the naming convention used for such files. Thus, the activities occurred in each core are logged for further analysis purposes.
Chapter 6
Results

This section presents the test results for the various partitioning algorithms implemented that are run using the load generation test framework. The test cases are run both on Linux and VxWorks and their results are presented in the following sections:

6.1 Linux:

*Test1: Compare the performance of various partitioning algorithms*

The test consists of 6 tasks, 3 semaphores and 6 message queues. The period of each task is 200 ms and the execution times for the tasks are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>30 ms</td>
</tr>
<tr>
<td>τ₂</td>
<td>34 ms</td>
</tr>
<tr>
<td>τ₃</td>
<td>26 ms</td>
</tr>
<tr>
<td>τ₄</td>
<td>50 ms</td>
</tr>
<tr>
<td>τ₅</td>
<td>26 ms</td>
</tr>
<tr>
<td>τ₆</td>
<td>40 ms</td>
</tr>
</tbody>
</table>

Table 8: Utilizations of task for Test1

The task graph for the tasks is shown below in Table 9 which shows the inter-process communication between the tasks using message queues. The table on the right shows the semaphores the task use along with the time each task takes the semaphore.

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>sem1</th>
<th>sem2</th>
<th>sem3</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₂</td>
<td>-</td>
<td>200 us</td>
<td>100 us</td>
</tr>
<tr>
<td>τ₃</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₄</td>
<td>200 us</td>
<td>100 us</td>
<td>-</td>
</tr>
<tr>
<td>τ₅</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₆</td>
<td>-</td>
<td>-</td>
<td>100 us</td>
</tr>
</tbody>
</table>

Table 9: Task graph and semaphores for Test 1

The tasks are to be partitioned into 2 shielded cores. The core assignments of tasks for different partitioning algorithms are shown in Table 10. The Parallelization and Blocking time are the cost functions that are run on the Cost based partitioning algorithm. The “No Partitioning” does not assign the tasks to any specific core but just instructs them to execute in the 2 shielded cores. So the tasks can migrate from one core to other.
<table>
<thead>
<tr>
<th></th>
<th>Worst Fit</th>
<th>Best Fit</th>
<th>Parallelization</th>
<th>Blocking Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>τ₂</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>τ₃</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>τ₄</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>τ₅</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>τ₆</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 10: Core assignments for tasks for Test1*

Figure 14 shows the response time of the tasks in nanoseconds plotted for the various partitioning algorithms. The blocking times of the tasks in nanoseconds plotted against the various partitioning algorithms is shown in Figure 15. As expected the response time is minimum for the parallelization algorithm. The “no partitioning” is the case where the tasks are assigned dynamically to the cores and hence the response times in this case closely match to those of parallelization. The Worst Fit algorithm tries to distribute the task loads among cores and hence achieves good but not the most optimal response times. Best Fit and Blocking time cost function show the worst performance in response times as they try to fit all the tasks into one core. They assign all tasks except task 3 to one core as task 3 does not have a semaphore.

The blocking time is minimum for the Blocking time cost function and the Best Fit, as the tasks is grouped together in one core. The blocking times for Worst Fit, Parallelization and No Partitioning are almost the same due to the fact that they try to distribute the tasks into the two cores and hence induce more blocking.

![Response Time](image)

*Figure 14: Task response times for Test1*
Test 2: Compare the performance of cost based partitioning algorithm using Blocking time and Parallelization Cost functions with varying weights.

The test consists of 6 tasks, 3 semaphores and 6 message queues. The period of each task is 200 ms and the execution times for the tasks are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
<th>$\tau_5$</th>
<th>$\tau_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization ($U_i$)</td>
<td>15%</td>
<td>17%</td>
<td>13%</td>
<td>12%</td>
<td>13%</td>
<td>20%</td>
</tr>
<tr>
<td>Execution time ($C_i$)</td>
<td>30 ms</td>
<td>34 ms</td>
<td>26 ms</td>
<td>24 ms</td>
<td>26 ms</td>
<td>40 ms</td>
</tr>
</tbody>
</table>

The task graph for the tasks is shown below in Table 12 which shows the inter-process communication between the tasks using message queues. The table on the right shows the semaphores the task use along with the time each task takes the semaphore.

The tasks are to be partitioned into 2 shielded cores. The core assignments for tasks for varying weights of Parallelization and Blocking time is shown in Table 13. The parallelization cost function weight is denoted by $P$ while the Blocking time weight is denoted by $B$. 
Table 12: Task graph and semaphores for Test2

<table>
<thead>
<tr>
<th></th>
<th>sem1</th>
<th>sem2</th>
<th>sem3</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₂</td>
<td>-</td>
<td>200 us</td>
<td>100 us</td>
</tr>
<tr>
<td>τ₃</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₄</td>
<td>200 us</td>
<td>100 us</td>
<td>-</td>
</tr>
<tr>
<td>τ₅</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>τ₆</td>
<td>-</td>
<td>-</td>
<td>100 us</td>
</tr>
</tbody>
</table>

Table 13: Core assignments for tasks for Test2

<table>
<thead>
<tr>
<th></th>
<th>P=1, B=0</th>
<th>P=0.5, B=0.5</th>
<th>P=0.25, B=0.75</th>
<th>P=0.2, B=0.8</th>
<th>P=0, B=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>τ₂</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>τ₃</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>τ₄</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>τ₅</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>τ₆</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 16 shows the response times of the tasks in nanoseconds plotted for the varying weights of the cost functions. The blocking times of the tasks in nanoseconds plotted for the varying weights of the cost functions is shown in Figure 17. The figures show that the response times for the tasks, especially the end tasks which are task 5 and 3 is minimum for P=1, B=0. However this case has the highest blocking time it is not optimized and the tasks which share resources have been assigned to different cores and run in parallel.

For the case when B=1, P=0, the figures show the inverse behaviour where the total blocking time of the tasks is minimum while the response time is the highest as all the tasks that share the resource have been assigned to core 0. It is to be noted that the blocking time of task 3 is always 0 as it does not have any semaphore.

Changing the weights to P=0.5, B=0.5 does not change the way the tasks are assigned to the cores when compared P=1, B=0, hence the response times and blocking times remain the same. Changing the weights to P=0.25, B=0.75, the core assignment changes. Now an increase is seen in the response times as task 2 which could have run in parallel has moved to core 0. Correspondingly a decrease is seen in the blocking time as the task 2 shares some common resources with other tasks in core 0.

Varying the weights value more, a different partitioning of the tasks is observed when P=0.2, B=0.8. This case continues with the trend that decreasing the parallelization weight tends to increase the response time. Also the blocking times of the tasks are further reduced in this new partitioning. However the total blocking time is still greater than the case when B=1.

These results are in agreement with our hypothesis of combining the optimization strategies using the cost functions. The weights P=0.25, B=0.75 and P=0.2, B=0.8 do not offer the best response time or the minimum blocking time. But they offer a better compromise where both the response time and the blocking time are close to the best possible values.
Figure 16: Task response times for Test2

Figure 17: Task blocking times for Test2
**Test 3: Validate that Cost Based partitioning algorithm generates a partition with the best compromise by doing an exhaustive search in all partitions**

The task set used for this test case consists of 5 tasks with the period equals to 200ms. The tasks utilizations and execution times are shown in Table 14. The task graph shown in Table 15 shows the inter-process communications between the tasks using message queues as well as the semaphores that the tasks use.

<table>
<thead>
<tr>
<th>Utilization (U_i)</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>T_5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25%</td>
<td>20%</td>
<td>15%</td>
<td>19%</td>
<td>13%</td>
</tr>
<tr>
<td>Execution time (C_i)</td>
<td>50 ms</td>
<td>40 ms</td>
<td>30 ms</td>
<td>38 ms</td>
<td>26 ms</td>
</tr>
</tbody>
</table>

*Table 14: Utilizations of task for Test3*

<table>
<thead>
<tr>
<th>sem1</th>
<th>sem2</th>
<th>sem3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T_2</td>
<td>-</td>
<td>1000 us</td>
</tr>
<tr>
<td>T_3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T_4</td>
<td>2000 us</td>
<td>1000 us</td>
</tr>
<tr>
<td>T_5</td>
<td>2000 us</td>
<td>-</td>
</tr>
</tbody>
</table>

*Table 15: Task graph and semaphores for Test3*

The aim of this test case is to verify that the partition generated by the Cost Based Partitioning algorithm is indeed the best compromise. To prove this, all the possible partitions are generated for the task set shown in Table 17 by assigning them to 2 shielded CPUs. The 5 tasks can be assigned to 2 CPUs in $2^5 = 32$ ways. However, it is assumed that CPU 1 and 2 are identical and so the unique number of partitions are $32/2 = 16$ which are listed in Table 15.

All the possible partitions are then run on the implemented test framework and their response times and total blocking times are also recorded in the Table 16. Then the task set is run with various partitioning algorithms to identify partitions generated by the partitioning algorithms. These partitions are the highlighted rows in Table 16.

As expected, the measured value of the response time is minimum for the Parallelization algorithm while the Blocking Time algorithm minimizes the total time the tasks are blocked when trying to lock a semaphore. When the Parallelization and Blocking time algorithms are combined using the Cost Based partitioning with weights of 0.3 and 0.7 respectively the partition in Case 10, Table 16 is generated. It is observed that the response time of this partition is more than the minimum response time of Case 12 and the blocking time is greater than the minimum blocking time of Case 1. However, by comparing the values in Table 16, it is seen that Case 10 has the best compromise in minimizing the values of both Response time and Blocking time.
<table>
<thead>
<tr>
<th>Case Number</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 1 Utilization</th>
<th>CPU 2 Utilization</th>
<th>Response time (ns)</th>
<th>Total Blocking Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\tau_1$</td>
<td>$\tau_2, \tau_3, \tau_4, \tau_5$</td>
<td>25</td>
<td>67</td>
<td>191,194,656</td>
<td>7,033,972</td>
</tr>
<tr>
<td>2</td>
<td>$\tau_2$</td>
<td>$\tau_1, \tau_3, \tau_4, \tau_5$</td>
<td>20</td>
<td>72</td>
<td>152,170,566</td>
<td>7,049,075</td>
</tr>
<tr>
<td>3</td>
<td>$\tau_3$</td>
<td>$\tau_1, \tau_2, \tau_4, \tau_5$</td>
<td>15</td>
<td>77</td>
<td>163,182,050</td>
<td>7,045,600</td>
</tr>
<tr>
<td>4</td>
<td>$\tau_4$</td>
<td>$\tau_1, \tau_2, \tau_3, \tau_5$</td>
<td>19</td>
<td>73</td>
<td>152,632,706</td>
<td>7,048,439</td>
</tr>
<tr>
<td>5</td>
<td>$\tau_5$</td>
<td>$\tau_1, \tau_2, \tau_3, \tau_4$</td>
<td>13</td>
<td>79</td>
<td>189,959,718</td>
<td>7,032,124</td>
</tr>
<tr>
<td>6</td>
<td>$\tau_1, \tau_2$</td>
<td>$\tau_3, \tau_4, \tau_5$</td>
<td>45</td>
<td>76</td>
<td>152,897,157</td>
<td>7,047,833</td>
</tr>
<tr>
<td>7</td>
<td>$\tau_1, \tau_3$</td>
<td>$\tau_2, \tau_4, \tau_5$</td>
<td>40</td>
<td>81</td>
<td>159,214,233</td>
<td>7,041,759</td>
</tr>
<tr>
<td>8</td>
<td>$\tau_1, \tau_4$</td>
<td>$\tau_2, \tau_3, \tau_5$</td>
<td>44</td>
<td>77</td>
<td>152,204,393</td>
<td>7,045,736</td>
</tr>
<tr>
<td>9</td>
<td>$\tau_1, \tau_5$</td>
<td>$\tau_2, \tau_3, \tau_4$</td>
<td>38</td>
<td>83</td>
<td>165,189,355</td>
<td>7,046,937</td>
</tr>
<tr>
<td>10</td>
<td>$\tau_2, \tau_3$</td>
<td>$\tau_1, \tau_4, \tau_5$</td>
<td>35</td>
<td>86</td>
<td>125,627,292</td>
<td>7,060,347</td>
</tr>
<tr>
<td>11</td>
<td>$\tau_2, \tau_4$</td>
<td>$\tau_1, \tau_3, \tau_5$</td>
<td>39</td>
<td>82</td>
<td>159,191,841</td>
<td>7,044,505</td>
</tr>
<tr>
<td>12</td>
<td>$\tau_2, \tau_5$</td>
<td>$\tau_1, \tau_3, \tau_4$</td>
<td>33</td>
<td>88</td>
<td>124,177,698</td>
<td>7,061,303</td>
</tr>
<tr>
<td>13</td>
<td>$\tau_3, \tau_4$</td>
<td>$\tau_1, \tau_2, \tau_5$</td>
<td>34</td>
<td>87</td>
<td>124,553,875</td>
<td>7,056,510</td>
</tr>
<tr>
<td>14</td>
<td>$\tau_3, \tau_5$</td>
<td>$\tau_1, \tau_2, \tau_4$</td>
<td>28</td>
<td>93</td>
<td>159,206,949</td>
<td>7,041,964</td>
</tr>
<tr>
<td>15</td>
<td>$\tau_4, \tau_5$</td>
<td>$\tau_1, \tau_2, \tau_3$</td>
<td>32</td>
<td>89</td>
<td>125,571,126</td>
<td>7,056,231</td>
</tr>
<tr>
<td>16</td>
<td>$\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$</td>
<td>-</td>
<td>92</td>
<td>0</td>
<td>191,226,701</td>
<td>7,033,105</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Case Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Time Algorithm</td>
<td>1</td>
</tr>
<tr>
<td>Parallelization Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>Best Fit Algorithm</td>
<td>16</td>
</tr>
<tr>
<td>Worst Fit Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>Cost Based Algorithm (P = 0.3, B = 0.7)</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 16: All possible partitions for Test3*
6.2 VxWorks

In VxWorks running on the Zynq7000 board the POSIX clock has a resolution of 1ms while in VxWorks running on x86 desktop it has a resolution of 10 µs. So it was not possible to get good results from the logging framework as values, like semaphore latency that are much smaller than the resolution, are reported as zero. Hence the VxWorks workbench system viewer is used to get the information about tasks and their interactions.

The Test Setup

The test consists of 6 tasks, 3 semaphores and 5 message queues. The period of each task is 200 ms and the execution times for the tasks are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
<th>$\tau_5$</th>
<th>$\tau_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization ($U_i$)</td>
<td>15%</td>
<td>17%</td>
<td>13%</td>
<td>12%</td>
<td>13%</td>
<td>15%</td>
</tr>
<tr>
<td>Execution time ($C_i$)</td>
<td>30 ms</td>
<td>34 ms</td>
<td>26 ms</td>
<td>24 ms</td>
<td>26 ms</td>
<td>30 ms</td>
</tr>
</tbody>
</table>

*Table 17: Task utilizations for Test on VxWorks*

The task graph for the tasks is shown below in Table 18 which shows the inter-process communication between the tasks using message queues. The table on the right shows the semaphores the task use along with the time each task takes the semaphore.

<table>
<thead>
<tr>
<th>Task</th>
<th>sem1</th>
<th>sem2</th>
<th>sem3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>-</td>
<td>200 us</td>
<td>100 us</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>200 us</td>
<td>100 us</td>
<td>-</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>100 us</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\tau_6$</td>
<td>-</td>
<td>-</td>
<td>100 us</td>
</tr>
</tbody>
</table>

*Table 18: Task graph and semaphores for Test on VxWorks*

The tasks are to be partitioned among two shielded cores. Figure 18 shows the starting procedure of the tasks. The tasks start and then get blocked when they try to lock the semaphore. The point where they request the semaphore is denoted by the red flag. After all tasks have started, the tMain task releases the semaphore and the tasks start executing.
**TEST 1: Tasks can be run as RTPs or Kernel tasks.**

In VxWorks the tasks can be run as either RTPs or kernel tasks. In both these tests, SMP is enabled and the tasks have two shielded cores available to execute. The tasks are not assigned to any core and hence can migrate from one core to another.

Figure 19 shows the tasks in the test case described above being run as RTP. The RTPs are denoted as .vxe files and each of them runs as a separate process.

Figure 20 shows the tasks executing as kernel tasks. Note the difference from RTPs that all the kernel tasks execute in the same process and hence the same memory space.
TEST2: Tasks assigned using Best Fit Algorithm:

In this test the tasks are run as kernel tasks on shielded cores. As the total utilization of all the tasks is 85%, all the tasks are assigned to same core (core 2). As shown in Figure 21, the tasks execute one after another and the total response time of the tasks is the sum of execution times of all tasks. The core graph shows that the core 2 utilization peaks to 100% when the tasks are running and goes to 0% during sleep. Hence the average utilization turns out to be 85%. Also, it can be seen in the Figure 21 that the other OS related tasks are executing in core 0 which has a utilization of 32%.

TEST3: Tasks assigned using Worst Fit Algorithm:

The worst fit algorithm tries to distribute the load among the shielded cores which are cores 2 and 3. As shown in Figure 22, the tasks 1, 5 and 6 are allocated to core 3 and task 2, 3, 4 are allocated to core 2. This results in core 3 having a utilization of (15 + 13 +15) % = 43% while core 2 having a utilization of (17 + 13 + 12) % = 42%. Hence the load is distributed in the best possible manner among the two cores.
TEST4: Task assigned using Cost Based Partitioning Algorithm with Parallelization cost function:

The parallelization cost function tries to minimize the total response time of the tasks by running the maximum tasks in parallel. The tasks 1, 2 and 3 are assigned to core 2 while task 4, 5 and 6 are assigned to core 3. The Figure 23 shows that this reduces the task response time to a minimum of 90ms. The core usage graphs show that the cores are busy and idle at the same time because of trying to run tasks parallelly.

---

Figure 22: System Viewer showing the kernel tasks execution Worst Fit Algorithm

Figure 23: System Viewer showing the kernel tasks execution and cores for Parallelization cost function
Chapter 7
Conclusions

The objective of this thesis was to study and develop partitioning strategies and measure their real-time performance characteristics. To this end, we first created a framework to specify real time systems. Then a synthetic load generation tool is developed which based on the real time system specifications in the XML, creates the tasks and executes them. The tool is developed both for Linux and VxWorks. As shown in the results, the synthetic load generation tool successfully creates the tasks both in Linux and VxWorks and is able to execute them as separate process in Linux and as RTP or Kernel tasks in VxWorks.

In order to compare and contrast the results of the novel algorithms developed, some basic partitioning algorithms like Best-fit and Worst-fit are implemented which provide a baseline for measurement. Most common partitioning algorithms try to optimize one or the other real time characteristics. The Cost based partitioning algorithm provided a base for developing a partitioning algorithm which can combine multiple optimization criteria. A modified Blocking time cost function proposed which complies with the modified cost based partitioning algorithm.

To test the premise of combining multiple optimization strategies in the Cost based partitioning algorithm, a novel Parallelization cost function is developed which tries to minimize the response times of the tasks. As evident from the comparison results between various partitioning algorithms in the results, the Parallelization cost function always achieves the best response time.

Finally the Parallelization cost function and the Blocking time cost functions are combined in the cost based partitioning algorithm with varying weights assigned to them. And as expected the algorithm finds a partition for which even though blocking time and the response time are not minimum but they are both close to the minimum values. Hence this partition offers the best compromise in which the response times and blocking times are close to the minimum values achieved using only Parallelization or Blocking time cost function.
Chapter 8
Challenges faced

- **Resolution of the POSIX Clock in VxWorks:** Throughout the whole test framework and task implementation the POSIX clocks are used to measure the time for logging purposes. However the resolution of the POSIX clock varies depending on the hardware and the OS. In VxWorks running on the ZC706 board the POSIX clock has a resolution of 1ms while in on x86 desktop it has a resolution of 10 µs. So it was impossible to get good results from the logging as many values like semaphore latency which are much smaller than the resolution are reported as zero.

  On ZC706, the timestamp timer provides a high resolution of 10µs. However the API to get the timestamp value is only available to kernel tasks. To solve this, a function was written which calculates the time from the timestamp timer and this function is registered as a custom system call. So the RTP can call this function using the syscall API. However this solution also runs into problems as the clocks for the timestamp counter and the POSIX clock are not synchronized. The duration of 1 sec varies on both clocks. This leads to various errors as the POSIX clock was used for sleeping to maintain the periodic behaviour while the timestamp counter was used for logging. An ideal solution would be to configure the POSIX clock in the VxWorks kernel to the timestamp counter or other high precision timers present. However due to time constraints it was not done.

- **Real-time Linux on ZC706 board:** Xilinx provides the kernel source code for the ZC706 through git repositories. However the kernel provided is not real time and needs to be rebuild with the real time patch. The kernel versions for Xilinx and the patches are not same. After many trial and errors in patching the kernels, it was decided to use the kernel 3.12 from the Xilinx website. This is rebuild with the real time patch for kernel 3.12.

- **System Clock on ZC706 on Real-time Linux:** The default system clock on ZC706 is 100Hz, which is too slow for real time measurements like latency and jitter. Hence the challenge was to use the ARM Cortex A9 global timer on ZC706 as the system clock. To enable the support for this clock, it is added to the device tree of the ZC706. The device tree file is then compiled to generate the binary. Then the Linux kernel is compiled with the CONFIG_ARM_GLOBAL_TIMER parameter as set. Once the kernel is built the clock can be seen as arm_global_timer.o file in linux-xlnx/drivers/clocksource. This clock now has a resolution of 333MHz which is good enough for the measurements.

- **Synchronizing the task start:** In order to achieve worst case performance the tasks need to start at the same time. This is achieved by the synchronization mechanisms discussed in the implementation. Synchronizing tasks using shared memory was tested which introduce a lot of overhead and complexity. Thus, signalling was chosen as an alternative which is quite simpler for addressing all the tasks together. However, it became an issue again in VxWorks where signalling was not available. Hence, semaphores were used to synchronize the start of the tasks.
Chapter 9
Future Work

Although the test framework works and the test results look promising there is always room for improvements. Following are some features which due to time constraints could not be implemented:

- Add support for more inter process communication methods apart from message queues, like shared memory, pipes, memory mapped files etc.
- For now only binary semaphores have been implemented. Next step could be to add support for counting semaphores as well.
- Currently in Linux tasks are launched as separate processes which run in separate memory space. A support can be added for threads all of which run in the same memory space and compare the performance to process. Further to make the test framework and XML more generic, the implementation could be that a test case can have multiple processes each with multiple threads inside it.
- In VxWorks currently the test case can have tasks as only RTPs or Kernel tasks. A next step could be to have both kernel tasks and RTPs in the same test case.
- Current XML design does not provide any provision to specify the interrupts in the system. This can be added so that the interrupts and their frequency are specified in the XML and the interrupt response time is measured for different platforms running different loads.
- Currently the partitioning algorithms just check if the utilization of each core is less than 100%. This can be improved to check if the partitions are schedulable. Also in the cost based partitioning algorithms when the task is assigned to a core, the only check performed is the utilization is less than 100%. This can be improved by checking the schedulability of each core when the task is assigned to the core with minimum cost. If some task becomes unschedulable the task is assigned to the core with the next minimum cost. If no schedulable assignment can be found the task set it not schedulable.
Chapter 10

References


3. Running AMP, SMP or BMP Mode for Multicore Embedded Systems. [Online] [Cited: 03 11 2014.]

4. RT PREEMPT HOWTO. [Online] [Cited: 03 11 2014.]


