Abstract

The most significant challenge in using platform based designs is to optimally map a set of communicating tasks on the available set of resources, which are present in the form of processing units and communication channels. Additional levels of challenges arise in multicore platforms when considering shared resources that are competed for access by multiple tasks, mapped on possibly different processing units. Thus, subtle changes in the choice of scheduling algorithms, resource access mechanisms and blocking methods have significant effects on the overall behavior of the system. This thesis proposes a simulation based approach, in the form of an analysis framework, to visualize the effects of shared resources on the overall behavior of the system. The early analysis is done before the cycle accurate system design, and thus disallowing any late changes in the system design flow. Scheduling algorithms, resource access mechanisms and blocking methods are treated as orthogonal concerns, hence allowing much greater flexibility to the system architect. The proposed framework allows semaphore-based shared resources as well as nested resource accesses. Moreover, it also allows the addition of new scheduling algorithms, resource access mechanisms, blocking methods and even new orthogonal concerns. The proposed framework is evaluated in terms of expressiveness, usability and extensibility. The main limitation of the proposed framework is the consideration of only the type of shared resources which are located outside any processing unit.
Acknowledgements

This thesis work was performed at OFFIS (Institute of Information Technology, Oldenburg, Germany), and was done in the context of EU-funded COMPLEX project. I would like to thank many people who have helped me in my thesis as well as in my studies. I wish to thank my (immediate) supervisor Philipp for his continuous support throughout the thesis project. I also want to thank all other colleagues and peers at Hardware/Software Design Methodology group at OFFIS (especially, Maher and Kim, for their continuous encouragement). I also want to thank my examiner, Ingo, for his many useful comments during the preparation of the report, and for his eternally fun-filled lectures at KTH. Moreover, I also wish to thank all my friends (both in Oldenburg and Stockholm) for their encouragement, and for the good time that I have spent with them. My parents, brother, grand parents and extended family also deserve a huge thanks for always putting trust in me. This milestone would not have been possible without their support and prayers. I also owe a thanks to Sweden and especially KTH for giving me opportunity to pursue my masters studies here. Most importantly, I wish to thank God for enabling me to finish this daunting task and also for everything else. I dedicate this thesis work to my grandfather, who has always remained my patron.
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX</td>
<td>COdesign and power Management in PLatform-based design space EXploration</td>
</tr>
<tr>
<td>DMA</td>
<td>Deadline Monotonic Algorithm</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>EPDF</td>
<td>Earliest Pseudo Deadline First</td>
</tr>
<tr>
<td>ERA</td>
<td>External Resource access Abstraction</td>
</tr>
<tr>
<td>EU</td>
<td>European Union</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FMLP</td>
<td>Flexible Multiprocessor Locking Protocol</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GP</td>
<td>Global Partitioning</td>
</tr>
<tr>
<td>G-EDF</td>
<td>Global Earliest Deadline First</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IRA</td>
<td>Internal Resource access Abstraction</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>LST</td>
<td>Least Slack Time first</td>
</tr>
<tr>
<td>MESH</td>
<td>Modeling Environment for Software and Hardware</td>
</tr>
<tr>
<td>MPCP</td>
<td>Multiprocessor Priority Ceiling Protocol</td>
</tr>
<tr>
<td>MSRP</td>
<td>Multiprocessor Stack-based Resource Protocol</td>
</tr>
<tr>
<td>PCP</td>
<td>Priority Ceiling Protocol</td>
</tr>
<tr>
<td>P-EDF</td>
<td>Partitioned Earliest Deadline First</td>
</tr>
<tr>
<td>RMA</td>
<td>Rate Monotonic Algorithm</td>
</tr>
<tr>
<td>RMA</td>
<td>Resource Management Abstraction</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-time Operating System</td>
</tr>
<tr>
<td>SRP</td>
<td>Stack-based Resource Protocol</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>tCFSM</td>
<td>timed Communicating Finite State Machine</td>
</tr>
<tr>
<td>TLM</td>
<td>Transactional Level Modeling</td>
</tr>
<tr>
<td>VPM</td>
<td>Virtual Processor Model</td>
</tr>
<tr>
<td>VPU</td>
<td>Virtual Processing Unit</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
</tbody>
</table>
## Contents

1. **Introduction** ................................................................. 9  
   1.1 Motivation ......................................................................... 9  
   1.2 Problem Statement .............................................................. 10  
   1.3 Contributions ...................................................................... 11  
   1.4 Organization ........................................................................ 13  

2. **Background** ............................................................................... 16  
   2.1 System Level Design Flow ...................................................... 16  
      2.1.1 System Specification ....................................................... 16  
      2.1.2 Architecture Allocation ................................................... 17  
      2.1.3 HW/SW Partitioning, Mapping and Scheduling .................. 19  
      2.1.4 Hardware/Software Synthesis .......................................... 19  
      2.1.5 System-Level Simulation ................................................ 19  
   2.2 Real-Time Model and Associated Terminology ......................... 19  
   2.3 Classification of Scheduling Algorithms ................................. 21  
      2.3.1 Fixed Priority Algorithms .............................................. 21  
      2.3.2 Task-Level Dynamic Priority Algorithms ......................... 21  
      2.3.3 Job-level Dynamic Priority Algorithms ............................ 21  
   2.4 Scheduling Approaches for Multiprocessors ............................. 22  
      2.4.1 Local/Partitioned Scheduling .......................................... 22  
      2.4.2 Global Scheduling .......................................................... 22  
      2.4.3 Hybrid Scheduling ........................................................ 24  
   2.5 Uniprocessor Approaches for Shared-Resource Access ............... 25  
      2.5.1 Resource Sharing and Synchronization Algorithms .......... 25  
      2.5.2 Basic Priority Inheritance Protocol ................................. 26  
      2.5.3 Priority Ceiling Protocol ............................................... 27  
      2.5.4 Stack-based Resource Policy Protocol (SRP) .................... 29  
      2.5.5 Pros and Cons .............................................................. 33  
   2.6 Multiprocessor-based Approaches for Shared Resource Access .... 33  
      2.6.1 Multi-processor Priority Ceiling Protocol (MPCP) .......... 34  
      2.6.2 Multiprocessor Stack Resource Policy Protocol (MSRP) .... 35  
      2.6.3 A Flexible Real-Time Locking Protocol for Multiprocessors 35  
   2.7 Alternative and Contemporary Approaches of Shared Resource Access Modeling .......................................................... 37
## 2.7.1 An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems
- Page 37

## 2.7.2 Shared Resources High-Level Modeling in Embedded Systems Using Virtual Nodes
- Page 38

## 2.7.3 Shared resource access attributes for high-level contention models
- Page 38

## 2.7.4 MESH based Hybrid Simulation/Analytical Approach for Modeling Shared Resource Contention
- Page 39

## 3 Functional Requirements of the Framework

### 3.1 Blocking Behaviors
- Bus Waiting
- Suspension-based Approach
- Page 41

### 3.2 Resource Access Mechanisms
- Non-preemptive Critical Section
- Priority Inheritance Approach
- Multiprocessor Priority Ceiling Protocol (MPCP)
- Multiprocessor Stack-based Resource Policy Protocol (MSRP)
- Flexible Multiprocessor Locking Protocol (FMLP)
- Page 42

### 3.3 Scheduling Algorithms
- Partitioned-Earliest Deadline First (P-EDF)
- Global Earliest Deadline First (G-EDF)
- Page 46

## 4 Design of the Framework

### 4.1 Task
- Page 49

### 4.2 Scheduler
- Page 50

### 4.3 Internal Resource Access Abstraction
- Page 51

### 4.4 External Resource Access Abstraction
- Page 52

### 4.5 Resource Management Abstraction
- Page 53

### 4.6 Resource
- Page 54

### 4.7 Request and Response
- Page 54

## 5 The Shared Resource Analysis Framework

### 5.1 Abstract Task Modeling and Mapping in SystemC
- Task
- Virtual Processing Unit (VPU)
- Task Manager
- Page 56

### 5.2 Design and Implementation of the Overall Framework
- Task
- Priority Hierarchy
- Request
- Shared Resource Access
- Internal Resource Access Abstraction
- External Resource Access Abstraction
- Resource Management Abstraction
- Page 58
6 Evaluation and Discussion

6.1 Features .................................................. 71
6.2 Limitations ............................................... 72
6.3 Metrics ..................................................... 72
6.4 Experimentation .......................................... 73
  6.4.1 Experimental Setup (for Experiments 1-8) .......... 74
  6.4.2 Experiment 1 ............................................. 75
  6.4.3 Experiment 2 ............................................. 78
  6.4.4 Experiment 3 ............................................. 80
  6.4.5 Experiment 4 ............................................. 83
  6.4.6 Experiment 5 ............................................. 86
  6.4.7 Experiment 6 ............................................. 88
  6.4.8 Experiment 7 ............................................. 90
  6.4.9 Experiment 8 ............................................. 93
  6.4.10 Experimental Setup (for Experiments 9 and 10) .. 98
  6.4.11 Experiment 9 ........................................... 103
  6.4.12 Experiment 10 ......................................... 105
6.5 Usability .................................................. 106
  6.5.1 Blocking methods and Resource Access Mechanisms . 107
  6.5.2 Scheduling .............................................. 107
  6.5.3 Number of Instances of a Resource .................. 107
  6.5.4 Mapping ................................................. 107
6.6 Extensibility .............................................. 108
  6.6.1 Scheduling Algorithm .................................. 108
  6.6.2 Resource Access Mechanism ............................ 108
  6.6.3 Task Set ................................................ 110
  6.6.4 Blocking Method ........................................ 110
  6.6.5 Shared Resource Arbitration Policy .................... 111
7 Conclusion and Future Directions ......................... 112
7.1 Conclusion ................................................. 112
  7.1.1 Limitations ............................................. 113
  7.1.2 Summary of thesis ..................................... 113
7.2 Future Directions ......................................... 113
  7.2.1 Flexible Shared Resources ............................. 114
  7.2.2 More Sophisticated Resource Access Mechanisms and Blocking Methods ... 114
  7.2.3 New Orthogonal Concerns .............................. 114
  7.2.4 Additional Scheduling Algorithms .................... 114
  7.2.5 Modeling of Delays .................................... 114
  7.2.6 Validation in Real World Scenarios ................. 115
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A Actual Simulation Results of Experiments</td>
<td>116</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Example of a busy-waiting task ........................................ 10
1.2 Example of a suspension-based waiting task ......................... 11
1.3 Orthogonality of Concerns ............................................. 13
1.4 Abstract Overall Model of the Framework .......................... 14
1.5 Abstract Internal view of a Virtual Processing Unit ............... 15
1.6 Abstract Resource Space Abstraction ................................ 15

2.1 System Level Design Flow .............................................. 17
2.2 System-Level Specification ............................................ 18
2.3 Architecture Allocation ................................................ 18
2.4 Local/Partitioned Scheduling Approach .............................. 23
2.5 Global Scheduling Approach .......................................... 24
2.6 Example of P-fair windows ............................................ 25
2.7 Hybrid Scheduling Approach ........................................... 26
2.8 Basic Priority Inheritance Protocol .................................. 27
2.9 Priority Ceiling Protocol .............................................. 30
2.10 Stack-based Resource Protocol ...................................... 32

4.1 Overall System Interaction Diagram ................................ 50
4.2 Task Class Diagram ................................................... 51
4.3 Scheduler Class Diagram .............................................. 52
4.4 Internal Resource Access Abstraction (IRA) Class Diagram .... 53
4.5 External Resource Access Abstraction (ERA) Class Diagram ... 53
4.6 Resource Management Abstraction (RMA) Class Diagram ....... 54
4.7 Resource Class Diagram ............................................... 55
4.8 Request and Response Class Diagram ............................... 55

5.1 State diagram of an abstract task ................................... 57
5.2 Overall Frame Design .................................................. 59
5.3 Two VPU’s and two Resource Management Abstraction units ... 60
5.4 Functionality of framework components inside a Virtual Process-
ing Unit (VPU) .............................................................. 61
5.5 Functionality of Resource Management Abstraction (RMA) ... 62
5.6 Framework priority hierarchy ........................................ 63
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Request Structure</td>
<td>63</td>
</tr>
<tr>
<td>5.8</td>
<td>Response structure</td>
<td>67</td>
</tr>
<tr>
<td>5.9</td>
<td>Flow chart showing the round robin algorithm</td>
<td>69</td>
</tr>
<tr>
<td>5.10</td>
<td>Flow chart showing the update part of above round-robin algorithm</td>
<td>70</td>
</tr>
<tr>
<td>6.1</td>
<td>Task states legend diagram</td>
<td>74</td>
</tr>
<tr>
<td>6.2</td>
<td>Task structure diagram for experiments 1-8</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>System diagram for experiments 1-8</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>Simulation results for VPU1 in experiment 1</td>
<td>78</td>
</tr>
<tr>
<td>6.5</td>
<td>Simulation results for VPU2 in experiment 1</td>
<td>79</td>
</tr>
<tr>
<td>6.6</td>
<td>Simulation results for VPU1 in experiment 2</td>
<td>80</td>
</tr>
<tr>
<td>6.7</td>
<td>Simulation results for VPU2 in experiment 2</td>
<td>81</td>
</tr>
<tr>
<td>6.8</td>
<td>Simulation results for VPU1 in experiment 3</td>
<td>82</td>
</tr>
<tr>
<td>6.9</td>
<td>Simulation results for VPU2 in experiment 3</td>
<td>83</td>
</tr>
<tr>
<td>6.10</td>
<td>Simulation results for VPU1 in experiment 4</td>
<td>84</td>
</tr>
<tr>
<td>6.11</td>
<td>Simulation results for VPU2 in experiment 4</td>
<td>85</td>
</tr>
<tr>
<td>6.12</td>
<td>Simulation results for VPU1 in experiment 5</td>
<td>87</td>
</tr>
<tr>
<td>6.13</td>
<td>Simulation results for VPU2 in experiment 5</td>
<td>88</td>
</tr>
<tr>
<td>6.14</td>
<td>Simulation results for VPU1 in experiment 6</td>
<td>89</td>
</tr>
<tr>
<td>6.15</td>
<td>Simulation results for VPU2 in experiment 6</td>
<td>90</td>
</tr>
<tr>
<td>6.16</td>
<td>Simulation results for VPU1 in experiment 7</td>
<td>91</td>
</tr>
<tr>
<td>6.17</td>
<td>Simulation results for VPU2 in experiment 7</td>
<td>92</td>
</tr>
<tr>
<td>6.18</td>
<td>Simulation results for VPU1 in experiment 8</td>
<td>94</td>
</tr>
<tr>
<td>6.19</td>
<td>Simulation results for VPU2 in experiment 8</td>
<td>95</td>
</tr>
<tr>
<td>6.20</td>
<td>Task structure for first task on VPU1 in experiments 9-10</td>
<td>99</td>
</tr>
<tr>
<td>6.21</td>
<td>Task structure for second task on VPU1 in experiments 9-10</td>
<td>100</td>
</tr>
<tr>
<td>6.22</td>
<td>Task structure for tasks on VPU2 and VPU3 in experiments 9-10</td>
<td>102</td>
</tr>
<tr>
<td>6.23</td>
<td>System diagram for experiments 9-10</td>
<td>103</td>
</tr>
<tr>
<td>6.24</td>
<td>Simulation results for VPU1 in experiment 9</td>
<td>104</td>
</tr>
<tr>
<td>6.25</td>
<td>Simulation results for VPU2 and VPU3 in experiments 9 and 10</td>
<td>105</td>
</tr>
<tr>
<td>6.26</td>
<td>Simulation results for VPU1 in experiment 10</td>
<td>106</td>
</tr>
<tr>
<td>6.27</td>
<td>Class diagram showing scheduler extensibility</td>
<td>108</td>
</tr>
<tr>
<td>6.28</td>
<td>Points of interest when extending the framework with a new resource access mechanism</td>
<td>109</td>
</tr>
</tbody>
</table>

#### A. Results

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Results of VPU1 in experiment 1</td>
<td>117</td>
</tr>
<tr>
<td>A.2</td>
<td>Results of VPU2 in experiment 1</td>
<td>118</td>
</tr>
<tr>
<td>A.3</td>
<td>Results of VPU1 in experiment 2</td>
<td>119</td>
</tr>
<tr>
<td>A.4</td>
<td>Results of VPU2 in experiment 2</td>
<td>120</td>
</tr>
<tr>
<td>A.5</td>
<td>Results of VPU1 in experiment 3</td>
<td>121</td>
</tr>
<tr>
<td>A.6</td>
<td>Results of VPU2 in experiment 3</td>
<td>122</td>
</tr>
<tr>
<td>A.7</td>
<td>Results of VPU1 in experiment 4</td>
<td>123</td>
</tr>
<tr>
<td>A.8</td>
<td>Results of VPU2 in experiment 4</td>
<td>124</td>
</tr>
<tr>
<td>A.9</td>
<td>Results of VPU1 in experiment 5</td>
<td>125</td>
</tr>
<tr>
<td>A.10</td>
<td>Results of VPU2 in experiment 5</td>
<td>126</td>
</tr>
<tr>
<td>A.11</td>
<td>Results of VPU1 in experiment 6</td>
<td>127</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

A.12 Results of VPU2 in experiment6 . . . . . . . . . . . . . . . . . . 128
A.13 Results of VPU1 in experiment7 . . . . . . . . . . . . . . . . . . 129
A.14 Results of VPU2 in experiment7 . . . . . . . . . . . . . . . . . . 130
A.15 Results of VPU1 in experiment8 . . . . . . . . . . . . . . . . . . 131
A.16 Results of VPU2 in experiment8 . . . . . . . . . . . . . . . . . . 132
A.17 Results of VPU1 in experiment9 . . . . . . . . . . . . . . . . . . 133
A.18 Results of VPU2 and VPU3 in experiments 9 and 10 . . . . . . . 134
A.19 Results of VPU1 in experiment 10 . . . . . . . . . . . . . . . . . 135
### Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Example of tasks’ structure for experiments 1–8</td>
<td>76</td>
</tr>
<tr>
<td>6.2</td>
<td>Structure of the first task (T1) mapped on VPU1 in experiments 9 and 10</td>
<td>94</td>
</tr>
<tr>
<td>6.3</td>
<td>Structure of the second task (T2) mapped on VPU1 in experiments 9 and 10</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Structure of the tasks (T3 and T4) mapped on VPU’s 2 and 3 in experiments 9 and 10</td>
<td>101</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter gives the motivation for carrying out the thesis. It also explains the problem area, and enlists the contributions of the thesis. Furthermore, it explains the organization of the thesis and also gives the outline of the rest of the thesis document.

1.1 Motivation

The thermal and power limitations in uniprocessor systems caused the advent of multicore systems and subsequently multicore platforms. New challenges have also arrived with the advent of multicore platforms and increased performance capabilities. These new challenges include constraining the given platform to meet certain real-time requirements, and power and cost limitations.

Hardware as well as software development in platform-based designs is tightly coupled. Thus, during the design-space exploration phase, an early analysis of software and real-time operating system’s (RTOS’s) effects is also very necessary [14]. Such an early analysis will definitely help to see the overall system-level picture of the envisioned system’s performance.

In a typical system-level design flow, parallelism is extracted from a given application set and its parallel entities are mapped onto different cores of a multicore platform. The optimal mapping of these parallel entities onto the available platform resources is a challenging task. These parallel entities (also called tasks) often need to interact with each other and share certain resources. The questions regarding the interactions within the task set, execution order, and interdependences with regards to shared resources need to be answered at early stage in the overall design process. The answers to these questions can pose grave effects on different properties of the system, especially on the desired real-time requirements. Secondly, it is evident that subtle settings of shared resource mechanisms can have major impact on the overall system performance [7]. But often the currently used approaches are inflexible and allow little roam for the system architect. Therefore, frequently, the system architect is bound
to use only the default settings of different synchronization algorithms and accompanying scheduling algorithms.

Figures 1.1 and 1.2 illustrate a simple scenario when different access behaviors produce subtly different real-time results.

Figure 1.1: Example of a task set missing deadline due to busy-waiting based shared resource access: Tasks are defined as $T(\text{execution time, period})$ with $T_1(1,2)$, $T_2(1,6)$, $T_3(2,9)$ and $T_4(2,4)$. The resource is alternatively allocated to two competing tasks.

1.2 Problem Statement

As evidenced from the literature study in the following sections, shared resource modeling techniques at higher abstraction levels are still in their infancy, and only a limited amount of work is available on shared resource access modeling. Moreover, in most of the available work, the effects of shared resource accesses are only considered later in the design cycle, after the availability of cycle-accurate executable model. Furthermore, the available approaches virtually provide no flexibility to the system architects using those approaches. Thus, the system architect not only needs to visualize the effects of shared resources early in the design cycle, but also needs to have more fine-grained control and flexibility over the different attributes of scheduling algorithms and synchronization mechanisms.
Figure 1.2: Example of a task set using suspension-based shared resource access: Tasks are defined as T(execution time, period) with T1(1,2), T2(1,6), T3(2,9) and T4(2,4). The resource is alternatively allocated to two competing tasks.

1.3 Contributions

The aim of the thesis is to lay the groundwork for the early analysis of shared resource accesses, in multicore scenarios, in the form of a simulation-based analysis framework. This early analysis is to be done before the availability of cycle-accurate system design to reduce or eliminate the impact of any late system changes.

The proposed analysis framework builds on the concept of orthogonality of concerns. According to this concept, different elements of the system are treated as independently variable entities. As shown in Fig 1.3, the three main orthogonal elements considered in the framework include scheduling algorithms, blocking behaviors and resource-access mechanisms. The main idea of this is to give the system architect much greater flexibility in designing the system. This essentially means that the analysis of the performance of the system can be made when tasks scheduled through an arbitrary scheduling algorithm, having any blocking behavior, access a shared resource managed by any given access mechanism. The proposed framework is designed for all major resource access mechanisms and scheduling strategies. It even allows the addition of new orthogonal concerns (like the arbitration policy for the waiting tasks on a shared
A high-level view of the overall architecture of the system is shown in Fig. 1.4. Virtual Processing Units (VPU’s for short) model an abstract processing element. Shared resources are located externally in resource space abstraction layer. VPU’s make requests for shared resources or communicate with each other through an external communication abstraction layer. In the proposed simulation framework, concurrently executable pieces of code are to be modeled as tasks and mapped to various VPU’s. Task parameters as execution times, blocking behaviors, local scheduling parameters, resource access mechanisms etc are to be captured to model the shared resource access environment. The abstract internal details of the VPU are shown in Fig. 1.5. The local scheduler schedules the tasks on the VPU according to a given scheduling algorithm. While the tasks request shared resources through internal resource-access abstraction layer. The shared resources are in turn located in a resource space abstraction layer. The abstract model of a resource space abstraction layer is shown in Fig. 1.6, where a centralized or distributed resource management abstraction manages one or more shared resources.

This thesis work has been performed in the context of EU-funded COMPLEX (COdesign and power Management in PLatform-based design space EXploration) project [13]. In COMPLEX, a holistic framework for iteratively exploring the design space of embedded hardware/software systems is envisioned. A Virtual Processing Unit (VPU), an abstract task model, utility functions (to update the states of the tasks) and convenience ports are the components which were already available prior to this thesis work. The components of Internal Resource access Abstraction (IRA), External Resource access Abstraction (ERA), Resource Management Abstraction (RMA) and Resource are conceived and implemented as part of this thesis work. While Request and Resource are the data-structures that are also introduced in this work. Moreover, the abstract task model was modified and extended to fulfill the requirements of the framework. Furthermore, a framework-level priority scheme is introduced and maintained to ensure correct synchronization behavior and fulfill other functional requirements. Two scheduler base classes are also introduced to ensure that the framework works correctly by enforcing framework level priority scheme, when new schedulers are implemented by extending these base classes. The generic design of these components appears in chapter while the implemented feature-set is summarized in section 6.1 and explained in chapter

1these convenience ports are used to generate pseudo-TLM transactions in the Internal Resource access Abstraction (IRA) module.
2Interrupt Handler, Shared_Resource_Access and waiting data structure are the other sub-components of the framework. The already-available template of an interrupt handler block for a VPU is modified and extended in this thesis work.
CHAPTER 1. INTRODUCTION

Figure 1.3: Orthogonality of Concerns: Scheduling algorithm, resource access mechanism and blocking behavior are regarded as the three main orthogonal concerns.

1.4 Organization

The rest of the thesis is organized into the sections of Background, Functional Requirements of the Framework, Design of the Framework, The Shared Resource Analysis Framework, Evaluation and Discussion, Conclusion and Future Directions, Appendices, and Bibliography. Chapter 2 (Background) puts into perspective the contributions of this thesis. It explains system-level design flow, scheduling policies, resource allocation algorithms and shared resource access analysis approaches. The functional requirements of a generic and flexible framework are defined in chapter 3 (Functional Requirements of the Framework). An object-oriented design, originating from the functional requirements, is presented in chapter 4 (Design of the Framework). Chapter 5 (The Shared Resource Analysis Framework) details the actual implementation of the framework, and describes the approaches used to fulfill different requirements under the constraints of abstract task modeling techniques [18] and Platform Architect tool. Chapter 6 (Evaluation and Discussion) evaluates the proposed framework using metrics of expressiveness, usability and extensibility. It also enlists different simulation results achieved under different scenarios. The overall contributions are summarized in the chapter 7 (Conclusion and Future Directions). Moreover, possible future enhancements and directions for research are also enlisted in it. Furthermore, appendices contain supplementary information in the
Figure 1.4: Abstract Overall Model of the Framework: Different Virtual Processing Units (VPU’s) access the resources located in Resource Access Abstraction through External Communication Abstraction.

form of actual simulation results. Finally, different scholarly works used during the thesis are acknowledged in Bibliography section.
CHAPTER 1. INTRODUCTION

Figure 1.5: Abstract Internal view of a Virtual Processing Unit (VPU): Different tasks are mapped to the same VPU and are scheduled by a scheduler. They interact with the outside world through Internal Resource Access Abstraction module.

Figure 1.6: Abstract Resource Space Abstraction: Different resources are located as part of various Resource Management Abstraction modules.
Chapter 2

Background

This chapter gives a detailed and extended background into system design flow, required terminology, scheduling policies and resource access mechanisms. The first section describes the traditional system-level design flow, and explains where the proposed simulation approach fits. In the second section, much of the required terminology concerning real-time system model is introduced. This introduction is necessary to understand the following sections of scheduling algorithms and resource access protocols. Next, major scheduling algorithms for both uniprocessor systems as well as multiprocessor scenarios are briefly described. The fourth section gives a detailed overview of important resource access approaches, and also briefly discusses their relative merits and demerits. This detailed account is needed to put into perspective several design choices explained in later chapters. Finally, the last section gives a broad overview of different approaches used in evaluating the effects of shared resource accesses on the overall system performance.

2.1 System Level Design Flow

This section gives an overview of the typical steps followed during the system-level design flow and explains where the proposed framework fits in. The major steps of System-Level Design Flow typically include System Specification, Architecture Allocation, HW/SW Partitioning and Mapping, HW/SW Synthesis, and System-Level Simulation \[28\]. These steps are graphically shown in fig. 2.1. Each of these steps is briefly explained below:

2.1.1 System Specification

System Specification forms the first step in System-Level Design Flow. The main goal of this step is to capture the desired functionality of the system in an abstract manner. The requirements and constraints of the system can be captured in terms of system design languages or more abstract methods like
Finite State Machines (FSM’s) [21], Data and Control Flow Graphs [22] or Petri Nets [20]. More hardware-oriented aspects of the system can be captured in terms of design languages like SystemC [16], SpecC [11] or SystemVerilog [31]. On the other hand, more software-oriented requirements can be described in terms of software-oriented languages like C/C++ [15] or Java [12]. Furthermore, more time-critical aspects of the system can be captured in terms of synchronous software languages like Esterel [24]. Fig. 2.2 shows two ways of collecting system specification.

2.1.2 Architecture Allocation

Architecture Allocation step includes the allotment of appropriate components that optimally satisfy the required constraints. Fig 2.3 shows two examples of different architecture and communication structure allocations. In this step, numbers and types of computation elements as well as the communication links between them are specified. The notable ways to model a computational element (or its effects) in an abstract way include Virtual Processor Model (VPM), C-Source Back Annotation, Compiled Code Instruction Set Simulator and Virtual Processing Unit (VPU) [27], [18].

In VPM, a simplified model of the actual processor is used that tries to copy the actual behavior of the processor in terms of computational cycles and memory address behavior. While in C-Source Back Annotation method, the actual behavior of the code on a specific processor is used to specify a higher-abstraction model. On the other hand, Compiled Code Instruction Set Simulator includes a very accurate model of the processor, including models of components like cache...
as well as compiler-based code optimizations. The most abstract way to model a computational element is provided by a VPU [18]. It allows one or more tasks to execute on it according to their execution times, its own clock frequency and the local scheduling policy. Every VPU includes a Task Manager that acts as a lean run-time system, including the capability to execute tasks according to a scheduling policy.

Figure 2.3: Architecture Allocation
2.1.3 HW/SW Partitioning, Mapping and Scheduling

In this step, the functionality defined in the system is mapped to the available (hardware/software) resources. An important consideration during this step is which functionality needs to be implemented in hardware and which must be part of software. Normally, a trade-off must be made between the execution speeds provided by the hardware-based solution compared to the flexibility provided by the software. Other constraints such as power consumptions, physical footprint of the system and costs are also directly affected by the decisions made at this stage of design.

When more than one task is sharing the same processing element, a viable scheduling considering all tasks needs to be found. The purpose of scheduling is to ensure that under the proposed partitioning and mapping, all tasks are able to meet their requirements like deadlines. The proposed framework intends to give much more flexibility to the system architect at the scheduling step. It intends to allow the architect to view the effects of different sophisticated shared resource access mechanisms, along with scheduling approaches, with regards to the selected task partitioning and mapping.

2.1.4 Hardware/Software Synthesis

In hardware/software synthesis step, the abstract design of the overall system is transformed into the actual physical implementation. In the hardware synthesis step, the high-level description of the hardware is transformed to a gate-level implementation through the use of high-level and logical synthesis tools. While in software synthesis step, the high-level description of the software is transformed into the actual machine-code level implementation.

2.1.5 System-Level Simulation

The last step of system-level design is Simulation. In this step, the whole system is simulated at a suitable abstraction level. Its purpose is to verify whether the set of requirements and constraints set in the System Specification are being met or not. A trade-off between accuracy and simulation speed must be made while selecting an appropriate abstraction level. Several commercially available tools like Platform Architect, Cocentric System Studio, Platform Express and MaxSim provide modeling, design space exploration and simulation capabilities.

2.2 Real-Time Model and Associated Terminology

This section introduces the real-time system model by explaining its different building blocks. This terminology is also important to understand subsequent
sections on scheduling algorithms and synchronization mechanisms. This terminology is in part based on [29], [20], [2] and [8], and is presented in a paraphrased form.

**Job:** A set of sequential code that is submitted for scheduling to the scheduler, requiring a certain execution time and needing to be finished within a deadline is defined as a job. Let the ith job of a task be denoted by \( J(i) \).

**Execution Time:** The processing time consumed by a job is defined as its execution time. Let the execution time of a job be denoted by \( E(i) \).

**Period:** The time after which a job becomes ready for execution is defined as the period of the task. Let the period of a job be denoted by \( P(i) \).

**Deadline:** The time period during which a task is required to finish its execution is defined as its deadline. Let the deadline of a job be denoted by \( D(i) \).

**Types (Intensity):** Deadlines can be divided into the following categories with regards to the consequences of missing them:

- **Hard Deadlines:** Deadlines are defined as 'hard' if the consequences of missing them can be fatal.
- **Soft Deadlines:** Deadlines are defined as 'soft', if efforts are made to achieve them but the consequences of failing to achieve them are not fatal.

**Types (Period):**

- **Implicit:** If the deadline is equal to the period of a task, then it is defined as implicit.
- **Constrained:** If the deadline is less than or equal to the period, then it is called a constrained deadline.
- **Arbitrary:** If the deadline can have arbitrary relationship to its period, then it is defined as arbitrary.

**Task:** A set of related jobs that together perform some useful work is defined as the task. A task may impose some precedence constraints regarding the relative execution order of the jobs. Let the ith task in the system be denoted by \( T(i) \).

**Shared Resource:** A resource which is required by more than one task at a time is defined as a shared resource. A shared resource may allow different types of accesses to it. For example, a serialized shared resource may only be accessed exclusively, while a reader-writer shared resource may allow multiple readers at the same time. To protect the consistency of accesses, a shared resource access policy is needed.
Blocking behaviors: Due to the unavailability of a shared resource, a task may suffer from one of the following blocking behaviors:

- **Busy-Waiting** In busy-waiting, a task keeps on consuming CPU cycles according to the available scheduling policy until the requested resource becomes available.

- **Suspension** In suspension-based blocking, a task gives up its execution to the scheduler until the requested resource becomes available.

**Priority:** The priority of a task/job is a numerical value that defines the urgency with which it is scheduled.

### 2.3 Classification of Scheduling Algorithms

Based upon the variability of priorities, the scheduling algorithms are generally classified into the following categories:

#### 2.3.1 Fixed Priority Algorithms

In fixed priority algorithms, the priorities of tasks/jobs remain constant throughout the lifetime of the system. Examples include Rate Monotonic Algorithm (RMA), Deadline Monotonic Algorithm (DMA)

**RMA**

In RMA, tasks are assigned priorities based upon their periods. Higher priority is given to a task with a shorter period.

**DMA**

In DMA, tasks are assigned priorities based upon their deadlines. A task with a shorter deadline earns a higher priority.

#### 2.3.2 Task-Level Dynamic Priority Algorithms

In task-level dynamic priority algorithms, the priority of a task may change at different times, but the priority of a given job remains the same. E.g. Least Slack Time first (LST).

**EDF**

In EDF, task priorities are continually assigned based upon the relative deadlines of tasks. Tasks having same relative deadlines can be scheduled arbitrarily.

#### 2.3.3 Job-level Dynamic Priority Algorithms

In job-level dynamic priority algorithms, a given job may have different priorities at different points in time. E.g. Least Slack Time first (LST).
LST

LST algorithms continuously keep track of the already-done execution and the remaining execution to calculate the slack. Slack is defined as  
Deadline-(already-done execution)-(remaining execution)  
A job having a lower slack is assigned a higher priority by LST algorithm.

2.4 Scheduling Approaches for Multiprocessors

Scheduling approaches for multiprocessors can be classified based on the role of the scheduler. Depending upon whether a local scheduler is available or a global one, the scheduling algorithms can be categorized in the following three broad categories (based upon [9]):

2.4.1 Local/Partitioned Scheduling

In this approach, tasks are partitioned among processors, and each processor is scheduled using its own local scheduler. Different multiprocessors may be scheduled by the same scheduling algorithm or different ones. So, the multiprocessor scheduling problem essentially gets reduced to a uniprocessor based scheduling problem. Any uniprocessor-based scheduling algorithm like Rate Monotonic (RM), Deadline Monotonic (DM) or Earliest Deadline First (called P-EDF) can be applied to schedule the tasks. An advantage is that local scheduling approaches do not suffer from any penalties due to task or job migrations. Fig. 2.4 shows an example where each processor has a local scheduling entity.

2.4.2 Global Scheduling

In 'global scheduling' approach, only a single global scheduler controls the execution of tasks on all the processors. Tasks may start their execution on one processor and resume on the other one. Due to the migration of tasks in global scheduling approaches, additional communication loads are incurred. The advantages of global approaches include better utilization of available execution capacity on individual processors. Fig. 2.5 shows an example of global scheduling approach.

Global scheduling approaches can themselves be divided into the following categories:

1In local scheduling approaches, the shared resources can be part of any local processor or located outside them. Moreover, the tasks may request for access to shared resources located at remote processors. A more detailed description of various resource-access mechanisms, for various scheduling approaches, appears in the later sections.

2Similar to local scheduling approaches, shared resources can be located as part of any processor or located outside.
CHAPTER 2. BACKGROUND

Non P-fair Scheduling

The most important example of non P-fair scheduling algorithm is Global Earliest Deadline First (G-EDF) algorithm. In G-EDF, the priorities are continuously calculated based on the relative deadlines of the tasks, and tasks are scheduled preemptively in a priority driven manner.

Proportionate-fair (P-fair) Scheduling

In P-fair scheduling algorithms, each task’s execution is approximately uniform over a period of time. This uniform rate corresponds to the overall utilization of the task. Understanding of the following terminology is necessary to grasp the main idea of P-fair scheduling:

- **Quantum**: The amount of execution time corresponding to a single time unit is defined as a quantum.
- **Subtask**: Each part of a task which executes in a single quantum is defined as a subtask.
- **Pseudo Release**: Pseudo Release gives the time before which a subtask must not become ready for execution.
- **Pseudo Deadline**: Pseudo Deadline gives the time before which a subtask must complete its execution.
- **Pseudo Window**: Pseudo Window gives the time period during which a subtask should complete its execution and is defined as Pseudo Window = Pseudo Deadline - Pseudo Release. Fig. 2.6 shows an example of pseudo windows for a task with execution time \( E(i)=2 \) and period \( P(i)=9 \).
- **Prerequisites**: For P-fair scheduling, the following prerequisites need to be fulfilled:

Figure 2.4: Local/Partitioned Scheduling Approach
Execution time is allocated to tasks only as chunks of time or quanta. Moreover, the start and end times of quanta are synchronized across all the processors.

Execution times and periods of tasks must be adjusted to make them integer multiples of quantum sizes.

**Definition:** If in a feasible task system, every subtask is able to complete its execution within the pseudo window then all the job deadlines of the task system are met.

**Selection of Subtasks:** Subtasks are selected using earliest deadline first principle.

**Types:** A set of rules need to be used to break the ties between subtasks having the same deadlines and based upon those tie-breaking rules, different types of P-fair algorithms are defined. Examples of P-fair algorithms include PD2 [1]. Earliest Pseudo Deadline First (EPDF). EPDF chooses a simpler implementation and does not use any tie-breaking rules. Instead subtasks are tie-broken arbitrarily.

### 2.4.3 Hybrid Scheduling

In hybrid scheduling approaches, a combination of local and global partitioning approaches is used. These approaches try to mitigate some of the overheads associated with local as well as global scheduling approaches. For example, one possibility maybe to have tasks statically bound to processors while others are allowed to migrate. Some of the hybrid approaches include Semi-Partitioned and Clustering. Fig. 2.7 shows an example of the hybrid scheduling approach.
Possible Execution Windows

\[
\text{Weight}(i) = \text{Execution}(i) / \text{Period}(i) = 2/9
\]

Figure 2.6: P-Fair (pseudo) windows for a task \( T(i) \) having \( E(i)=2 \) and \( P(i)=9 \). The task must execute for exactly 1.0 units of time in each window to comply with the requirements of P-fair based scheduling.

2.5 Uniprocessor Approaches for Shared-Resource Access

2.5.1 Resource Sharing and Synchronization Algorithms

A synchronization mechanism is needed when one or more resources are shared among more than one task. The synchronization mechanism ensures that the tasks can access shared resources without fearing corruption of their states. In case of scenarios, where only one task can access a shared resource, mutual exclusion capabilities are also provided by the resource-sharing algorithms. Moreover, synchronization mechanisms also strive to limit the waiting times for accessing shared resources. Typical situations requiring resource-sharing algorithms include a shared resource accessible in only mutually exclusive manner and reader-writer scenarios.

Most important resource-sharing algorithms used in the context of uniprocessor approaches include basic Priority Inheritance Protocol [29], Priority Ceiling Protocol (PCP) [29] and Stack-based Resource Policy protocol (SRP) [2]. Before describing the specifics of each algorithm, the terminology common to the algorithms is explained below:
CHAPTER 2. BACKGROUND

Figure 2.7: Hybrid Scheduling Approach

Priority Inversion

The phenomenon of priority inversion is observed when a lower priority task is able to block a higher priority task. This situation can occur when shared resources, protected by a semaphore, are accessed in a mutually exclusive manner. If a shared resource is already held by a lower priority task, then the higher priority task would inevitably need to wait until the lower priority task releases the shared resource.

Uncontrolled Priority Inversion

Uncontrolled priority inversion occurs when a lower priority task, blocking a higher priority task, itself gets blocked by one or more intermediate priority tasks. So, in uncontrolled priority inversion, the execution of the higher priority task becomes dependent on non-intervening intermediate priority tasks.

2.5.2 Basic Priority Inheritance Protocol

The basic priority inheritance protocol was proposed by Shat et. al [29]. Its definition is as follows:

A lower priority task, which has locked a shared resource, attains the priority of any higher priority task trying to access the same shared resource. The lower priority task executes at its newly attained higher priority until it releases the lock to the shared resource. At the end of the shared resource access, the lower priority task regains its original priority.
CHAPTER 2. BACKGROUND

The basic priority inheritance protocol ensures that there would be no uncontrolled priority inversion. But it offers no guarantees regarding the liveness of the solution. Thus the tasks, employing only the basic priority inheritance protocol, can very well suffer from deadlocks. Moreover, priority inheritance protocol suffers from chained-blocking. Fig. 2.8 shows an example of a deadlock situation when tasks are scheduled using the basic priority inheritance protocol.

Figure 2.8: Basic Priority Inheritance Protocol: The figure shows the access mechanism scenario with three tasks T1, T2 and T3. Their priorities are p1, p2 and p3 respectively, with p1>p2>p3. Moreover, the release times are 2, 2 and 0 respectively. Tasks T1 and T3 need to gain access to both shared resources (1 and 2) but they instead get caught in a deadlock situation.

2.5.3 Priority Ceiling Protocol

Priority Ceiling Protocol was proposed by Sha et.al. to overcome the problems of deadlocks and chained blockings associated with the basic priority ceiling protocol. Fig. 2.9 shows an example of a task set accessing a shared resource managed through priority ceiling protocol. Following definitions are needed to introduce the concept of priority ceiling:

---

4In chained blocking, a higher priority task would need to wait for an arbitrary long time before its request for a shared resource is served. The explanation of chained blocking comes later in section 2.5.3.
Assigned Priority
The ’assigned priority’ of a task is the priority assigned to it at the beginning of execution.

Current Priority
The ’current priority’ of a task is the priority of the task at the current point in time. This priority can be same as or different to the assigned priority.

Resource Priority Ceiling
The ’priority ceiling’ of a resource is defined as the priority of the highest potential access to it. In other words, if three tasks with priorities A, B and C access a resource ’R’, then priority ceiling of ’R’ is max(A,B,C).

System Priority Ceiling
System Priority Ceiling is defined as the highest priority among all the currently active resources. For example, if resources ’X’, ’Y’ and ’Z’ with priority ceilings ’A’, ’B’ and ’C’ have currently been locked then the system priority ceiling is max(A,B,C).

Scheduling
Priority ceiling protocol assumes static priority, preemptive scheduling. Thus, priority ceiling protocol is best suited for any static priority-based scheduling algorithm like Rate Monotonic Algorithm (RMA) or Deadline Monotonic Algorithm (DMA).

Type of Shared Resource
Priority Ceiling Protocol assumes a mutually exclusive shared resource, which is protected by a binary semaphore.

Allocation
The priority ceiling protocol assigns a shared resource to the requesting task based on the completion of the following conditions:

- The shared resource is free AND
- The current priority of the requesting task is greater than the current system priority ceiling OR
- The requesting task holds the lock to another resource with priority ceiling greater than or equal to the current system priority ceiling.
CHAPTER 2. BACKGROUND

Priority Inheritance Rule

The lower-priority task executes at its inherited priority until it releases every resource whose priority ceiling is equal to or higher than its inherited priority. At that moment, the lower priority task reverts to the priority before acquiring the higher priority.

Transitive Priority Inheritance

Priority Inheritance is transitive in nature. To elaborate, we assume that there are three tasks A, B and C with priorities X, Y, Z, with \(X > Y > Z\). If task A fails on a resource held by task B then task B inherits the priority of task A (i.e. X). Then, if task B also fails on a resource held by task C then task C also inherits the priority of task A (i.e. X).

Forms of Blocking

Push-through Blocking: A task 'Ti' can be blocked by a lower-priority task, which has inherited the priority from a higher priority task, and when the inherited priority is equal to or greater than the priority of task 'Ti'.

Avoidance Blocking: A task 'Ti' can be blocked by a lower-priority task when it tries to access a currently free resource. In this case, the lower-priority task holds another resource whose priority ceiling is greater than or equal to the priority of task 'Ti'.

Chained Blocking/Multiple Priority Inversion: The phenomenon of 'chained blocking' or 'multiple priority inversion' is observed when a higher-priority task needs to serially access more than one shared resource, and when each of the shared resource is already occupied by lower-priority tasks. In this case, the higher-priority task would need to be blocked for each attempt of accessing an already occupied shared resource.

Shortcomings

Some of the shortcomings of 'priority ceiling protocol' include:

- It is not optimized for dynamic priority scheduling algorithms.
- It assumes that only one copy of the shared resource is available and does not allow modeling of reader-writer scenarios.

2.5.4 Stack-based Resource Policy Protocol (SRP)

SRP [2], basically, extends the ideas of Priority Ceiling Protocol (PCP) to enable it to account for:

- Better performance of dynamic priority algorithms like Earlier Deadline First (EDF)
- Multiple resources and reader-writer locks
Figure 2.9: Priority Ceiling Protocol: The figure shows the access mechanism scenario with three tasks T1, T2 and T3. Their priorities are p1, p2 and p3 respectively, with p1 > p2 > p3. Moreover, the release times are 2, 2 and 0 respectively. The tasks T1 and T3 need to access both of the shared resources (R1 and R2) to proceed. The resource access pattern is shown in the figure.
CHAPTER 2. BACKGROUND

• Efficient implementation by sharing run-time stack resources

Preemption Level
To allow simpler implementation and better performance of dynamic priority scheduling algorithms, SRP introduces the concept of preemption levels. A preemption level (pi) is statically assigned to a job. Unlike priorities, preemption levels remain fixed throughout the run time of the system. In SRP, preemption levels are based on the relative deadlines of jobs by default. That is, a closer deadline means a higher preemption level.

Scheduling
The processor is assigned to the jobs in a preemptive and priority-based manner, with jobs of equal priority getting access in First In First Out (FIFO) order. A job can preempt another job only if its preemption level is higher than the preemption level of the other job.

Deadlocks and Multiple Priority Inversions
SRP prevents 'deadlocks' and 'multiple priority inversions' by enforcing the following two conditions through the use of abstract priority ceilings:

• A job can only start if the currently available resources are enough to meet all the resource requirements of the job.

• A job can only start if the resources are also enough to meet the demands of any other job that may preempt it.

Abstract Priority Ceilings
The priority ceiling is denoted by \([R]\), and any definition of ceiling that fulfills the following condition is acceptable:

If a Job (J) is being executed or has the possibility and ability to execute (by preempting currently executing task), and if its request for any of the resources (R) can make it blocked then its preemption level (pi) is less than or equal to the abstract ceiling of R \((R)\).

In SRP, resource ceilings are based on preemption levels unlike in PCP where they are based on task priorities. Therefore, resource ceilings need not be dynamically computed in case of a dynamic scheduling algorithm like EDF.

Definition
Under SRP, a job (J) can only start execution, if it is the oldest job with the maximum priority and has higher preemption level (pi) than the current priority ceiling of the system. Fig. 2.10 shows an example of a task set accessing a shared resource which is managed through stack-based resource protocol.
Figure 2.10: Stack-based Resource Policy protocol: The figure shows the access mechanism scenario with three tasks T1, T2 and T3. Their priorities are p1, p2 and p3 respectively, with p1>p2>p3. Moreover, the release times are 2, 2 and 0 respectively. The tasks T1 and T3 need to access both of the shared resources (R1 and R2) to proceed. The resource access pattern is shown in the figure.
Modeling of Shared Resources

Binary semaphores are modeled by having the number of resources \( (Nr) = 1 \). Reader-Writer locks can be modeled by making \( Nr \) greater than or equal to the total number of tasks requiring the lock. Before ‘Writing’, a task has to obtain all the ‘Nr’ resources. For Reading one ‘Nr’ is needed, but other writers must be blocked while reading.

2.5.5 Pros and Cons

Early Blocking

In SRP, ‘early blocking’ is introduced by ensuring that a job is not allowed to execute unless all its future resource accesses are ensured to be satisfied. While in PCP, a job becomes a candidate to be blocked when it makes its first resource request.

Context Switches and Extra Priority Inversions

SRP makes less context switches than PCP, and in some conditions this maybe beneficial (if the preempting job requires a non-preemptive resource). SRP may suffer from ‘priority inversion’ (as compared to PCP: because of ‘early blocking’) when an intermediate job does not use any non-preemptive resource.

Nested Resource Accesses

Both PCP and SRP do not allow nested resource accesses.

2.6 Multiprocessor-based Approaches for Shared Resource Access

In multiprocessors, additional complexity is introduced for synchronization mechanisms. The reason is that in multiprocessor scenarios, tasks not only need to wait for resources residing on local processors but also for resources residing on remote processors or outside any processing unit. The followings definitions are necessary to be described before explaining the details of each synchronization algorithm:

Local Processors

The host processor or the processor on which a task starts its execution is defined as the local processor.

Local Resource

A resource which is only accessed by the tasks allocated to the processor on which it resides is known as the ‘local resource’.
Global (or Remote) Resource
A resource which is accessed by tasks having different local processors is known as the 'global (or remote) resource'.

Synchronization Processors
A processor on which a 'global resource' resides is known as the 'global processor'.

Main approaches used when the workload is in the form of sporadic set of tasks are Partitioning, P-fair Global Scheduling and Non-Pfair Global Scheduling \(^4\). Earliest work on semaphore support for multi-processors included that done by Rajkumar on variations of Priority Ceiling Protocol (PCP) \(^25\). Lopez and Gai et el presented implementations for Stack-based Resource Policy protocol (SRP) for Partitioned EDF \(^10\).

2.6.1 Multi-processor Priority Ceiling Protocol (MPCP)
MPCP \(^{25}\) extends the ordinary version of PCP for the case of multiprocessors. Before jumping to the actual definition of MPCP, let’s first look at some of the underlying assumptions of MPCP:

Partitioned Scheduling
MPCP assumes 'partitioned scheduling' on each local processor. As defined earlier, partitioned scheduling means that each processor has its own local scheduler, tasks are bound to a given processor, and tasks do not migrate from one processor to another.

Type of Scheduler
A 'fixed-priority' algorithm is said to run on each local processor. Each scheduler is also aware of the priorities and the resource requirements of tasks accessing the global resources scheduled using it.

Resource Attributes
- MPCP assumes that only one copy of a resource is available at a time, and the resources are protected by binary semaphores.
- Resources are always located on processors, and are distributed into classes of 'local' and 'remote' resources.
- Global resources are executed on the synchronizing processors.

\(^4\)These scheduling approaches are explained earlier in section 2.4
CHAPTER 2. BACKGROUND

Definition

MPCP basically follows the same definition as its uniprocessor version (PCP). The only difference occurs regarding the execution of tasks on remote processors. In MPCP, the tasks accessing global shared resources are executed with a priority higher than the priority of any local tasks of synchronization processors. One way to ensure this condition is to execute the tasks at priorities given by the formula: $p_i(i) - p_i(lowest)$

where $p_i(i)$ is the current priority of the task, while $p_i(lowest)$ is the lowest priority of the task in whole priority space.

Shortcomings

One major shortcoming of the protocol is that it is suited only for the case of 'partitioned scheduling'. Moreover, the protocol discourages nested resource accesses, when the accessed resources reside on different processors: as deadlocks are no longer prevented and the blocking times become unbounded.

2.6.2 Multiprocessor Stack Resource Policy Protocol (MSRP)

Definition

The algorithm acts the same as SRP for 'local' resources. The 'global' resources are accessed in a non-preemptive manner. The task executes the 'critical section' of the global resource in a non-preemptive manner if it is available. If the critical section is not available, the task 'busy-waits' or 'spins' on the global resource as part of a 'queue'. After releasing a 'global' resource, the task again becomes preemptive. Any other tasks, also waiting on the global resource, are granted access in a first-come-first-serve manner [10].

Shortcomings

Like MPCP, MSRP is only suitable for 'partitioned scheduling' case. Moreover, it also discourages the use of nested critical sections.

2.6.3 A Flexible Real-Time Locking Protocol for Multiprocessors

Flexible Multiprocessor Locking Protocol (FMLP) ([3], [6]) was developed to overcome restrictions of nesting of critical sections and for extending lock-based approaches to Global Partitioning (GP) schemes. It was designed to deal with sporadic set of tasks. It can be used with global as well as partitioned scheduling approaches for multiprocessors. Scheduling algorithms tied with FMLP include G-EDF, PD2 and P-EDF.
Types of Resources
FMLP distinguishes between two types of resources:

- **Short Resources**: The resources which are accessed for short periods of time are categorized as 'short resources'.
- **Long Resources**: The resources which need to be locked for long durations of time are referred as 'long resources'.

Type of Blocking Mechanisms
FMLP allows two types of blocking mechanisms:

- Busy Waiting
- Suspension

In FMLP, accesses to 'short' shared resources are blocked using 'busy-waiting', and tasks accessing 'long' critical sections wait as 'suspended tasks'.

Resource Groups
In order to allow nested resource accesses, the concept of 'resource groups' is introduced. Resources are combined into groups with each group containing only short resources or long resources. If requests for a resource are contained within the requests of another resource, and both resources are categorized as either 'short' or 'long', then both resources must come within the same group. "Group locks" need to be attained to access the resources. For "Short Resources", a "queue-lock" is used, and jobs become non-preemptive before trying to acquire the lock. For long resources, a "semaphore" lock is used instead. A job, having the current lock for long resources, also inherits the highest priority from all the tasks waiting on the lock. (if this priority is higher than its own priority).

Scheduling Algorithms
**Global EDF (GEF)**: In G-EDF, unlike the original version, jobs may become non-preemptive for short periods of time. Moreover, a differentiation is made between the 'linking' and 'scheduling' of jobs. At all times, any given job is either 'linked' or 'unlinked' to a processor. For 'm' processors, the 'm' jobs with the highest priorities are 'linked' to a processor. If a job is 'linked' to a processor but not 'scheduled' then this means that a non-preemptive, low-priority job is currently executing on the processor.

**Request Rules**: Before accessing a shared resource, a task must first acquire its 'group lock'. Following access rules are used by FMLP when accessing 'short' and 'long' resources:

- **Short Resources**: Before requesting a group lock protecting a 'short resource', a job becomes non-preemptive. It remains non-preemptive until it releases its 'group lock'.
• Long Resources: A task preemptively accesses a 'long resource'. When a task wins a grant to a 'group lock', it also inherits the priority of the highest-priority task waiting for the 'long resource'. The task reverts to its original priority after completion of its shared resource access.

Partitioned EDF (P-EDF): A modified version of P-EDF is used with FMLP. In this modified version, resources are distinguished between 'local' and 'global' resources. Only the 'global' resources need to be governed with FMLP. Moreover, all resource holding jobs are scheduled non-preemptively. Furthermore, priority inheritance is allowed for local blocking tasks.

Static Priority Scheduling: FMLP can also be used in conjunction with a static priority scheduling algorithm. In this approach, all resource-holding jobs are scheduled non-preemptively. Moreover, a resource-holding job is scheduled as the highest priority task to restrict any undue blocking by non-resource holding jobs.

2.7 Alternative and Contemporary Approaches of Shared Resource Access Modeling

Over time, a number of approaches have been proposed to model the impact of shared resource accesses schemes and scheduling policies. These schemes can basically be divided into the broad categories of analytical methods, simulation-based methods and hybrid approaches. Some of these approaches are summarized below:

2.7.1 An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems

An alternative approach (End-to-End Scheduling) to solve the problem of shared resource access in multiprocessor scenarios is proposed in [30]. In End-to-End Scheduling approach, tasks are divided into smaller subtasks based on the shared resource accesses made in them. Later each of the subtasks is assigned a priority and a relative deadline. If the sum of the worst-case response times of each of the subtasks is within the overall the deadline of the task, then the task is schedulable. Similarly, if all the tasks are schedulable then the overall system is schedulable.

In the mapping step, each part of the task is divided into a smaller subtask if accesses are made to remotely available resources, or to local resources or no resources. In case of nested accesses to shared resources, each outermost critical section is regarded as a subtask. An effort is also made to map consecutive subtasks onto different processors. Later a variety of methods (like Rate Monotonic, Global Deadline Monotonic or Relative Deadline Monotonic) can be used to assign deadlines to the subtasks. In the third step, Worst-case
Execution Time (WCET) is calculated for each of the subtasks of the system. Finally, WCET is calculated for each task in the system based on the sum of the WCET’s of all the subtasks. If the WCET’s of all the tasks in the system are within the deadlines, the system is schedulable under this partitioning of subtasks.

Some of the limitations of the approach include the assumption that resources accessed within a nested critical section are available on the same processor. Moreover, to satisfy the precedence constraints between different subtasks, the concept of "phase" is used. But such a concept requires the need of tight clock synchronization between different processors.

### 2.7.2 Shared Resources High-Level Modeling in Embedded Systems Using Virtual Nodes

A rudimentary method of modeling and analyzing the effects of shared resources is presented through the concept of "virtual nodes" [17]. A virtual node controls the access to a shared resource. Each virtual node also contains a queue that stores the pending requests to a shared resource. Virtual nodes support basic scheduling policies like 'priority-based' access or 'round-robin'. They also support the scenario when a heterogeneous group of sources try to access a shared resource. Such heterogeneous accesses are allowed through hierarchical stacking of virtual nodes. In hierarchical stacking, a separate virtual node controls the access from each group of similar competitors. While in the second level, another virtual node arbitrates between the winners from different groups of contenders.

Shortcomings of 'virtual node methodology' include that elaborate synchronization policies (like Priority Ceiling) (that disallow dead-locks and limit blocking times due to priority inversion) do not come in the scope of the Virtual Node scheme.

### 2.7.3 Shared resource access attributes for high-level contention models

A simulation-based method to capture the effect of shared resources on the overall performance of the system is presented in [4]. The resource access attributes are obtained from the cycle-accurate simulations of the system. These resource access attributes are used to train a statistical model, and later this statistical model is merged with the high-level abstract simulations to find out the impact of shared resource contentsions. Based on numerous simulations and various trade-off considerations, (like goodness of fit, ease of training, amount of computation and calculation, and sampling length) three attributes are chosen. *Average access utilization*, *access balance* and *thread concurrency* are used as shared resource access attributes.

*Average access utilization* is the measure of busyness or the demand of the shared resource. While *access balance* captures how much the requested utilization of a particular thread varies with respect to the overall average utilization of
all the threads. On the other hand, thread concurrency counts the total number of threads concurrently accessing a shared resource.

The limitations of this method include that the approach is constrained in its scope. High-level simulation model can only be augmented with resource-specific information relatively late in the design cycle (only after cycle-accurate simulations are available). This approach may not be very useful because at this late stage, changes in the system design will be costly. Moreover, the approach is limited in the scope because the only type of contention that is shown to be modeled at higher abstraction levels is ‘shared memory access’.

2.7.4 MESH based Hybrid Simulation/Analytical Approach for Modeling Shared Resource Contention

This approach takes the middle path, between pure simulation and analytical approaches, to model shared resource access contention\cite{5}. Shared resources are represented in Modeling Environment for Software and Hardware (MESH) simulation framework with attached analytical model. Shared resource access behavior is modeled by adding penalties to the execution time of a task accessing a shared resource. These penalties are in turn calculated by the analytical model attached with the shared resource.

To model shared resources, three layers are present in MESH framework. These layers are named Software, Scheduler and Hardware Resource. In the software layer, tasks are represented as threads which compete for processing units located at the hardware resource layer. These tasks are arbitrated by the ‘task scheduler’ located at the scheduler layer. ‘Resource schedulers’ map requests from tasks to shared resources located at the hardware resource layer. They also apply execution penalties to the tasks after their accesses are finished. The amount of applied penalty is decided by the shared resource access model attached with each resource.
Chapter 3

Functional Requirements of the Framework

This chapter presents the functional requirements of a generic and flexible analysis framework for shared resources in multicore environments. The functional requirements are discussed in the order of major blocking behaviors, resource access mechanisms and related scheduling algorithms.

This following sections present the functional requirements of the analysis framework in terms of major blocking methods, resource access mechanisms and associated scheduling algorithms. These functional requirements stem from the background information explained in chapter 2. The functional requirements are specified in terms of the required prerequisite information, subsequent required actions and additional degrees of freedom to implement the functional requirements. The degrees of freedom document extra design space available to the framework designer as well as the system architect using the framework. Framework designer can use one of the possible methods to achieve the same functional requirement, while asserting possibly different simulation performance. While some of these degrees of freedom differ from the default behavior, but provide additional orthogonal concerns to the system architect, thus allowing greater flexibility in designing the system. These two different types of degrees of freedom are also marked by * and † respectively, where ** represents the design space available to the framework designer while † represents the additional orthogonal concerns available to the system architect.

3.1 Blocking Behaviors

This section discusses the design goals of the blocking behaviors that are considered in the proposed framework.²

²An introduction to blocking methods appears previously in section 2.2
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

3.1.1 Busy Waiting

Information
Knowledge of availability of resource

Actions
- Termination of further execution of task
- Continuous consumption of processor cycles

Degrees of Freedom
Following are the possible degrees of freedom:

Local Approach ★ In a loop, the task looks for the availability of a shared resource continuously or after algorithmically-determined time. It consumes processor cycles by performing some dummy operations.

Scheduler-based Approaches ★ The scheduler simulates the busy-wait effect on the processor. An attribute in the task or a system call indicates to the scheduler that the task is busy-waiting.

Direct Approach In direct approach, the scheduler consumes dummy cycles in the context of the requesting task and resumes the normal operation of the task once the requested resource becomes available.

Indirect Approach The indicating task is suspended and instead a dummy task with the same attributes is scheduled. The dummy task executes (consuming processor cycles) as long as the required resource is not available.

3.1.2 Suspension-based Approach

Information
Knowledge of availability of resource

Actions
Change of the state of the task to sleep (or suspension) mode

Degrees of Freedom
Following are the possible degrees of freedom:

Direct Approach ★ The task suspends itself, waiting for the occurrence of an event associated with the grant of shared resource.
Indirect Approach ★ The task indicates to the scheduler that it intends to do suspension-based waiting for a shared resource. It is the responsibility of the scheduler to wake up the task after the resource gets available.

Choice of a Blocking Approach ★ Additional degrees of freedom can even be provided within the choice of the blocking approach. An arbitrary combination of busy-waiting and suspension-based approach can be used, with a given approach being determined by a custom algorithm or the scheduler.

3.2 Resource Access Mechanisms

This section discusses the functional requirements of the various resource access mechanisms considered in the proposed framework.

3.2.1 Non-preemptive Critical Section

Information
Grant of a shared resource

Action
Ability to make the task non-preemptive during the access to a shared resource.

Degrees of Freedom
Following are the possible degrees of freedom:

Task-based approach ★ A task raises its priority to the highest possible priority and subsequently lowers it, corresponding to the grant of a shared resource.

Scheduler-based approach ★ The scheduler makes the task non-preemptive corresponding to the grant of a non-preemptive shared resource, and also restores the priority afterwards.

3.2.2 Priority Inheritance Approach

Information
- List of previous set of priorities of tasks
- Confirmations of completions of requests
- Knowledge of failed attempt’s priority

---

An introduction to major resource access mechanisms for uniprocessor and multiprocessor scenarios previously appears in sections 2.5 and 2.6 respectively.
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

- Knowledge of currently held resources

Action
- Ability to update the current priority of the task for a failed attempt, and revert to the original priority after an unlock request.

Degrees of Freedom
Following are the possible degrees of freedom:

Task-based approach In task-based approach, a task manages its previous and current priorities itself.

Independent approach The scheduler or an independent entity may do the priority management. The entity may do management for a single task or all tasks using a common scheduler.

3.2.3 Multiprocessor Priority Ceiling Protocol (MPCP)

Resource-wise Priority Ceiling

Information Maximum priority access to a resource

Degrees of Freedom Following are the possible degrees of freedom:

- Lifetime
  - Static: The resource ceilings remain static throughout the lifetime.
  - Dynamic: The resource ceilings may change depending upon the dynamic-priority local scheduling algorithm (e.g. Earliest Deadline First algorithm) or the local algorithm being used by the resource (e.g. as a function of time or some property of the system).

- Updation Whenever the priority of a task gets changed
  - All resources that are ever used by the task are informed.
  - Only potential future resources are informed.
  - All actual future accesses are known and are thus informed.
  - Updates of resource ceilings are done only at certain synchronization points on the initiations of tasks or resources

\[^3\text{when priority inheritance is used as part of Multiprocessor Priority Ceiling Protocol (MPCP)}\]
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

Lowest System Priority

Degrees of Freedom Following are the possible degrees of freedom:

- Propagation
  - Priority information is fed into the schedulers at the beginning of the simulation.
  - Tasks report their priorities to remote schedulers when the priority gets changed or as required by a custom algorithm (e.g. at a synchronization point).

- Scope
  - Global: All schedulers share the same knowledge about the lowest priority task.
  - Local: Different schedulers may assume a different lowest priority task.

- Lifetime
  - Static: The designated lowest priority task always remains the same.
  - Dynamic: The lowest priority can change depending upon the dynamic priority scheduling algorithm or a locally employed algorithm by the scheduler.

- Alternatives
  - Lowest priority information is not needed; instead each task executes at a certain algorithmically predictable priority at the remote processor. e.g. remote priority = - (original priority)
  - Lowest priority information is not needed; instead each task executes at its current or assigned priority even at remote processors.

System-Wide Priority Ceiling

Information Highest priority active resource

Degrees of Freedom Following are the possible degrees of freedom:

- Visibility
  - Global: There is only a single value of system-wide priority ceiling.
  - Local: Different system-wide priority ceilings may exist for different components.

- Locality
– Central: A central 'resource manager' manages the overall system-wide priority ceiling. The 'start' and 'end' of each resource access is registered with the central resource manager.

– Distributed: System-wide priority ceiling maybe managed by the resources (Any given resource knows the state of all other resources as well. Each access eventually gets propagated to each resource) or schedulers (Each scheduler maintains its own view of priority ceiling and the access request is propagated to the resource only if the system-wide priority requirement is met).

– Hybrid: Groups of resources are managed by arbitrary combinations of central and distributed approaches with the ability of a resource to dynamically change from one approach to another.

3.2.4 Multiprocessor Stack-based Resource Policy Protocol (MSRP)

Preemption Level

**Degrees of Freedom** Following are the possible degrees of freedom:

- **Visibility**
  - Global: Preemption level can be part of the global preemption-level space with each resource having the same view.
  - Local: Resources may have different views of preemption levels.

- **Uniqueness**
  - Unique: Each preemption level is unique in whole preemption-level space.
  - Same: Two tasks may have the same preemption level.

Abstract Ceilings

**Information**

- No. of available units.
- Resource requirements for all tasks for that resource

**Degrees of Freedom** Following are the possible degrees of freedom:

- Static Calculation: Abstract Ceilings can be calculated statically and stored as part of resource-managing entity.
- Dynamic Calculation: Abstract Ceilings can even be calculated dynamically by the resource-managing entity.
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

3.2.5 Flexible Multiprocessor Locking Protocol (FMLP)

Resource Groups

Degrees of Freedom  Following are the possible degrees of freedom:

- Life-time
  - Static: Resources are grouped into short and long before elaboration phase and they remain the same.
  - Dynamic: Resources may change their default behavior from short to long on the initiations of the resource itself or managing entity.

- Locality
  - Central: A central resource manager manages the group-locks (as well as needed interactions with tasks and schedulers) for all resources.
  - Distributed: A separate resource manager exists for each group of resources.
  - Scheduler-based Approaches: Local schedulers also do the job of a resource manager. This approach is best-suited if all the resources reside on the same processor.

3.3 Scheduling Algorithms

This section discusses the functional requirements of scheduling algorithms tied in with Flexible Multiprocessor Locking Protocol (FMLP).

3.3.1 Partitioned-Earliest Deadline First (P-EDF)

Priority Boosting

Information  Identification of ‘outermost requests’ by schedulers.

Degrees of Freedom  Following are the possible degrees of freedom:

- Scope
  - Global: All the schedulers may choose to employ priority boosting.
  - Local: Each scheduler has an option to employ priority boosting or not.

- Value

---

4 For dynamic definition of resource types, dynamic definitions of resource groups is also necessary.
5 A brief introduction to these and other scheduling algorithms appears in sections 2.4.1 and 2.4.2.
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

– Global: Every scheduler must boost the priority by the same value.
– Local: Every scheduler has the option to algorithmically choose the best possible value for priority boosting.

### 3.3.2 Global Earliest Deadline First (G-EDF)

**Information**
- Priority Information of all tasks.
- State information of all tasks.
- Information about which task is scheduled on which processor.
- Information about which tasks is linked on which processor.

**Actions**
- Differentiation between different states of the task (like Released/Resumed, Preemptive/Non-preemptive, Suspended/Complete).
- Invocation of scheduler based upon the state of the task.

### 3.3.3 PD^2

**Information**
Knowledge of weight [(execution time)/(period)] of each task.

**Actions**
- Synchronization between different processors based upon 'quantum sizes'.
- Adjustment of execution cost and period to be integer multiple of quantum size.
- Invocation of scheduler at quantum boundaries.

**Degrees of Freedom**

**Quantum Size**
- Lifetime
  - Static: Same quantum size is used for the lifetime of the system.
  - Dynamic: Quantum size may change over time, based upon some algorithm.
CHAPTER 3. FUNCTIONAL REQUIREMENTS OF THE FRAMEWORK

Tie-breaking Rules

• Scope
  
  – Global: All schedulers must use the same tie-breaking rules.
  – Local: All schedulers may use different tie-breaking rules.
Chapter 4

Design of the Framework

This chapter introduces the object-oriented design of the proposed shared resource analysis framework. The framework is conceived in the form of Task, Scheduler, Internal Resource access Abstraction, External Resource access Abstraction, Resource Management Abstraction, Resource, Request, Response and Waiting Data Structure as major components. The chapter begins by first presenting the overall view of the system. Later each of the major components is explained in subsequent sections.

To meet the widely varying design demands, object-oriented approach is used in designing, and the proposed framework is designed in the form of interacting entities or objects. Various entities and their interactions are visualized in the system diagram 4.1. The task communicates through the data structures of request and response to the resource, which is located as part of a resource access abstraction module. Internal Resource access Abstraction is mainly responsible for managing all the internal interactions, within the processing unit. External Resource access Abstraction is located at the boundary of a processing unit, and manages all the outward direction communication. The detailed object-oriented design of each of the entities appears below:

4.1 Task

Task class hierarchy is illustrated in figure 4.2. Task forms the root of class hierarchy. It is further divided in terms of static and dynamic priority tasks, which are in turn derived from a priority-based task. Resource access mechanism information is kept as part of a shared resource access task. Shared resource access task, itself, is part of priority based task. There exists a one-one correspondence between task and Internal Resource access Abstraction modules.

Each task must contain information about its id and state. A priority based task must contain the information about its local priority\(^1\) and whether it is

\(^1\)the priority information can also be managed by the scheduler.
preemptive or not. Rate monotonic or deadline monotonic tasks must contain the information about the rate or the deadline, which inform the scheduler about the urgency with which they must be scheduled. A degree of freedom in dynamic priority tasks is to contain the information about the schedulers which would be updated with the new priority of the task. A task scheduled by (dynamic priority) Global Earliest Deadline First algorithm must contain the id’s of the scheduled as well as linked processors. A p-fair task must contain information about the execution time, weight and period.

A task may (optionally) also be a shared resource access task. It must contain the list of shared resources and announce the updated (dynamic) priority to these shared resources. This needs to be done to update the priority ceilings of the shared resources, if the task has dynamic priority. Preemption level information is contained in the tasks using Multiprocessor Stack Resource Policy protocol (MSRP). Moreover, a priority-inheritance based task must contain information about its previous priorities as well as current resource priority ceiling. The latter information is required if the priority inheritance protocol is to be used in conjunction with a priority ceiling protocol.

4.2 Scheduler

The scheduler class hierarchy is illustrated in figure 4.3. Schedulers are divided into priority based and non-priority based schedulers. Priority based schedulers are further divided into Pfair based and non-Pfair based portions. Resource access mechanism specific information is kept as part of a shared resource access scheduler.

A P-fair based scheduler must contain information about the quantum size. Similarly, a Partitioned Earliest Deadline First (P-EDF) scheduler may contain a flag telling whether priority boosting is to be used or not. Instead of a task, a Global Earliest Deadline First (G-EDF) scheduler may contain information about which task is scheduled and linked to which processor. If a shared resource access is also to be used with a given scheduler, then instead of a task itself, a
scheduler may contain the list of previous priorities of the task when priority inheritance is used. Similarly, the information about the lowest priority task must be stored if the scheduler is to be used with Multiprocessor Priority Ceiling Protocol (MPCP).

4.3 Internal Resource Access Abstraction

Internal resource access abstraction module is designed to mainly manage all the internal communication within the processing unit, on the behalf of the task. It communicates with the scheduler, and also implements the required blocking methods. The class diagram of internal resource access abstraction module appears in figure 4.4 and its further details are described below.

Each Internal Resource access Abstraction (IRA) module interacts with only one scheduler. It must identify which tasks are busy-waiting or suspended on which resources. abstract_request functions are meant to provide an additional degree of freedom, as the IRA module must algorithmically decide the best
CHAPTER 4. DESIGN OF THE FRAMEWORK

4.4 External Resource Access Abstraction

The role of external resource access abstraction unit is to manage all the communication going outward from the processing unit. It needs to translate the incoming communication into a suitable form for the outward interconnection network. Similarly, it also translates and processes the response. A class diagram depicting external resource access abstraction unit appears in figure 4.5.

4.5 Resource Management Abstraction

Resource management abstraction unit is responsible for accounting all the access requests for a particular resource. It also needs to implement resource access
mechanism functions needed at the resource side. Figure 4.6 illustrates the class diagram structure for a resource management abstraction unit. The required information for major resource access mechanisms is captured in terms of being priority-inheritance based or not.

Resource Management Abstraction (RMA) unit must manage the resource access mechanism on the resource side. It must process the request depending upon the request's nature and the availability of the shared resource. It must also later send back a response. It must also inform any inherited priority in case of priority inheritance protocol. System ceiling information must be maintained and calculated when Multiprocessor Priority Ceiling Protocol (MPCP) or Multiprocessor Stack Resource Policy protocol (MSRP) is used. In case of Flexible Multiprocessor Locking Protocol (FMLP), the information about which resource belongs to which group must be maintained.

4.6 Resource

Resource is the entity managed by a resource management abstraction unit. It contains information as the number of its available instances, successful task(s) and any waiting task(s). The class hierarchy diagram is shown in the figure 4.7.
Each resource contains an associated waiting data structure to contain information about the waiting tasks. It also contains information about the number of its available instances to handle semaphore based shared resources. It must also contain information about the priority inherited by each task, if priority inheritance is supported. In case of Multiprocessor Stack Resource Policy protocol (MSRP), information about abstract ceiling table must be maintained. Resource priority ceiling information must be maintained for Multiprocessor Priority Ceiling Protocol (MPCP). Type of resource and its group information must also be maintained for Flexible Multiprocessor Locking Protocol (FMLP).

### 4.7 Request and Response

Request and response structures are used to convey information from a task to a resource management abstraction unit and vice versa. Different resource access mechanisms require different information to be kept as part of request and response data structures. Their structures are shown in figure 4.8.

Task priority, priority ceiling among currently held resources and preemption level information is needed in case of priority inheritance protocol, Multiprocessor Priority Ceiling Protocol (MPCP) and Multiprocessor Stack Resource Policy protocol (MSRP) respectively. Similarly, response must contain any inherited priority as well as id of inheriting task for priority inheritance protocol, and resource ceiling information for Multiprocessor Priority Ceiling Protocol (MPCP) and Multiprocessor Stack Resource Policy protocol (MSRP) respectively.
CHAPTER 4. DESIGN OF THE FRAMEWORK

Figure 4.7: Resource Class Diagram

Figure 4.8: Request and Response Class Diagram
Chapter 5

The Shared Resource Analysis Framework

This chapter explains the realization of the analysis framework using SystemC based abstract task modeling and mapping concepts. It begins by explaining the foundations of the framework in terms of an abstract task model and processing elements in the form of virtual processing units. Later, it explains each of the components of the framework, and elaborates how these components work together to form the resource analysis framework. Furthermore, various design decisions taken during the implementation are also discussed.

5.1 Abstract Task Modeling and Mapping in SystemC

The work in [18] forms the underlying basis of the proposed shared resource analysis framework. In [18], a SystemC [16] based simulation framework was put forward that allowed the evaluation of an arbitrary mapping of an application (modeled in the form of a set of timed communicating tasks) on a given (processing and communication ) architecture. The following subsections summarize the important and relevant concepts presented in that research or are an extension of that.

5.1.1 Task

A task is specified through its C/C++ based functionality as well as the time required to execute that functionality. Each task is a timed Communicating Finite State Machine (tCFSM) and it is modeled as a SystemC based thread, which is in-turn part of an sc_module. It can use ports and channels to communicate with the external world.

1Each module in SystemC must be derived from sc_module class.
The states of the task are defined as CREATED, READY, RUNNING, WAITING, SUSPENDED and DESTROYED.

The task enters CREATED state after its creation. It needs to be started to get into READY state. It can only be activated and allocated processor time if it is in READY state. Once a task is allocated processor time, it gets into RUNNING state. A task waiting for an event or passage of time goes into WAITING state. Moreover, a task can suspend/destroy itself or get suspended/destroyed by other tasks to go into SUSPENDED/DESTROYED state. A task into SUSPENDED state can be resumed by any other task to go into READY state. Once a task goes into DESTROYED state, it cannot be resumed. The state transitions of the task are graphically shown in the figure 5.1.

![State diagram of task states](image)

Figure 5.1: State diagram illustrating the states of the task in abstract task modeling. Wait(0) means that the control is voluntarily passed back to the scheduler.

### 5.1.2 Virtual Processing Unit (VPU)

VPU represents an abstract processing resource for the tasks mapped on it. It acts as the middle entity between the event driven simulation kernel and time tasks mapped onto it. A VPU can have one or more memory ports, one or more interrupt ports and a single clock port. The default settings of VPU recommend all outward communication to be done through the memory port(s). The number of interrupt and memory ports is also configurable; while memory ports are TLM2 [23] based ports. It is also possible to add a custom port to the VPU. Only outward communication can be done through the custom ports.

A VPU also contains a number of XML file based configurable properties. `scheduler_type` parameter must contain the name of the currently selected scheduler. `pre-emption` parameter specifies if the currently executing task must be suspended and the scheduler be called, in case a new task becomes ready state. `time-slicing` specifies if the scheduler must be called after a fixed interval of time. Moreover, it is also possible to specify the duration of a `time-slice`. Furthermore, it is also possible to specify if the currently executing
task must be preempted in case of an interrupt or when there is a change in the ready tasks.²

5.1.3 Task Manager

A task manager is the unit that takes care of the state of the tasks. Each VPU has a task manager associated with it. There is also a default task manager in case no VPU is used in the design. In the latter case, all the tasks are managed by the default task manager. It is the responsibility of the task manager to activate the tasks and call the associated scheduler to select the next executing task.

5.2 Design and Implementation of the Overall Framework

The design of the overall framework is illustrated in figures 5.2 and 5.3, while the functionality of each component is shown in figures 5.4 and 5.5. The task makes the request for a shared resource. Internal Resource access Abstraction (IRA) manages all the interaction that is to be done inside a processing unit, in response to a request. External Resource access Abstraction (ERA) manages the outward direction communication, while Resource Management Abstraction (RMA) manages concurrent accesses to a shared resource. The functionality of each component of the framework is elaborated in the following sections.

5.2.1 Task

A task follows the same structure as defined earlier in the object-oriented design. It uses its resource port to request a shared resource. Each task is internally assigned a unique id by the framework. Moreover, its task pointer is also internally stored as one of its parameters by the framework. For a request, a task needs to set its task id, task pointer,³ requested resource’s id, and flags telling whether the task wants non-preemptiveness and/or priority inheritance. The task can use one of the different types of access functions of its port, depending upon the blocking behavior it wants.

5.2.2 Priority Hierarchy

The framework introduces the priority hierarchy in which the framework level tasks are given the highest priority. After the framework-level tasks, user-defined non-preemptive tasks have the highest priority. Then, a special priority class (used only for synchronization purposes) is defined. At the end of priority

²It is important to note that such a preemption can only happen if the task is in the process of consuming processor cycles.
³That can be accessed through one of the framework’s convenience functions.
5.2.3 Request

A request is a data structure that is used to carry information starting from a task (and eventually) to a Resource Management Abstraction (RMA). Various components add, modify and act on the information, while it on the way to destination (and back). A request contains the fields as the \textit{task id}, \textit{resource id}, \textit{task pointer}, \textit{priority}, \textit{type of blocking}, \textit{type of request}, and \textit{flags} indicating whether priority-inheritance or preemptiveness is enabled. The request structure is also shown in the figure \ref{fig:5.7}.

5.2.4 Shared Resource Access

Shared Resource Access is a simple module whose role is to implement the convenience interface provided to the task. The task can use one of the functions of the interface depending upon the type of blocking method it needs and whether preemptiveness is requested. The request is passed on to Internal Resource Access Abstraction (IRA) module, after filling in the required details.

Figure 5.2: Overall Frame Design: Inside a Virtual Processing Unit (VPU). 
\textit{clk} represents the clock port, while \textit{p\_interrupt[0]} represents the interrupt port. \textit{p\_mem[0]} is the TLM2 based memory port for outward communication.
5.2.5 Internal Resource Access Abstraction

Internal Resource Access Abstraction (IRA) is one of the building blocks of the framework. Its main functionality is to interact with all the components located internally in a VPU to ensure correct behavior of a resource access. In particular, the roles of an IRA module include management of resource accesses of a task, blocking behavior and resource-access-mechanism related management, communication with the scheduler, and ensuring correct synchronization behavior of the overall framework. Each IRA unit has an owner task, which uses the services of a particular IRA module.

Each IRA unit is inherently a driver module. All the external communication must pass through TLM2 based memory ports of the VPU. The responsibility of IRA module is also to implement the communication interface for the task. It later translates this communication into TLM2 styled transactions, for External Resource Access Abstraction (ERA) module. The code of IRA module is executed in terms of the owner task, since it implements the communication interface for the task. The role of IRA module is also to do synchronization between the timed task network, mapped on the VPU, and the event based SystemC scheduler. It also needs to ensure that the currently executing task is not interrupted while communication with the outside world is being done.

The functionality and other design issues of an IRA module are discussed in more detail below:
Figure 5.4: Functionality of framework components inside a Virtual Processing Unit (VPU). \( \text{clk} \) represents the clock port, while \( p\text{-interrupt}[0] \) represents the interrupt port. \( p\text{-mem}[0] \) is the TLM2 based memory port for outward communication. The request gets modified at several points, while it travels from the task to the memory port (\( p\text{-mem}[0] \)). Interrupt listener task in Internal Resource access Abstraction (IRA) is responsible for waking up any suspended task and communicating with the scheduler. It also sends a special clarification request after receiving an interrupt. A new request must be sent periodically from IRA for a busy-waiting task. External Resource access Abstraction (ERA) generates actual TLM2 transactions for the outside world.
Figure 5.5: Functionality of Resource Management Abstraction (RMA). RMA module generates the response as well as the interrupt (in case of a failed request or a grant to a suspended task). It also contains the resource module.

Management of Resource Accesses of a Task

IRA manages several aspects of a resource access of a task. It translates the lock/unlock requests received from the previous module into TLM2 styled transactions for the next component of the framework. Moreover, it stores the identifying information of its owner task like its priority and its task pointer. It also fills up the request structure with the current priority of the requesting task. The request structure is always passed as an extension of the TLM2 styled transaction. IRA also internally stores the information like the requested resource’s id, type of blocking requested, whether priority inheritance and non-preemptiveness is enabled, and if the request has been successful or not in a separate data structure.

---

4 The next component of the framework is External Resource access Abstraction (ERA) module.

5 This priority information is used in latter parts of the framework like Resource Management Abstraction (RMA) module.
Nested Resource Accesses

To enable nested resource accesses, the resource access information (as mentioned above) is stored in the form of a Last-In-First-Out (LIFO) stack\(^6\). A new lock request causes new information to be placed on this stack, and this information is removed as soon as a request is finished. It is assumed that the outer requests have higher precedence than inner requests in terms of parameters as priority-inheritance and non-preemptiveness\(^7\).

Management of Blocking Behavior and Resource Access Mechanism Information

An IRA module also manages the requested blocking behavior and the resource-access-mechanism information.

---

\(^6\)It is assumed that the resource accesses are always properly nested. This means that the requests occurring later also always finish earlier.

\(^7\)This means that if priority inheritance/non-preemptiveness is enabled in one of the outer requests, they get enabled by default in all subsequent inner requests.
Suspension-based Blocking IRA uses the owner task’s pointer information to suspend the unsuccessful task. It is also the responsibility of the IRA module to wakeup the suspended task, once its request gets granted. Resource Management Abstraction (RMA) generates an interrupt signal in case a suspended task gets access to the intended shared resource. Each IRA module has a framework-level interrupt listening task that continuously listens for an event on its interrupt port. As soon as an interrupt is received, a clarification request is sent, if IRA is expecting to wakeup a suspended task or is expecting a priority inheritance boost. If the response of the clarification is successful and is intended for the owner task of the particular IRA, the task is woken up.

Busy-Waiting IRA does busy-waiting by continuously consuming processor time (in the context of the unsuccessful task) and sending the lock requests. In the case of busy-waiting, Resource Management Abstraction (RMA) does not generate an interrupt signal. IRA consumes processor time in chunks of two time units and also periodically passes the control back to the scheduler. This consumption is done in chunks of two time units to ensure correct synchronization of the overall framework in case of a time-slice based synchronization mechanism. The scheduler can only interrupt the currently executing task when it is in the middle of consuming processing time, not in the case of perfect alignment of boundaries of time-slice and processor consumption period. The control is periodically passed back to the scheduler to give execution chance to other framework-level higher priority tasks, and when these framework-level higher priority tasks cause another user-level higher priority task to execute instead.

Non-preemptiveness IRA also manages non-preemptiveness of a task by raising its priority to NON_PREEMPTIVE_PRIORITY. Non-preemptiveness (and also priority-inheritance) is managed with the help of a local priority stack. This priority stack contains the original priority of the task everytime a successful lock request is made and non-preemptiveness/priority-inheritance is enabled. The original priority of the task is restored after a successful unlock request.

Priority Inheritance As mentioned in the above section, priority stack is also used to manage priority inheritance. Priority inheritance information is also conveyed through the interrupt port of IRA. Interrupt listener task also receives an interrupt in case of possible priority inheritance. After the subsequent clarification request, it is decided (depending upon the current priority of the task) whether priority of the task needs to be raised or not.

---

8. This interrupt listening task is assigned SYSTEM_PRIORITY, the highest possible priority in the framework.
9. More details of this mechanism appear in later sections.
10. This is the second highest possible priority in the framework after SYSTEM_PRIORITY.
11. The rest of the details about the management of priority inheritance in other components appear in later sections.
Synchronization with the Scheduler

IRA module passes the control back to the scheduler at the end of every request and also while doing busy-waiting (as explained earlier). This is required for the correct synchronization behavior of the overall framework, and ensures the proper scheduling of higher priority interrupt handling tasks. In the case of a non-priority based algorithm (like round-robin), special care needs to be taken to ensure that the control again returns to IRA before any other user-level task gets scheduled. This is done by assigning a special priority called IRA_PRIORITY to the owner task of IRA, before voluntarily passing the control back to the scheduler.

5.2.6 External Resource Access Abstraction

The role of External Resource access Abstraction (ERA) is to manage the outward direction communication. It is situated between the IRA and the outward direction memory ports of VPU, and is in fact a transactor\(^{12}\). It implements the utility communication interface required by an IRA module, and generates TLM2 based transactions. Moreover, it fills in the address field of the transaction based upon the resource_id of the request\(^{13}\).

It is important to note that all the IRA modules located within a VPU must use the single ERA unit. Mutually exclusive access is not separately required, since the task manager is locked by all IRA modules when they use the services of ERA\(^{14}\).

5.2.7 Resource Management Abstraction

Resource Management Abstraction (RMA) is one of the cornerstones of the framework. Its functionality involves managing concurrent accesses to a shared resource, managing resource access mechanisms and generation of interrupts for priority inheritance or suspension-based accesses. It basically manages the shared resource module contained within it.

RMA must accept TLM2 based transactions coming from multiple VPU’s and thus must process these transactions in a mutually exclusive manner. It achieves mutual exclusion by using a standard SystemC based mutex.

An RMA unit differentiates between the lock, unlock and clarification requests. It uses the interface functions of a shared resource to manage these different requests. In case of a lock request, it checks the available resource access mechanism function. If successful, it grants the request. Otherwise, the

---

\(^{12}\)Transactors are convenience modules that are placed between drivers and TLM2 based ports of the VPU. The role of a transactor is to provide a high-level abstraction interface between the transactions posted by the driver and the low-level details of the sockets to send actual TLM2 transactions.

\(^{13}\)For convenience, it is assumed that the addresses of all the resources correspond to their respective id’s.

\(^{14}\)When the task manager is locked, no other task can be scheduled.
CHAPTER 5. THE SHARED RESOURCE ANALYSIS FRAMEWORK

66

Task information is added in the wait list associated with the shared resource. Similarly, it acts accordingly upon an unlock request. In case of a clarification request, it checks if the requesting task has been granted the resource, and it also fills in the corresponding (if any) priority inheritance value in the response.

A interrupt is generated by RMA in case an unlock request results in the subsequent grant of a waiting suspension-based task. Moreover, an interrupt is also generated every time a lock request fails. The latter is done to inform all tasks interested in priority-inheritance to send in their clarification requests.

5.2.8 Resource

A resource is the entity or module managed by an RMA unit. The resource module in turn manages a granted task list, a waiting task list and a priority-inheritance list. The framework allows for the specification of the resource as either a mutually exclusive resource or a semaphore-based resource (which is shared among a fixed number of tasks).

By default in the framework, the waiting list is kept as a First In First Out (FIFO) list. Moreover, the resource needs to differentiate between suspension-based and busy-waiting based tasks in the waiting list. If the unlock request results in the grant of a suspension-based task in the waiting list, it is simply removed and added in the granted task list. On the other hand, a busy-waiting task is not removed from the waiting list immediately. Instead, the busy-waiting task is internally marked as granted but kept in the waiting list, and the resource is marked as expecting a pending busy-waiting based request. Once the corresponding busy-waiting request arrives, the task is finally removed from the waiting list and added in the granted task list.

A priority-inheritance list is separately kept corresponding to all the granted tasks. A separate priority-inheritance value for different tasks is needed to differentiate between the priority inheritance values of older tasks and those of newly granted tasks. A priority value in the inheritance list for a specific task is only added if the new priority is higher than the previous value.

5.2.9 Response

A response is a data structure that is used to carry information from RMA back to the task. Various components of the framework act on it. It contains the fields as the task id, resource id, result of request, granted task id and inherited priority. Fig. 5.8 shows the structure of response.

5.2.10 Interrupt Handler

All the inward communication to the VPU must be done through its interrupt port(s). An interrupt handling module is needed which handles the events coming onto the interrupt ports.

\footnotesize{15} To be explained later.

\footnotesize{16} The id of any waiting task that is granted access as a result of an unlock request.
An interrupt handler module simply listens to the interrupt port of VPU and requests the scheduling of the associated interrupt service routine. It is interesting to note that this listening is done in the context of underlying event driven SystemC simulation kernel, thus allowing synchronization between timed task network and SystemC kernel. The corresponding interrupt service routine also generates an event for the interrupt listening task of the corresponding IRA module. It is important to note that interrupt service routine is assigned SYSTEM_PRIORITY, the highest priority in the framework.

### 5.2.11 Schedulers

The scheduler must select the most appropriate task from the list of ready tasks provided to it. All the schedulers defined within the framework must respect the priority hierarchy defined above in section 5.2.2. For this reason, two framework-level scheduler base classes are defined, for priority-based and non-priority based scheduling algorithms. These scheduling algorithm base classes ensure that any ready framework-level task always gets scheduled first. The control is only passed to any user-defined custom scheduling algorithm if no framework-level task is in the ready state.

Two scheduling algorithms are defined as part of the core framework; priority based and round-robin based scheduling algorithms. Moreover, the framework is also compatible with any time-slice based scheduling algorithm, defined using the framework scheduler base classes.

**Priority-based Scheduling Algorithm**

Priority-based scheduling algorithm is simply a framework compliant extension of the default priority-based algorithm. Because non-preemptiveness of tasks is also built as part of priority hierarchy, it can be used with the priority-based scheduling algorithm without any change.

---

17 As defined in the section 5.2.5, differentiation between priority-based and non-priority based algorithms is necessary for proper synchronization of the overall framework.
Round-robin Scheduling Algorithm

Round-robin scheduling algorithm is a modification and extension of the original round-robin algorithm provided as part of VPU. Firstly, the algorithm is defined using the scheduling base classes for the framework. Secondly, when the tasks are suspending, waiting and later resuming, the original algorithm no more remains truly round-robin. Thirdly, support for non-preemptive tasks had to be provided within the scheduler. The new algorithm internally keeps a separate First In First Out (FIFO) based ready list. This internal list is filled up from the provided ready task list when the scheduling algorithm runs for the first time or when it gets empty. While filling the list, it is ensured that the last selected task is only added at the end. Othertimes, the internal list is updated with any new ready tasks. While updating, it is also ensured that the last found task is added at the end, if it hasn’t since become non-preemptive. Any task, that has gone from non-preemptiveness to being preemptive and is still ready to be executed, is removed from the head of the list and added at the tail. The algorithm also removes non-preemptiveness from a task that meanwhile executes a suspend or wait command. The algorithm is also shown in terms of flow-charts in figures 5.9 and 5.10.
Figure 5.9: Flow chart showing the round robin algorithm
Figure 5.10: Flow chart showing the update part of above round-robin algorithm
Chapter 6

Evaluation and Discussion

This chapter begins by summarizing the features of the proposed framework. Later the shortcomings of the framework are discussed in limitations section. The proposed framework is evaluated using metrics of expressiveness, usability and extensibility. The results of experimentation involving different configuration scenarios showing the expressiveness of the framework are also detailed. Furthermore, usability and extensibility of the framework is also discussed at the end of this chapter.

6.1 Features

This section briefly explains the functional capabilities of the proposed framework. The framework is able to model the accesses to a shared object from different multicores. These shared accesses are modeled while keeping in view the accessing tasks’ blocking behavior, scheduling policy and resource access mechanism as independent concerns.

- **Blocking Behavior**
  - Busy-waiting
  - Suspension-based blocking
- **Resource Access Mechanisms**
  - Non-preemptive shared resource access
  - Priority-inheritance based access
- **Scheduling Algorithms**
  - Round-robin based scheduling
  - Priority-based scheduling
  - Compatibility with any time-slice based scheduling scheme
- **Arbitration policy of shared resource**
  - First-In First-Out (FIFO)
- **Types of Shared Resource**
  - Shared resource as a mutex (having a single instance)
  - Shared resource as a semaphore (having multiple copies)
CHAPTER 6. EVALUATION AND DISCUSSION

Number of Shared Resources
No restriction

Type of resource accesses
- Non-nested accesses
- Nested accesses

Types of Tasks
- Non-periodic tasks

Delays
- Ability to model delays on the task-side

6.2 Limitations

This section explains the limitations of the proposed framework from modeling point-of-view:

The major limitation of the framework is that it assumes that a shared resource is located outside any virtual processing unit. The situations in which a shared resource may be located inside a virtual processing unit may also be modeled by doing some manipulation and extension of the existing components of the framework; but such scenarios are currently out-of-the-scope of the current framework.

Moreover, in the proposed framework, a busy-waiting based task may incur a delay of one additional time unit. This delay is necessary to make the framework work perfectly in any time-slice based scheduling scheme, as detailed in section 5.2.5.

The current framework only allows the modeling of delays on the task-side. Other important delay types like scheduling delays, context switching delays, propagation delays etc can also be added as part of the framework, but are not included in the current implementation.

Furthermore, the modeling situations when the overall performance of the system depends upon the ordering of concurrent shared resource accesses may not be fully conceivable. The framework imposes no total-order on such shared resource accesses but instead relies on the underlying mechanics of the simulation engine, TLM and SystemC. For example, a deadlock situation which involves a particular total-order of the occurrence of concurrent shared accesses (by two or more tasks) may or may not be captured by the framework. The reason being that such a situation depends upon the way the underlying simulation engine imposes any total-order and thus lies beyond the scope of the framework.

6.3 Metrics

This section explains various metrics against which the analysis framework is judged.
CHAPTER 6. EVALUATION AND DISCUSSION

- **Expressiveness:** The first metric is the expressiveness of the model compared to the pen-and-pencil based approaches. This metric classifies the types of scenarios that can be modeled using the framework.

- **Usability:** The second important metric is the usability or ease-of-use of the framework. This means how much effort is required to change from one configuration to another. For example, how many (if any) lines of code need to be changed, when is (if any) recompile of the systems employing this framework is necessary etc.

- **Extensibility:** This metric demonstrates how is it possible to add a new orthogonal concern like a scheduling algorithm or a resource access mechanism to the core framework.

### 6.4 Experimentation

This section explains the set of experiments used to evaluate the analysis framework in Platform Architect. It elaborates the experimental setup, set of results, and the analysis performed on the results. Earlier experiments explain a relatively simple scenario in which a set of tasks mapped on two different cores try to access a single shared resource. The later experiments elaborate a more sophisticated scenario involving nested resource accesses and semaphore-based shared resource. In the rest of the sections below, first the experimental setup and the structure of the task-set is explained. Later the configuration of each experiment, obtained results and the analysis is presented.

As explained earlier in the section of feature-set, the configurability of the framework allows a very large number of scenarios to be simulated. But to elaborate the proof-of-concept and measure other metrics, only a limited set is chosen. These different scenarios are elaborated in terms of numerous experiments.

In all subsequent experiments, a task T having an execution time ‘E’ and period ‘P’ is defined as T(E,P). An access of a single time unit to a shared resource ‘R’ is denoted as R(1). A nested access to resource ‘W’ within an access to resource ‘R’ is denoted as R(1 [W(0.5)]). Here, the inner resource access to ‘W’ is of 0.5 time units, while the access to outermost shared resource ‘R’ is of 1 unit in total. All tasks are mapped to a virtual processing unit (VPU), while a globally shared resource is modeled as a module located outside any VPU. Although separate specification of overheads (like request latencies, shared resource access latency etc) can be easily made possible, but the following experiments assume a simpler approach in which the overheads are part of each task’s execution. SCExplorer tool is used to trace the task states. Experiments 1-8 model a simple shared resource access scenario, while experiments 9 and onwards model a more sophisticated nested shared resource access example.

---

1The experiments are explained by using the terminology introduced earlier in the report and is repeated here for convenience.
The task state legend for the simulation results (described below) is shown in figure 6.1.

<table>
<thead>
<tr>
<th>State</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Green</td>
</tr>
<tr>
<td>Ready</td>
<td>Yellow</td>
</tr>
<tr>
<td>Suspended</td>
<td>Orange</td>
</tr>
<tr>
<td>Waiting</td>
<td>Red</td>
</tr>
</tbody>
</table>

Figure 6.1: Task states legend diagram

### 6.4.1 Experimental Setup (for Experiments 1-8)

Suppose we have a task set consisting of six identical priority-based tasks. We need to quickly evaluate different configurations (mapping, task access behavior, resource access mechanism and scheduling algorithm) to find out best possible strategy with regards to overall system execution time, without causing any deadlock.

These tasks model attempts to lock a particular shared resource and later unlock it. During the lock and unlock attempts, the tasks also execute a host of functions, including consuming processor time, going into a waiting state and voluntarily giving back the access to the scheduler. The exact structure of each of the tasks is shown in the figure 6.2 and also in terms of pseudo-code in listing 6.1. Fig. 6.3 shows the actual structure of the system.

These tasks access two shared resources. The shared resource access pattern for each task is R(22) for both resources. Priorities are assigned to the tasks with the priority of a task T(i) defined as Pr(i), and Pr(i)>Pr(i+1). The priorities of tasks are defined as 10, 15, 7, 6, 20 and 25. Tasks are divided into two logical sets, with members of a set always using a particular blocking policy. Tasks with priorities 10, 7 and 20 make the first set (S1), while tasks with priorities 15, 6 and 25 form the second set (S2) of tasks. Moreover, tasks always access the shared resources using priority-inheritance based approach. Furthermore, tasks with priorities 10, 15, 7 are mapped onto one VPU, while tasks with priorities 6, 20 and 25 are mapped onto the other VPU. Priority-based scheduling algorithm is always used for both VPU’s. An additional requirement is that only the tasks with priorities lower than 10 can be remapped. All the tasks execute identical code, as shown in listing 6.1. The results of all the experiments are redrawn for clarity, while the actual results are part of the appendices. As the task mapping is also variable, tasks are referred with their base priorities (instead of their id’s) in the analysis below. Each of the set of experiments is evaluated in terms of its functional behavior and expressiveness.

---

2. By identical we mean that the code executed by all the tasks is the same.
3. This means that each task keeps the lock for atleast 22 time units.
4. Lower numerical value of priority means higher importance
5. This, essentially, means that only the tasks with priorities 6 and 7 can be remapped.
At the end of the experiments, a brief evaluation in terms of the *usability* matrix is also performed.

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>Consume</th>
<th>Lock</th>
<th>Consume</th>
<th>Wait</th>
<th>Wait</th>
<th>Consume</th>
<th>Wait</th>
<th>Unlock</th>
<th>Consume</th>
<th>Wait</th>
<th>Suspend</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6.2: Task system function diagram showing the sequence of system functions used in all the tasks for experiments 1-8. The left hand side column shows the time units consumed by each system function. A wait of zero time units means that the control is voluntarily passed back to the scheduler.

### 6.4.2 Experiment 1

Example of a task set involving a combination of busy-waiting and suspension-based blocking, priority-based scheduling algorithm and a FIFO based resource access.

#### Configuration

Tasks with priorities 10, 15, 7 are mapped to one VPU, while tasks with priorities 6, 20 and 25 are mapped onto the other VPU. S1 uses suspension-based, non-preemptive shared resource access. S2 uses busy waiting based, preemptable shared resource access.
while (true)
{
    // request structure
    request req;
    // response structure
    response rsp;
    // delay structure
    sc_time my_delay = sc_time(10, SC_NS);
    // delay structure
    sc_time running_delay = sc_time(2, SC_NS);
    // setting the appropriate parameters of a request
    req.fill_request();
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // sending the lock request
    rsp = resource->lock(req);
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // voluntarily entering WAIT state for 10 time units
    tm_wait(my_delay);
    // voluntarily passing the control back to the scheduler
    tm_wait();
    // utilizing processor for 10 time units
    tm_consume(my_delay);
    // voluntarily passing the control back to the scheduler
    tm_wait();
    // sending the unlock request
    rsp = resource->unlock(req);
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // voluntarily passing the control back to the scheduler
    tm_wait();
    // utilizing processor for 10 time units
    tm_consume(my_delay);
    // voluntarily passing the control back to the scheduler
    tm_wait();
    // finished execution, thus going into SUSPENSION state
    tm_suspend();
}
CHAPTER 6. EVALUATION AND DISCUSSION

Figure 6.3: System diagram for experiments 1-8, showing two VPU’s on the left competing for two shared resources on the right. The VPU’s and shared resources are connected through a bus.

Results

Figures 6.4 and 6.5 show the results of the simulation.

Analysis

VPU1

Task with priority 7 makes non-preemptive access from 2 time units till 24 time units.

Task with priority 10 goes into suspended state at 6 time units and resumes again at 46 time units. It accesses the resource from 46 time units till 68 time units.

Task with priority 15 accesses shared resource at 8 time units and goes into busy-waiting state till 86 time units. It relinquishes the resource at 108 time units. All the tasks finish their execution at 120 time units.

VPU2

Task with base priority 6 successfully accesses the shared resource at 2 time units and gives it up at 24 time units.

Task with base priority 20 accesses the resource at 6 time units and immediately goes into suspended state till 24 time units, when it non-preemptively accesses the shared resource till 46 time units.

Task with base priority 25 accesses the shared resource at 8 time units and immediately goes into busy-waiting state till 62 time units. It accesses the shared resource from 62 time units till 84 time units. Meanwhile, it also inherits the priority of 15 from 82-84 time units.

It is interesting to note that R2 becomes free at 24 time units. But at 8 time units, the request of task with base priority 25 had reached before the request of task having base priority 15. So, the resource is only granted to task having base priority 25 in FIFO order at 62 time units.
CHAPTER 6. EVALUATION AND DISCUSSION

Figure 6.4: Simulation results for VPU1 in experiment 1. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Base priority 10 task remains in suspended state from 6-46 time units. Base priority 15 task does busy waiting from 8-86 time units.

6.4.3 Experiment 2

Example of a task set involving a combination of busy-waiting and suspension-based blocking, priority-based scheduling algorithm and a FIFO based resource access.

Configuration

Task sets S1 and S2 switch their blocking methods, with S1 using busy waiting based, preemptable shared resource access and S2 using suspension-based, non-preemptive shared resource access. Other configuration remains the same as the former experiment.

Results

Figures 6.6 and 6.7 show the results of the simulation.
Figure 6.5: Simulation results for VPU2 in experiment 1. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Base priority 20 task is in suspension state from 6-24 time units. Base priority 25 task does busy-waiting from 8-62 time units.

Analysis

VPU1
Task with base priority 7 accesses the shared resource at 2 time units, while relinquishes it at 24 time units. It finishes its execution at 36 time units. While this, highest priority task is executing, no other task is able to execute. (only in the window of \texttt{tm\_wait( )}, any other task could execute)

Task with base priority 10 starts busy-waiting at 6 time units, till 72 units. It gives up the lock at 104 time units.

Task with base priority 15 gets access to the shared object at 76 time units and gives it up at 98 time units. It finishes its execution at 128 time units as system’s last task.

VPU2
Task with base priority 6 accesses the lock at 2 time units and gives it up at 24 units.

Task with base priority 20 accesses the lock at 6 time units and starts busy-waiting. It ends the busy-waiting and gets the lock at 38 time units. It also
immediately inherits the priority of 10, also at 38 units. It gives up the lock at 70 time units, when it also regains its original priority.

Task with base priority 25 accesses the lock non-preemptively at 42 time units and gives up the lock at 64 time units (when it regains its original priority). It finishes its execution at 94 time units as the last task of the system.

It is interesting to note that R₁ and R₂ both become free at 24 time units. R₁ gets again allocated at 38 time units to task with base priority 20. R₂ gets again allocated at 42 time units.

### 6.4.4 Experiment 3

Example of a task set involving busy-waiting based preemptive shared resource access, priority-based scheduling algorithm and a FIFO based resource access.

#### Configuration

In this experiment, both S₁ and S₂ use busy-waiting based, preemptable shared resource access. The rest of the configuration remains the same as the former
CHAPTER 6. EVALUATION AND DISCUSSION

Figure 6.7: Simulation results for VPU2 in experiment 2. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Base priority 20 task does busy-waiting from 6-38 time units.

Results

Figures 6.8 and 6.9 show the results of the simulation.

Analysis

VPU1

Task with base priority 7 locks the shared resource at time 2 units. It gives up the shared resource at 24 time units.

Task with base priority 10 accesses the resource at 6 time units and immediately starts busy-waiting. It becomes successful at 62 time units and gives up the lock at 84 time units.
Figure 6.8: Simulation results for VPU1 in experiment 3. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 does busy-waiting from 6-62 time units. Task with base priority 15 does busy-waiting from 66-98 time units.

Task with base priority 15 accesses the resource at 66 time units and immediately starts busy-waiting. It finally gets the lock at time 98 time units and gives up the lock at 120 time units. Also being the last task, it finishes its execution at 132 time units. (with 132 time units also being the execution time of the system)

VPU2

Task with base priority 6 gets the lock to the shared resource at 2 time units and gives it up at 24 time units.

Task with base priority 20 accesses the shared resource at time 6 time units and immediately enters busy-waiting state. It gets the lock at 38 time units and also inherits the priority of 10 from the corresponding task. It gives back the lock and also regains its original priority at 20 time units.

Task with base priority 25 locks the shared resource at 42 time units. It inherits the priority of 15 at 66 time units. It unlocks the resource at 76 time units and also regains its original priority. It ends its execution at 94 time units.
Figure 6.9: Simulation results for VPU2 in experiment 3. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 does busy-waiting from 6-38 time units.

It is interesting to note that R1 and R2 become free at 24 time units. Task with base priority 20 was added in the FIFO resource list earlier than task with base priority 10. That is why task with base priority 10 needs to wait.

6.4.5 Experiment 4

Example of a task set involving suspension-based non-preemptive shared resource access, priority-based scheduling algorithm and a FIFO based resource access.

Configuration

In this experiment, both S1 and S2 use suspension-based, non-preemptive shared resource access. The rest of the configuration remains the same as the former experiment.
Results

Figures 6.10 and 6.11 show the results of the simulation.

Figure 6.10: Simulation results for VPU1 in experiment 4. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 remains in suspension state from 6-56 time units. Task with base priority 15 remains in suspension state from 8-58 time units.

Analysis

VPU1

Task with base priority 7 accesses the resource at 2 time units and gives it up at 24 time units.

Task with base priority 10 accesses the resource at 6 time units and immediately goes into suspension state. It finally gets the access to the shared resource at 56 time units. It non-preemptively accesses the shared resource till it voluntarily gives up the control back to the scheduler at 78 time units.
Figure 6.11: Simulation results for VPU2 in experiment 4. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 goes into suspension-state from 6-24 time units. Task with base priority 25 goes into suspension-state from 8-24 time units. Task with base priority 15 accesses the shared resource at 8 time units and immediately goes into suspension state. It gets the lock at 58 time units and releases it at 88 time units. It finishes its execution at 112 time units. It is interesting to note that it is possible for two tasks to have non-preemptive privileges at the same time, if one of the tasks involves a wait( ) statement. In that case, the scheduler arbitrarily selects one of the ready non-preemptive tasks.

**VPU2**

Task with base priority 6 accesses the shared resource from 2-24 time units. It finally finishes at 60 time units.

Task with base priority 20 attempts to get the lock at 6 time units and releases the lock at 88 time units. It finally finishes its execution at 100 time units.
immediately goes into suspension state. It finally gets access to the lock at 24 time units. It non-preemptively accesses the lock till 46 time units, at that time it voluntarily gives up the execution to another non-preemptive task (with priority 25). It gives up the lock at 56 time units. It completes its execution at 72 time units.

Task with base priority 25 attempts to get the lock at 8 time units and immediately goes into suspension state. It gets the access to the lock at 24 time units but cannot resume its execution till 26 time units when the non-preemptive task (with priority 20) voluntarily goes into the waiting state. It relieves the lock at 56 time units. It finally completes its execution at 84 time units.

6.4.6 Experiment 5

Example of a task set involving a combination of busy-waiting and suspension-based blocking, priority-based scheduling algorithm, a FIFO based resource access and an altered mapping of tasks.

Configuration

We have already gone through all the possible blocking configurations for a given mapping. Let us observe the system behavior when we remap the tasks with priorities 6 and 7 to VPU’s 1 and 2 respectively. The set of tasks S1 uses suspension-based, non-preemptive shared resource access. S2 uses busy waiting based, preemptable shared resource access. The rest of the configuration remains the same as the former experiment.

Results

Figures 6.12 and 6.13 show the results of the simulation.

Analysis

VPU1

The task with base priority 6 successfully accesses the lock at 2 time units and gives it up at 24 time units.

The task with base priority 10 attempts the lock at 6 time units and immediately goes into suspension state. It gains the access to the resource at 46 time units and non-preemptively accesses it till 68 time units. It finishes its execution at 80 time units.

The task with base priority 15 unsuccessfully attempts at 8 time units and immediately enters the busy-waiting state. It finally gains access to the resource at 86 time units and gives it up at 108 time units. It finishes its execution at 120 time units.

VPU2

The task with base priority 7 successfully gains access at 2 time units and gives up at 24 time units.
Figure 6.12: Simulation results for VPU1 in experiment 5. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 remains in suspension-state from 6-46 time units. Task with base priority 15 does busy-waiting from 8-86 time units.

The task with base priority 20 unsuccessfully attempts lock at 6 time units and subsequently enters suspension state. It gains non-preemptive access at 24 time units and gives up the resource at 46 time units. It finishes its execution at 60 time units.

The task with base priority 25 unsuccessfully attempts lock at 8 time units and immediately starts busy-waiting. It becomes successful at 62 time units and gives up the lock at 84 time units. Between 82-84 time units, it inherits the priority of 15. It finishes its execution at 96 time units.

Task with base priority 25 gets access earlier than task with base priority 15 because of FIFO based waiting.
CHAPTER 6. EVALUATION AND DISCUSSION

88

Figure 6.13: Simulation results for VPU2 in experiment 5. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 goes into suspension-based state from 6-24 time units. Task with base priority 25 does busy-waiting from 8-62 time units.

6.4.7 Experiment 6

Example of a task set involving a combination of busy-waiting and suspension-based blocking, priority-based scheduling algorithm, a FIFO based resource access and an altered mapping of tasks.

Configuration

Task sets S1 and S2 switch their blocking methods, with S1 using busy waiting based, preemptable shared resource access and S2 using suspension-based, non-preemptive shared resource access. The rest of the configuration remains the same as the former experiment.
CHAPTER 6. EVALUATION AND DISCUSSION

Results

Figures 6.14 and 6.15 show the results of the simulation.

![Simulation Results Diagram]

Figure 6.14: Simulation results for VPU1 in experiment 6. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 does busy-waiting from 6-72 time units.

Analysis

VPU1

Task with base priority 6 successfully accesses the lock from 2 time units-24 time units. It finishes its execution at 36 time units.

Task with base priority 10 attempts lock at 6 time units and enters busy-waiting till 72 time units. It enters the lock at 72 time units and gives up the lock at 104 time units. It finishes its execution at 116 time units.

Task with base priority 15 accesses the resource from 76 time units-98 time units. It finishes its execution at 128 time units.

VPU2
Figure 6.15: Simulation results for VPU2 in experiment 6. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 does busy-waiting from 6-38 time units.

Task with base priority 7 successfully attempts the lock at time 2 time units and gives it up at 24 time units.

Task with base priority 20 accesses the shared resource at time 6 time units and enters busy-waiting till it gains access at 38 time units. It also inherits the priority of 10 from the corresponding task at 38 time units. It gives up the lock at 70 time units and also regains its original priority. It finishes at 82 time units.

Task with base priority 25 successfully and non-preemptively accesses the lock at 42 time units. It gives up the lock at 64 time units and eventually finishes at 94 time units.

6.4.8 Experiment 7

Example of a task set involving busy-waiting based preemptive shared resource access, priority-based scheduling algorithm, a FIFO based resource access and an altered mapping of tasks.
Configuration

In this experiment, both S1 and S2 use busy waiting-based, preemptable shared resource access. The rest of the configuration remains the same as the former experiment.

Results

Figures 6.16 and 6.17 show the results of the simulation.

Figure 6.16: Simulation results for VPU1 in experiment 7. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 does busy-waiting from 6-62 time units. Task with base priority 15 does busy-waiting from 66-98 time units.

Analysis

VPU1

Task with base priority 6 successfully gets the lock from 2 time units-24 time units. It finishes its execution at 36 time units.
Figure 6.17: Simulation results for VPU2 in experiment 7. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 does busy-waiting from 6-38 time units.

Task with base priority 10 attempts lock at 6 time units and goes into busy-waiting state till 62 time units. At 62 time units, it gets the lock and at 84 time units, it gives up the lock. It finishes its execution at 96 time units.

Task with base priority 15 attempts the lock at 66 time units and goes into busy-waiting state till 98 time units. It accesses the lock from 98 time units till 110 time units. It finishes its execution at 132 time units.

VPU2

Task with base priority 7 successfully gets the lock from 2 time units-24 time units.

Task with base priority 20 starts busy waiting from 6 time units-38 time units. It gives up the resource at 60 time units. It inherits the priority of 10 at 38 time units and also relinquishes it once it finishes its execution.

Task with base priority 25 gets access to the shared resource at 42 time units. It inherits priority 15 at 66 time units, and gives up the resource and also the
priority at 76 time units. It is also the last task to finish at 94 time units.

6.4.9 Experiment 8

Example of a task set involving suspension-based non-preemptive shared resource access, priority-based scheduling algorithm, a FIFO based resource access and an altered mapping of tasks.

Configuration

In this experiment, both S1 and S2 use suspension-based, non-preemptive shared resource access. The rest of the configuration remains the same as the former experiment.

Results

Figures 6.18 and 6.19 show the results of the simulation.

Analysis

VPU1

Task with base priority 6 non-preemptively accesses the lock from 2-24 time units. It finishes its execution at 36 time units.

Task with base priority 10 unsuccessfully attempts lock at 6 time units and enters suspension state. It remains suspended till 56 time units. It non-preemptively accesses the lock from 56 time units till it gives up the lock at 88 time units.

Task with base priority 15 unsuccessfully attempts lock at 8 time units and enters suspension state. It remains suspended till 58 time units. It non-preemptively accesses the lock from 56 time units till it gives up the lock at 88 time units. It is also the last task to finish at 112 time units.

VPU2

Task with base priority 7 successfully accesses the lock at 2 time units and gives it up at 24 time units. It completes its execution at 60 time units.

Task with base priority 20 unsuccessfully attempts lock at 6 time units and enters suspension state. It remains suspended till 24 time units. It non-preemptively accesses the lock from 24 time units till it gives up the lock at 56 time units. It finishes its execution at 72 time units.

Task with base priority 25 unsuccessfully attempts lock at 8 time units and enters suspension state. It remains without lock till 24 time units. It is able to resume and non-preemptively access the lock at 26 time units. It gives up the lock at 56 time units. It is also the last task to finish at 84 time units.

The results show that no deadlock situation occurs in any of the simulations of the experiments. Moreover, the results show that system’s response time is the largest when both groups of tasks (S1 and S2) use busy-waiting blocking. This
Figure 6.18: Simulation results for VPU1 in experiment 8. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 10 goes into suspension-state from 6-56 time units. Task with base priority 15 goes into suspension-state from 8-58 time units.

happens in case of experiments 3 and experiment 7, and the system finishes its execution at 132 time units. The earliest response time of the system occurs when both the group of tasks (s1 and s2) use suspension-based blocking. In that case, system is able to finish its execution at 112 time units. Moreover, it is interesting to note that remapping does not cause any change in the behavior of tasks, and the system's response remains the same.6

```c
while (true) {
    // request structure
}
```

---

6It is also possible to easily verify the null affect of remapping through intuition because the remapped tasks have the same structure and are the highest priority tasks on their respective VPU’s.
Figure 6.19: Simulation results for VPU2 in experiment 8. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. Task with base priority 20 goes into suspension-state from 6-24 time units. Task with base priority 25 goes into suspension-state from 8-24 time units.

```c
request req;
// response structure
response rsp;
// delay structure
sc_time my_delay=sc_time(10,SC_NS);
// delay structure
sc_time running_delay=sc_time(2,SC_NS);
// setting the appropriate parameters of a request
req.fill_request();
// utilizing processor for 2 time units
tm_consume(running_delay);
// sending the first lock request for resource1
rsp=resource->lock(req);
```
// sending the second lock request for the same resource
rsp=resource->lock(req);
// setting the appropriate parameters of a request
req.set_resource_id(resource_id2);
// sending the third lock request for resource 2
rsp=resource->lock(req);
// setting the appropriate parameters of a request
req.set_resource_id(resource_id3);
// sending the fourth lock request for resource 3
rsp=resource->lock(req);
// utilizing processor for 2 time units
tm.consume(running_delay);
// voluntarily passing the control back to the scheduler
tm.wait();
// utilizing processor for 10 time units
tm.consume(my_delay);
// voluntarily passing the control back to the scheduler
tm.wait();
// sending the first unlock request for resource 3
rsp=resource->unlock(req);
// utilizing processor for 2 time units
tm.consume(running_delay);
// setting the appropriate parameters of a request
req.set_resource_id(resource_id2);
// sending the unlock request for resource 2
rsp=resource->unlock(req);
// utilizing processor for 2 time units
tm.consume(running_delay);
// setting the appropriate parameters of a request
req.set_resource_id(resource_id1);
// sending the first unlock request for resource 1
rsp=resource->unlock(req);
// sending the second unlock request for resource 1
rsp=resource->unlock(req);
// voluntarily passing the control back to the scheduler
tm.wait();
// utilizing processor for 10 time units
tm.consume(my_delay);
// voluntarily passing the control back to the scheduler
tm.wait();
// finished execution, thus going into
CHAPTER 6. EVALUATION AND DISCUSSION

Listing 6.2: Structure of the first task (T1) mapped on VPU1 in experiments 9 and 10.

```c
//SUSPENSION state
tm_suspend();
}
```

```c
while(true)
{
    //request structure
    request req;
    //response structure
    response rsp;
    //delay structure
    sc_time my_delay=sc_time(10,SC_NS);
    //delay structure
    sc_time running_delay=sc_time(2,SC_NS);
    //setting the appropriate parameters of a request
    req.fill_request();
    //utilizing processor for 2 time units
    tm_consume(running_delay);
    //sending the first lock request for resource1
    rsp=resource->lock(req);
    //sending the second lock request, again
    rsp=resource->lock(req);
    //for resource1
    rsp=resource->lock(req);
    //setting the appropriate parameters of a request
    req.set_resource_id(resource_id3);
    //sending the third lock request for resource3
    rsp=resource->lock(req);
    //setting the appropriate parameters of a request
    req.set_resource_id(resource_id2);
    //sending the fourth lock request for resource2
    rsp=resource->lock(req);
    //utilizing processor for 2 time units
    tm_consume(running_delay);
    //voluntarily passing the control back to
    //the scheduler
    tm_wait();
    //utilizing processor for 10 time units
    tm_consume(my_delay);
    //voluntarily passing the control back to
    //the scheduler
    tm_wait();
    //sending the unlock request for resource2
    rsp=resource->unlock(req);
}
```
Listing 6.3: Structure of the second task (T2) mapped on VPU1 in experiments 9 and 10.

6.4.10 Experimental Setup (for Experiments 9 and 10)

Experiments 9 and 10 are designed to show the ability of the framework to model semaphore-based shared resources and nested locks. These experiments also show the subtle effects observed by the selection of a scheduling algorithm. This ability is shown in terms of a hypothetical scenario involving three processors and three shared resources. The first resource models a semaphore-based resource, allowing two tasks to simultaneously access it. The other resources model a mutex-based resource, providing only mutually exclusive access. Two tasks are mapped on the first processor, while the other processors contain only one task mapped onto them. It is assumed that each of the tasks makes a priority-inheritance based shared resource access. The first task on the first processor uses suspension-based blocking, while the second task on the first processor uses busy-waiting based blocking. The structure of each task and the shared access strategy are shown in figures 6.20, 6.21 and 6.22. The structures of the tasks are also elaborated in terms of pseudocode in listings 6.2, 6.3 and
Figure 6.20: Task system function diagram showing the sequence of system functions used in the first task (T1) on VPU1. The left hand side column shows the time units consumed by each system function. A wait of zero time units means that the control is voluntarily passed back to the scheduler.
Figure 6.21: Task system function diagram showing the sequence of system functions used in the second task (T2) on VPU1. The left hand side column shows the time units consumed by each system function. A wait of zero time units means that the control is voluntarily passed back to the scheduler.
while(true) {
    // request structure
    request req;
    // response structure
    response rsp;
    // delay structure
    sc_time my_delay = sc_time(10, SC_NS);
    // delay structure
    sc_time running_delay = sc_time(2, SC_NS);
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // sending the first lock request for resource
    rsp = resource->lock(req);
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // voluntarily passing the control back to
    // the scheduler
    tm_wait();
    // utilizing processor for 10 time units
    tm_consume(my_delay);
    // voluntarily passing the control back to
    // the scheduler
    tm_wait();
    // sending the unlock request
    rsp = resource->unlock(req);
    // utilizing processor for 2 time units
    tm_consume(running_delay);
    // voluntarily passing the control back to
    // the scheduler
    tm_wait();
    // utilizing processor for 10 time units
    tm_consume(my_delay);
    // voluntarily passing the control back to
    // the scheduler
    tm_wait();
    // finished execution, thus going into
    // SUSPENSION state
    tm_suspend();
}
Figure 6.22: Task system function diagram showing the sequence of system functions used in the tasks (T3 and T4) on VPU2 and VPU3 respectively. The left hand side column shows the time units consumed by each system function. A wait of zero time units means that the control is voluntarily passed back to the scheduler.
In terms of terminology defined above, the nested lock access pattern for the first task on VPU1 is defined as $R_1(16 R_1(16 R_2(14 R_3(12))))$ while the nested access pattern for the second task on VPU1 is defined as $R_1(16 R_1(16 R_3(14 R_2(12))))$. The access pattern for tasks mapped on VPU2 and VPU3 is defined as $R(12)$, where $R$ means $R_3$ and $R_2$ for VPU2 and VPU3 respectively.

Tasks mapped on VPU1 are referred as T1 and T2, while tasks mapped on VPU2 and VPU3 are referred as T3 and T4 respectively. Figure 6.23 shows the actual structure of the system.

**6.4.11 Experiment 9**

Example of a task set involving a semaphore-based resource, nested locks, combination of a suspension-based non-preemptive shared resource access and busy-waiting based non-preemptive shared resource access, round-robin based scheduling algorithm, and FIFO based resource access.

---

7 This means that the two locks to $R_1$ are the outermost ones and are kept for 16 time units each. $R_2$ is the inner nested lock and is kept for 14 time units. $R_3$ is the innermost nested lock and is kept for 12 time units.

8 This means that the two locks to $R_1$ are the outermost ones and are kept for 16 time units each. $R_3$ is the inner nested lock and is kept for 14 time units. $R_2$ is the innermost nested lock and is kept for 12 time units.

9 This means that the lock is kept for 12 time units.
CHAPTER 6. EVALUATION AND DISCUSSION

Configuration

In this experiment, VPU1 uses round-robin based scheduling policy.

Results

Figures 6.24 and 6.25 show the results of the simulation.

Analysis

VPU1

Task T1 successfully gains the nested lock access to resource 1 at 2 time units and becomes non-preemptive. But its access to resource 2 fails and it goes into suspension state till 14 time units; since, resource 2 had already been locked by T4. It ends its suspension state at 14 time units and executes non-preemptively till 30 time units. At 30 time units, it gives up all its locks and again becomes preemptive. Due to round-robin scheduling, it gives up the control of VPU to T2 after becoming preemptive. It is able to re-execute at 48 time units. It voluntarily gives the control back to the scheduler at 58 time units. It finishes its execution at 68 time units.

T2 starts its execution at 2 time units, while T1 is in the suspension state. It attempts the first of the locks at 4 time units and immediately enters busy-waiting. When T1 finally wakes up, T2 loses the processor due to round-robin scheduling policy. T2 is able to execute only when T1 gives up all its locks at 30 time units and again becomes preemptive. T2 is able to end its busy-waiting at
Figure 6.25: Simulation results for VPU2 and VPU3 in experiments 9 and 10. The end or the beginning of each square in the figure of task states indicates the point where the scheduler gains the control. A small square in the figure of task states corresponds to two time units. The task finishes its execution at 26 time units.

32 time units and gain access to all its locks. T2 enters non-preemptive state as soon as it gains access to its first lock. T2 again becomes preemptive when it relinquishes all its locks at 48 time units and T1 is able to execute again due to round-robin scheduling. T2 is able to execute again when T1 voluntarily gives back the control to the scheduler at 58 time units. It finishes its execution at 68 time units.

**VPU2**

T3 successfully accesses the lock to the shared resource 3 at 2 time units. It finishes its execution at 26 time units.

**VPU3**

T4 successfully accesses the lock to the shared resource 2 at 2 time units. It finishes its execution at 26 time units.

6.4.12 Experiment 10

Example of a task set involving a semaphore-based resource, nested locks, combination of a suspension-based non-preemptive shared resource access and busy-waiting based non-preemptive shared resource access, priority based scheduling algorithm, and FIFO based resource access.

**Configuration**

The configuration of experiment 10 is the same as in experiment 9 except that T2 on VPU1 has been assigned a priority of 4, while T1 on the same VPU is assigned the priority of 5. The scheduling algorithm on VPU1 is changed to priority-based scheduling.

**Results**

Figures 6.26 and 6.25 show the results of the simulation.
Analysis

VPU1
Due to priority-based scheduling, task with priority 4 executes first and gains nested access to the shared resource $R_1$ at 2 time units. When it tries to gain access to $R_3$, it immediately enters busy-waiting state (since $R_3$ has already been allocated to T3). It resumes from busy-waiting state at 16 time units and accesses all its locks. It finishes its execution at 42 time units.

Task with priority 5 starts its execution at 42 time units and gains access to all the requested locks at 44 time units. It finishes its execution at 70 time units.

VPU2
T3 successfully accesses the lock to the shared resource 3 at 2 time units. It finishes its execution at 26 time units.

VPU3
T4 successfully accesses the lock to the shared resource 2 at 2 time units. It finishes its execution at 26 time units.

6.5 Usability
This section explains how the core features of the framework can be used in any user-defined system built in Platform Architect, and how many changes in the system are needed when altering between different features of the framework.
The components of the framework, including the shared resource model, must be defined in the system in which the shared resource accesses are needed to be modeled. In order to use the services of the system, a user-defined task must inherit from the appropriate task class-set of the framework and use the inherited resource port to access the shared resource. The task may programmatically specify the appropriate blocking method and/or the shared resource access mechanism. The appropriate schedulers must be registered with the simulation engine and also specified in the virtual processing unit. The type of the shared resource model, that whether it is semaphore or mutex based, must also be specified.

The rest of the section explains the usability matrix in terms of changes required in the system when moving from one configuration to another.

6.5.1 Blocking methods and Resource Access Mechanisms

For blocking methods, a tradeoff needs to be made between flexibility and re-compilation time. Being flexible means enabling a task to use any blocking method for any given resource. Thus, such a flexible configuration requires the programmatic specification of a given blocking method but in turn requiring a recompile of the whole system. If we fix a given blocking method for a given resource or for a given task, then such a blocking method can be specified through the Graphical User Interface (GUI) of the tool, and thus such a configuration by-passes a complete recompile\textsuperscript{[10]}

In the current framework, the blocking methods are programmatically specified.

6.5.2 Scheduling

A scheduler is specified through a GUI-based dialog for a particular VPU, and using a configuration file to initialize it in sc\_main\( ()\) function. Thus, a change of a scheduler always requires a recompile of the whole system.

6.5.3 Number of Instances of a Resource

The framework allows for specifying the number of instances of a resource by updating a GUI-based variable of the particular Resource Management Abstraction (RMA). Thus, a system recompile is not required when updating the number of instances of a particular resource.

6.5.4 Mapping

The change in the mappings of a task set always requires a complete recompile of the whole system.

\textsuperscript{10}In Platform Architect, such a specification can be easily made by using XML file based inputs
6.6 Extensibility

This section elaborates how the core framework may be extended to add new types of orthogonal concerns like a scheduling algorithm, a resource access mechanism, a task set or a blocking method.

6.6.1 Scheduling Algorithm

Any user-defined scheduling algorithm that needs to be used with the framework can be added in a modular manner. Such a scheduling algorithm must derive itself from the scheduling base classes of the framework, keeping in view if it is priority-based or not. Moreover, non-preemptive code section of Internal Resource Access (IRA) abstraction must be enabled. The class diagram, shown in figure 6.27, elaborates the extensibility of the scheduling algorithm.

![Class diagram showing scheduler extensibility](image)

Figure 6.27: Class diagram showing scheduler extensibility

6.6.2 Resource Access Mechanism

A resource access mechanism is inherently tightly coupled with several aspects of a system modeling shared resource accesses. Depending upon the type of the shared resource access mechanism, changes may be needed at several points.

\[\text{The differentiation between priority-based and non-priority based algorithms is needed to ensure the correct synchronization behavior of the overall framework in all possible situations.}\]
in the framework. For example, for management at the resource-side, the 
\texttt{resource\_access\_mechanism()} function needs to be overloaded in Resource Management Abstraction (RMA) module. Moreover, the already-existing services of the framework like non-preemptive shared resource access and priority-inheritance may also be used as part of such a framework. Furthermore, the \texttt{process\_response()} function in Internal Resource Abstraction (IRA) module may also need to be extended to fulfill the requirements of a shared resource access mechanism. Furthermore, the fields of \texttt{request} and associated \texttt{response} may also need to be updated. Figure 6.28 shows the same points of interest when adding a new resource access mechanism.

**Figure 6.28: Points of interest when extending the framework with a new resource access mechanism**

The use-case of extending the framework by Multiprocessor Priority Ceiling based Protocol (MPCP) is briefly explained as follows. First each prerequisite or functional requirement is mentioned, and later it is explained that how the requirement can be met within the framework.

**Knowledge of Resource Ceiling**

Each Resource Management Abstraction (RMA) must maintain the resource ceiling information of the resource(s) managed by it. This can either be built into RMA or managed as a GUI-based input argument.

**Knowledge of Lowest Priority Task**

This is not needed in the current architecture of the framework since the resources are located outside any processing unit.

**Knowledge of System Priority Ceiling**

If a single RMA unit is used for the management of all resources, then it can internally maintain the value of \texttt{system\_priority\_ceiling}. If multiple RMA units are used for multiple resources, then a granted \texttt{response} must contain a new field with the \texttt{resource\_ceiling} value. Moreover, Internal Resource Abstraction (IRA) must generate an extra transaction for all RMA units. This transaction must in-turn contain information about the resource ceiling of the

\footnote{The prerequisites and functional requirements of MPCP have already been explained in earlier sections}

\footnote{Platform Architect allows for the input of GUI-based arguments as XML file based inputs.}
successful grant. All RMA units will use this new information to judge if system priority ceiling value needs to be updated or not. RMA also needs to overload \texttt{resource\_access\_mechanism()} function to check the value of system priority ceiling everytime a request comes to it.

**Dynamic Priorities**

When the priorities of the accessing tasks are dynamic, the resource priority ceilings must also be updated. Thus, whenever the priority of a task gets updated, an extra transaction must be generated by Internal Resource Access (IRA) abstraction towards the shared resources which are accessed. This is done by extending the \texttt{process\_response()} function in IRA.

**Reverting to Original Priority**

A task (with inherited priority) can only return to its original priority, if it is not holding any resource with a resource priority ceiling greater than or equal to its current priority. Thus, the successful response must contain an extra field of resource ceiling, and this field must be maintained in a list by IRA.

**Allocation of a Resource**

The \texttt{check\_resource\_access\_mechanism()} function needs to be overloaded in an RMA unit. The greatest priority ceiling among all the resources, currently held by the requesting task, must be sent as part of the request to an RMA unit.

### 6.6.3 Task Set

More sophisticated types of tasks may be introduced by extending the existing task-set hierarchy of the framework. Such tasks need to inherit from the appropriate class of the framework, depending upon where they fit in the hierarchy.

### 6.6.4 Blocking Method

Like a shared resource access mechanism, a blocking method requires to interact with various components of the system to ensure the required correct behavior. So, in order to add a new blocking method in the core framework, the \texttt{process\_response()} function in Internal Resource Access (IRA) abstraction module needs to be extended. Moreover, new fields maybe needed to be introduced in the request and response classes of the framework. Depending upon the type of the new blocking method, there may even be a need to add a new scheduler class.

---

14 This also means that a connection must exist between all processing units and all RMA’s.
15 This also means that each IRA unit contains the list of resources accessed by the associated task through a GUI-based input, available through XML file based arguments.
6.6.5 Shared Resource Arbitration Policy

The default arbitration policy for shared resources is First In First Out (FIFO). But it is possible to add a new arbitration policy by overloading the `add_waiting()` function of the `resource` class.
Chapter 7

Conclusion and Future Directions

This chapter concludes the thesis by summarizing important results obtained during the thesis. Moreover, it also explains possible future extensions to the proposed framework.

7.1 Conclusion

Multicore platforms not only provide greater flexibility to system architects but also pose additional challenges. These additional challenges lie in the form of the optimal mapping of the target application (modeled in the form of a group of tasks) on the available set of processing and communication resources. Further challenges emerge when considering the subtle effects of real-time operating systems, shared resources and resource access policies.

This thesis work proposes a generic and flexible framework enabling the analysis of shared resources in multicore platforms. It builds upon the work of [18] when actually realizing the proposed analysis framework. The analysis framework gives much greater flexibility to a system architect by treating different entities as orthogonal concerns. In particular, the entities like scheduling algorithms, blocking methods and resource access mechanisms are considered the main orthogonal concerns. Thus, a system architect can analyze the effects of using an arbitrary combination of scheduling algorithms, blocking methods and resource access mechanisms, while deciding upon a suitable mapping of a task set upon a given multicore platform. Therefore, in addition to evaluating a particular mapping with an abstract performance model, the effects of shared resource accesses can also be evaluated early in the design cycle. Moreover, the framework also provides the ability to analyze the effects of nested resource accesses upto any arbitrary level, as well as the ability to separately define mutex and semaphore based shared resources. Furthermore, it is also possible to add further orthogonal concerns like the arbitration method for waiting tasks on a
particular shared resource. What’s more, the generic design of the framework makes it suitable for a wide variety of scheduling algorithms, resource access mechanisms and blocking methods.

### 7.1.1 Limitations

The major limitation of the proposed framework is that it only considers the form of shared resources which are located outside any processing unit. The shared resources which may be located inside a processing unit fall out of the scope of the current framework, and may be considered in an extension of the framework.\(^1\)

### 7.1.2 Summary of thesis

A brief, chapter-wise summary of the whole thesis documents appears in this section. Chapter 1 (Introduction) gives the motivation for pursuing the thesis, explains the problem statement and discusses the main contributions. Chapter 2 (Background) gives the necessary background by explaining the traditional design flow and necessary nomenclature. It also gives a detailed survey of scheduling algorithms and resource access mechanisms for both uni and multicore scenarios. Contemporary approaches to modeling shared resources are also briefly presented in chapter 2. Chapter 3 (Functional Requirements of the Framework) discusses the functional requirements of a generalized and flexible shared-resource analysis framework in terms of blocking methods, resource access mechanisms and associated scheduling algorithms. Chapter 4 (Design of the Framework) presents the object-oriented design of the whole framework, and elaborates the main components of the framework. Chapter 5 (The Shared Resource Analysis Framework) discusses the implementation of the proposed framework using the abstract task-modeling concepts presented in \[18\]. Chapter 6 (Evaluation and Discussion) evaluates and discusses the proposed framework using metrics of expressiveness, usability and extensibility.

### 7.2 Future Directions

To increase the expressiveness and usability, the proposed framework can be extended with more flexible types of shared resources, more sophisticated resource access algorithms and blocking methods, new orthogonal concerns, and dynamic scheduling approaches. Moreover, the framework also needs to be validated using bigger case-studies involving realistic applications, modeled through abstract task-modeling concepts. Possible future directions of work are discussed in this section.

---

\(^1\)A more detailed description of limitations appears in section 6.2.
CHAPTER 7. CONCLUSION AND FUTURE DIRECTIONS

7.2.1 Flexible Shared Resources

The current framework allows all shared resources to be only located outside all processing units. But there exist other types of resources as well which may be located inside a particular processing unit and are requested by tasks located on other processing units. Moreover, some of these processor-specific resources may only be accessed by only local tasks.\footnote{the tasks located on the same processing unit} Thus, a possible future work is to extend the proposed framework allowing more flexible shared resources.

7.2.2 More Sophisticated Resource Access Mechanisms and Blocking Methods

The current form of the framework includes non-preemptive critical sections and priority inheritance as the built-in resource access mechanisms. Moreover, two blocking methods which enable busy-waiting and suspension based waiting are part of the framework. There exist a possibility to extend the framework with more sophisticated resource access mechanisms (like Multiprocessor Priority Ceiling Protocol (MPCP) etc) and new blocking methods as described in chapter 4 and evaluated in sections 6.6.2 and 6.6.4.

7.2.3 New Orthogonal Concerns

The current framework treats scheduling policies, blocking methods and resource access mechanisms as three major orthogonal concerns. There exist a possibility to add new orthogonal concerns as presented in the generic design in chapter 4 and as evaluated in section 6.6.5.

7.2.4 Additional Scheduling Algorithms

The proposed framework currently boasts of only two static scheduling algorithms, namely round-robin and priority-based. As a future work, the framework can be extended with new scheduling algorithms (especially dynamic priority ones), using the generic design described in section 4.2 and as evaluated in section 6.6.1.

7.2.5 Modeling of Delays

The current implementation of the shared resource framework allows for the modeling of delays at the task-side. The expressiveness of the framework can be increased by adding the ability to model other types of delays connected to a shared resource access like scheduling delays, context switching delays, propagation delays etc.
7.2.6 Validation in Real World Scenarios

The performance of the current framework hasn’t been compared with shared resource scenarios occurring in the real world. There exists a need to verify the performance of the framework by modeling a real world scenario using task modeling methodology and comparing the simulation statistics with that occurring in the real world.
Appendix A

Actual Simulation Results of Experiments

The appendix presents the actual simulation results of the experiments. These results are presented in the form of screenshots taken from SCExplorer tool, which was used to plot the results.
Figure A.1: Results of VPU1 in experiment 1
Figure A.2: Results of VPU2 in experiment 1
Figure A.3: Results of VPU1 in experiment2
Figure A.4: Results of VPU2 in experiment 2
Figure A.5: Results of VPU1 in experiment 3
Figure A.6: Results of VPU2 in experiment3
Figure A.7: Results of VPU1 in experiment 4
Figure A.8: Results of VPU2 in experiment4
Figure A.9: Results of VPU1 in experiment 5
Figure A.10: Results of VPU2 in experiment 5
Figure A.11: Results of VPU1 in experiment 6
Figure A.12: Results of VPU2 in experiment 6
Figure A.13: Results of VPU1 in experiment 7
APPENDIX A. ACTUAL SIMULATION RESULTS OF EXPERIMENTS

Figure A.14: Results of VPU2 in experiment7
Figure A.15: Results of VPU1 in experiment 8
APPENDIX A. ACTUAL SIMULATION RESULTS OF EXPERIMENTS

Figure A.16: Results of VPU2 in experiment8
APPENDIX A. ACTUAL SIMULATION RESULTS OF EXPERIMENTS

Figure A.17: Results of VPU1 in experiment9
Figure A.18: Results of VPU2 and VPU3 in experiments 9 and 10
Figure A.19: Results of VPU1 in experiment 10
Bibliography


