Title
Adaptive Real-Time Systems

Permalink
https://escholarship.org/uc/item/2zv398g6

Author
Springer, Tom John

Publication Date
2015

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,
IRVINE

Adaptive Real-Time Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Tom Springer

Dissertation Committee:
Professor Tony Givargis, Chair
Professor Nikil Dutt
Professor Eli Bozorgzadeh
Dr. Steffen Peter
DEDICATION

To

my wife for her love in encouraging me to follow my dreams, for her support in staying up with me on all those late nights and for her patience and understanding.
# Table of Contents

List of Figures v
List of Tables vii
Acknowledgements viii
Curriculum Vitae ix
Abstract of the Dissertation x

1. Introduction 1
   1.1 Research Motivation 1
   1.2 Research Objectives 3
   1.3 Research Contributions 3
   1.4 Thesis Structure 5

2. Recent Research Trends in Real-Time Scheduling for Embedded Systems 9
   2.1 Introduction 9
   2.2 Evolving Application Requirements 11
   2.3 Limitations with the Existing Approach 13
   2.4 Challenges and Future Work 14

3. Hierarchical Scheduling Framework 20
   3.1 Hierarchical Scheduling 20
   3.2 Server Schedulability Analysis 22
   3.3 Server Bandwidth Allocation 27

4. Resource Synchronization In Hierarchical Scheduled Systems 29
   4.1 Introduction to Resource Synchronization in Hierarchical Scheduled Systems 29
   4.2 Related Work in Resource Synchronization for Hierarchically Scheduled Systems 36
   4.3 Resource Synchronization in HSF using Preemption 37
   4.4 Adaptive Resource Synchronization in HSF 47

5. Hierarchical Scheduling for Periodic Tasks in Symmetric Multiprocessing 65
   5.1 Introduction to Real-Time Task Scheduling in Hierarchical Scheduled SMP-Based Systems 65
   5.2 Related Work in Hierarchical Scheduling for SMP-Based Environments 71
   5.3 HSP Algorithm Description 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Schedulability Analysis</td>
<td>81</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance Analysis</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>Fuzzy Logic Based Adaptive Hierarchical Scheduling For Periodic Real-Time Tasks</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction to Fuzzy Based Adaptive Hierarchical Scheduling for Real-Time Tasks</td>
<td>91</td>
</tr>
<tr>
<td>6.2</td>
<td>Related Work in Adaptive Hierarchical Scheduled Systems</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>AHS-RT Architecture</td>
<td>98</td>
</tr>
<tr>
<td>6.4</td>
<td>Task Scheduling Example</td>
<td>104</td>
</tr>
<tr>
<td>6.5</td>
<td>Subsystem Reallocation Example</td>
<td>106</td>
</tr>
<tr>
<td>6.6</td>
<td>Simulation</td>
<td>107</td>
</tr>
<tr>
<td>7</td>
<td>Applied Application for Resource Sharing in Hierarchical Scheduled Systems</td>
<td>110</td>
</tr>
<tr>
<td>7.1</td>
<td>Hardware/Software Components</td>
<td>111</td>
</tr>
<tr>
<td>7.2</td>
<td>RACPwP Application Results</td>
<td>112</td>
</tr>
<tr>
<td>7.3</td>
<td>ARAP Application Results</td>
<td>113</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>117</td>
</tr>
<tr>
<td>9</td>
<td>References</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>Appendix</td>
<td>124</td>
</tr>
<tr>
<td>10.1</td>
<td>VxWorks Implementation</td>
<td>124</td>
</tr>
<tr>
<td>10.2</td>
<td>RTSIM Modifications</td>
<td>129</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1-1: Structure of the Thesis .................................................. 6
Figure 3-1: Hierarchical Scheduling Framework ............................. 21
Figure 3-2 Periodic Server Example ................................................ 21
Figure 3-3: Worst-case scenario for n periodic tasks and a periodic server with the highest priority ................................................. 23
Figure 3-4: Schedulability bound for periodic tasks and a periodic server as a function of the server utilization factor ......................... 26
Figure 4-1: Task execution time with shared resources .................. 30
Figure 4-2: Stack Resource Policy Example .................................... 32
Figure 4-3: Server budget exhaustion inside a critical section .......... 34
Figure 4-4: Server budget inside a critical section with overrun (no payback) ................................................................. 35
Figure 4-5: Example Resource Access Control Protocol with Preemption ............................................................ 40
Figure 4-6: Traditional Locking Mechanism ..................................... 40
Figure 4-7: Preemptable Critical Section .......................................... 40
Figure 4-8: ARAP architecture flowchart .......................................... 48
Figure 4-9: Feedback architecture .................................................... 50
Figure 4-10: Critical section execution times - projected vs. actual .... 51
Figure 4-11: PID critical execution time estimation error ................. 52
Figure 4-12: Supply bound function virtual processor model ............ 54
Figure 4-13: Average resource utilization ........................................ 57
Figure 4-14: Task acceptance rate ................................................... 58
Figure 4-15: ARAP architecture in RTSIM with sample task set ......... 59
Figure 4-16: Instantaneous load as a function of time ..................... 62
Figure 4-17: Hard real-time task miss rate ....................................... 63
Figure 4-18: Soft real-time task miss rate ......................................... 63
Figure 5-1: Supply bound and lower supply function for a subsystem .... 70
Figure 5-2: Hierarchical Scheduling for multi-core processor .............. 74
Figure 5-3: Partitioned task core assignments ................................... 76
Figure 5-4: Split task across two processors ..................................... 77
Figure 5-5: Core 1 task schedule ...................................................... 80
Figure 5-6: Core 2 task schedule ...................................................... 80
Figure 5-7: Task set schedule simulation (2 cores) ......................... 86
Figure 5-8: Task set schedule simulation (4 cores) ......................... 86
Figure 5-9: Task set simulation (8 cores) ......................................... 87
Figure 5-10: HRT Task Set Response Time Average ...................... 89
Figure 5-11: SRT Task Set Response Time Average ...................... 89
Figure 6-1: AHS-RT Architecture .................................................... 94
Figure 6-2: Internal structure of feedback controller 103
Figure 6-3: Membership function for linguistic variables 104
Figure 6-4: Number of Deadline Misses (All Tasks) 108
Figure 6-5: Number of Deadline Misses ( Highest Criticality Tasks) 109
Figure 7-1: Telemetry Uplink System for Hardware-in-the-Loop Simulations 110
Figure 7-2: HWIL Avionics Subsystem Simulation 111
Figure 7-3: Hard Real-Time Response Times 113
Figure 7-4: Soft Real-Time Response Times 113
Figure 7-5: Hard Real-Time task miss rate 114
Figure 7-6: Soft real-time task miss rate 115
Figure 10-1: HSF implementation in VxWorks (single processor) 127
Figure 10-2: HSF implementation in VxWorks ( multi-core) 128
Figure 10-3: RTSIM AbsTask Class Architectural Diagram 129
Figure 10-4: RTSIM ResManager Class Architecture 137
LIST OF TABLES

Table 4-1: Server Parameters ........................................................................................................... 45
Table 4-2: Server average worst-case response times .................................................................. 45
Table 4-3: Task parameters ............................................................................................................. 46
Table 4-4: Task worst case response times ....................................................................................... 46
Table 4-5: PID tuning parameters .................................................................................................... 60
Table 4-6: Example Task Set ............................................................................................................ 61
Table 5-1: Example subsystem task set ............................................................................................ 75
Table 5-2: Example task set with response times ............................................................................ 83
Table 5-3: Simulated Periodic Task Set ............................................................................................ 88
Table 6-1: Rules base for fuzzy controller ....................................................................................... 104
Table 6-2: Subsystem parameters .................................................................................................... 104
Table 6-3: Task Parameters ............................................................................................................. 105
Table 6-4: Scheduling snapshot at time 0 ....................................................................................... 105
Table 6-5: Scheduling snapshot at time 10 ..................................................................................... 105
Table 6-6: Scheduling snapshot at time 20 ...................................................................................... 105
Table 6-7: Subsystem Parameters ................................................................................................... 106
Table 6-8: Subsytem Reallocation snapshot .................................................................................... 106
ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my committee chair and advisor, Professor Tony Givargis, for his insight and direction of research topics that were both challenging and rewarding. Without his guidance, patience and persistent help this dissertation would not have been possible. I would also to express my gratitude to Dr. Steffen Peter for all his help in providing reviews of this research and making valuable suggestions on how to improve its quality.

Additionally, I would like to thank my committee members, Professor Nikil Dutt and Professor Eli Bozorgzadeh for their insights and reviews of this thesis.

I also would like to thank the Boeing Company that supported me while I performed my doctoral research and provided valuable real-world problems that could be utilized as use cases to verify some of the research topics.

This work was supported in part by the National Science Foundation under NSF grant number 1136146.
CURRICULUM VITAE

Tom Springer

1989 B.S. in Computer Science, California State University, Fullerton
1994 M.S. in Computer Science, California State University, Fullerton
1995-2015 Adjunct Faculty, Lecturer, Computer Science, California State University, San Marcos.
2015 Ph.D. in Information and Computer Science,
University of California, Irvine

FIELD OF STUDY

Adaptive Scheduling for Real-Time Embedded Systems

PRESENTATIONS

Software Simulation Conference, Anaheim, CA 2003
*SWIL/HWIL Simulations using Discrete and Continuous Model Simulators.*
Data Acquisition Conference, Canoga Park, CA 2005
*Data Acquisition using a System Simulation Infrastructure*
Boeing Software Conference, Brea CA 2008
*Service Oriented Architecture Approach to Designing and Developing Embedded Systems.*

PUBLICATIONS


Modern embedded systems are required to work in ever increasing dynamic environments, where predicting the computational load on those systems is intractable. However, timely responses to events have to be provided within precise timing constraints in order to guarantee a required level of performance. Consequently, embedded systems by their very nature exhibit real-time characteristics which impose an additional set of restrictions than those in a typical general purpose system. In addition to the limitations of having to perform to strict timing constraints, most embedded systems are constrained by size, weight, energy consumption and cost limitations. As a result, efficient resource management is a critical aspect in embedded systems that must be considered at multiple architectural levels.

The main objective of this work is to present our work on real-time systems that progress to make the next generation embedded systems more predictable and adaptive to dynamic computational changes. To achieve these goals, this phase of our research has focused on the resource synchronization and adaptive scheduling of real-time embedded applications in uniprocessor and multi-core environments. The analysis and experiments show that our resource
synchronization protocols outperformed other state-of-the art resource access control protocols used in hierarchical scheduled systems. Implemented in VxWorks and applied to applications used in the aerospace industry response times for hard real-time tasks were improved and deadline misses for hard real-time tasks were substantially reduced.
1. INTRODUCTION

This chapter provides the overall scope and context of the research and the applications applied in this thesis. It discusses the primary motivation for this research as well as the overall theoretical and applied contributions.

1.1 Research Motivation

Next generation real-time embedded systems are evolving from computational stand-alone components to more of a network of interacting elements between both computational and physical entities. These new systems, known as cyber-physical systems (CPS), comprise a collection of computational components that control or monitor the physical components, such as sensors. Therefore, to realize the usability of CPS further research is needed to improve the adaptability and reliability of current embedded systems.

In order to increase the adaptability and reliability various resources will need to be allocated at run-time rather than at design time. Resources such as processor bandwidth or memory utilization could benefit from a dynamic resource allocation strategy that bases the allocation on the operating mode or the environmental changes in the system at run-time. There are a number of scenarios where a real-time embedded system could benefit from a more dynamic approach to resource allocation. Consider an autonomous unmanned vehicle (AUV) where the operating modes are defined as fully autonomous, semi-autonomous or remote control. Fully autonomous is the operational mode where no human intervention is required. Semi-autonomous mode requires some level or operator intervention. Remote control mode requires almost continuous input from the operator. In legacy AUV systems, which are designed
according to static resource allocation techniques the operator is required to switch between operating modes to support these functions. The fundamental design strategy of these legacy systems is to optimize resource allocation according to operating mode. The problem with this approach is that it requires the allocation decisions to be embedded within the application components, instead of the overall system which hinders adaptability. The result is extensive integration and testing is required whenever the system capabilities or requirements change.

Dynamic resource allocation has the potential to provide improved system adaptability and reliability. For instance, consider a mode change from remote to fully autonomous mode where sensory process workloads could increase considerably. The use of dynamic allocation could be used to adjust the computational resources to manage the increased processing required to service the sensory data. In this way, a quality of service could be provided for the more critical tasks while still maintaining a level of service for less critical tasks.

Future embedded systems, such as AUVs are just one example of the general evolution of embedded systems from a statically configured, tightly-coupled system to a more open source component based environment operating in a less deterministic situation. In general, cyber-physical systems such as those within aerospace, transportation, energy, entertainment healthcare and manufacturing are becoming increasingly interconnected with other real-time and even non-real-time systems. Due to the desire to react to this varying workloads real-time embedded system design is evolving to be more adaptive during run-time that during design time. This evolution motivates our research into adaptive real-time embedded software systems that dynamically allocate resources to provide a level of service based upon environmental changes.
1.2 Research Objectives

The objective of our research is to develop an adaptive software scheduling framework focused on the following aspects of adaptability. First, provide for a resource based scheduling approach that manages operations under overload conditions. Secondly, achieve higher resolution of existing computations by varying CPU availability that results from computational workload changes. Lastly, increase the ability to handle mixed criticality tasks with deterministic and non-deterministic execution times.

Mixed criticality real-time systems must perform in harsh environments where they are expected to perform a sub-set of their critical functions under fault conditions. The occurrence of faults is classified as shared resources, such as processors, being overloaded and therefore some tasks will miss their deadlines. The primary goal of the work in this thesis is to develop an approach that manages tasks of real-time systems specifically under overload conditions where the fault of one task(s) does not cause a catastrophic collapse of the system as a whole. That is during nominal conditions the system would perform as if there were no overload management while the adaptive capability would take effect in the presence of an overload condition. This adaptive approach would then be able to react to the occurrence of a fault or overload event and then allocate resources to tasks based upon some Quality of Service (QoS) parameters. The use of these QoS parameters is what allows the tasks to be scheduled based upon their criticality or importance.

1.3 Research Contributions

Adaptive real-time scheduling is one path toward the roadmap to adaptive real-time systems. The primary mechanism used to provide adaptive real-time scheduling is via hierarchical scheduling. For this reason this work focuses on improving the current state-of-the-
art in hierarchical scheduling for uniprocessor and multi-core systems. Therefore the research contributions by this work are provided here:

- A new resource synchronization protocol for semi-independent hierarchical scheduled systems using preemption. The protocol analysis illustrated that this approach provided improved response times for high priority tasks as compared to other state-of-the-art synchronization protocols developed for hierarchically scheduled systems. Implemented in VxWorks and applied to an aerospace application the response time improvements for high-priority tasks were demonstrated as compared to traditional synchronization protocols.

- A new resource synchronization protocol for semi-independent hierarchical scheduled systems using prediction. Results of the protocol were verified using an open source real-time scheduler (RTSIM) simulation developed by the Real-Time Systems Lab (ReTiS) at St. Anna's School of Advanced Studies, Pisa, Italy. Implemented in VxWorks and applied to a hardware-in-the-loop vehicle simulation the deadline misses were substantially reduced or eliminated as compared to traditional synchronization protocols.

- The modifications made to the RTSIM simulation to support the new resource synchronization protocols have been requested by the ReTiS lab so it can be folded back into the user community.

- A new hierarchical real-time scheduling framework for mixed-criticality systems applied to symmetric multiprocessing (SMP) platforms. Provides a framework for a course grain mapping of tasks to a specific processor or cluster of processors. In this way tasks can be grouped together based upon their application requirements or
constraints. Leverages the benefits of both the global and partitioned scheduling schemes supported by SMP-based schedulers to provide determinism for hard real-time tasks and improved response times for soft real-time tasks. Implemented in VxWorks the results were demonstrated improvement of response times for soft real-time tasks and schedulability guarantees for hard real-time tasks where no deadlines were missed during periods of overload.

- The first known approach to dynamically adapt the parameters of a hierarchical scheduled system. The result is a novel fuzzy-logic based adaptive real-time hierarchical scheduler where hierarchically scheduled applications can dynamically adapt to changing computation environments.

- Research work was applied and demonstrated effective in actual industry based embedded applications.

1.4 Thesis Structure

The structure of this thesis contains nine chapters. The first chapter is an Introduction to the motivation, research objectives and contributions of this work. Chapter 2 identifies the current research trends in real-time scheduling for embedded systems. Chapter 3 provides an overview of hierarchical scheduling which provides the basic framework for this work. Preemptive and adaptive research sharing for semi-independent systems in hierarchical scheduled systems is presented in Chapter 4. Hierarchical scheduling for multi-core symmetric based processors is presented in Chapter 5 while adaptive hierarchical scheduling for uniprocessors and multi-core platforms are presented in Chapter 6.
Chapter 2 – Recent Research in Real-Time Computing for Real-Time Embedded Systems

identifies the major research trends in real-time embedded computing. Embedded systems are evolving from the single-purpose highly specialized processing environments to more general purpose open-source computing platforms. Traditional embedded processing models do not provide an ideal mapping to these next generation systems. This chapter presents the problems of the current approaches and discusses the new research trends in operating systems and scheduling emerging to overcome them.
Chapter 3 – Hierarchical Scheduling Framework provides an overview of hierarchical scheduling in uniprocessor based platforms. This chapter presents the overall framework as well as the server mechanism which is the primary method for enforcing temporal isolation in hierarchical scheduled systems. The schedulability analysis of the server mechanism is provided as well as the bandwidth allocation required for system schedulability.

Chapter 4 – Resource Sharing in Hierarchical Scheduled System presents how resources are shared in semi-independent systems, specifically mutually exclusive resources (i.e. semaphores). Traditionally, the subsystems were independent and not considered to share resources across subsystems, similar to how virtual machines operate. However most practical systems are not entirely independent and may need to share resources across the subsystems. This chapter presents some of the protocols developed by researchers to share resources across subsystems as well as the limitations with the current approaches, namely unbounded holding times. Some of the contributions of this thesis are provided in this chapter as solutions to the limitations of current state-of-the-art protocols. This work specifically examines two approaches to resolving unbounded holding times; preemptive and predictive. The preemptive approach simply preempts a task then rolls back the transaction if the task holds the resource for too long. The predictive approach uses feedback control to predict how long a particular task will hold the task then determines based upon that holding whether the task should be granted access to the resource.

Chapter 5 – Hierarchical Scheduling for Periodic Tasks in Symmetric Multiprocessing provides a description to how hierarchical scheduling can be applied to a symmetric multiprocessor based system. This chapter presents other contributions of this thesis such
as how tasks are scheduled and mapped to multi-core platforms based upon the task
criticality factor. Hierarchical scheduling is used to group applications based upon their
functionality for improved resource aggregation and temporal isolation between
applications. The task criticality factor is used to map the tasks to the cores in the
subsystem cluster. High-criticality tasks are statically mapped to specific cores for
determinism guarantees while lower criticality tasks are allowed to migrate across cores
in the cluster to improve performance.

Chapter 6 – Fuzzy Logic Based Adaptive Hierarchical Scheduling for Periodic Real-Time
Tasks presents an approach where a fuzzy logic based heuristic method is used to
provide system adaptability. This chapter presents contributions of this thesis to
schedule adaptability of hierarchical scheduling where the scheduling of subsystems
adjusts based upon a tasks level of criticality instead of just a strictly deadline based
approach. The result is an embedded system that is better equipped to adapt to
computational variations providing timing guarantees for high criticality tasks while
providing a minimum level of service to lower criticality tasks.

Chapter 7 – Practical Application for Resource Sharing in Hierarchical Scheduled Systems
presents an actual aerospace based hardware in the loop simulation. The preemptive and
predictive based resource synchronization protocols were used in the simulation and
proven to be effective in improving response times and eliminating deadline misses for
hard real-time tasks as compared to traditional resource synchronization protocols.
This chapter presents some of the foremost research trends in real-time scheduling for embedded systems. The main objective is to improve the link between the cyber and physical elements of next generation embedded systems by enhancing their reliability and predictability. The characteristics and behavior of current and future real-time embedded applications are discussed as well as the challenges and new research trends that are emerging to address these challenges.

2.1 Introduction

The use of embedded devices in our daily lives has increased dramatically with the global market exceeding over 1 trillion dollars in 2014.\(^1\) Processors are embedded into many of the devices we use every day, such as cell phones, tablets, televisions, household appliances, automobiles and just about everything else you can plug in.

Considering the proliferation of next generation embedded devices into cyber-physical systems such as the smart energy grid the trend is expected to continue well into the future. While next generation embedded devices do continue to evolve they still possess many of the same properties as identified by authors in [57].

- **Limited resources.** Many embedded systems are designed with resource constraints, such as size, weight and power (SWaP) requirements. As a result, many embedded systems have limited processing power and memory capacity as compared to general-

---

\(^1\) New Venture Research http://www.newventureresearch.com
purpose systems. Due to these types of restrictions it is even more imperative that the scare resources that are available are managed as efficiently as possible.

- **Real-time constraints.** Embedded systems typically interact with the physical environment typically in the form of sensing devices. The processing in these systems has strict timing constraints to be able to react to changes in the physical environment. The operating system is responsible for enforcing these timing constraints and ensuring the predictable behavior of the process.

- **Dynamic behavior.** Next-generation embedded systems are evolving from single-threaded processing to much more complex multi-threaded applications where multiple processes interact and compete for shared resources. To further increase this complexity applications are more closely linked to the physical environment which makes their processing behavior much harder to predict in advance. Additionally, modern hardware optimizations meant to improve overall performance, such as caching, pipelining, pre-fetching and DMA, can introduce non-deterministic behavior making worst-case execution time estimation very unpredictable. As a result, the combination of variations introduced by hardware optimizations and varying processing workload makes the static predicting of system behavior much more difficult.

Because the next generation embedded device are being designed more like general-purpose platforms coupled with the fact that they still possess the same properties as legacy systems presents additional challenges at multiple architectural levels. In the past the typical approach has been to design systems based upon the worst-case execution times but in new
highly dynamic environments this method would waste the limited resources and increase overall cost.

A more intelligent approach would be for the system to be able to react to the varying physical environment instead of just statically designing for the worst case. However, this requires that the system must now be adaptive in order to adjust to a dynamic environment and change internal responses to keep the system performance at the desired level or to at least degrade the performance to an acceptable level. In order to provide this type of adaptability changes are required at multiple levels of the software stack. Modifications would be needed at the operating system level, specifically in the kernel, at the middleware layer and quite possibly the application layer. However, because the kernel is closest to the hardware it would be the most efficient to implement adaption at this layer. Additionally, by implementing the details of adaption at the layer nearest the resources the layers of abstraction paradigm is maintained at the higher layers like the application layer.

2.2 Evolving Application Requirements

Embedded applications are evolving from the special-purpose single-threaded tasks to multi-purpose multi-threaded applications with complex interactions. In effect when it comes to workload processing embedded systems are becoming more like general-purpose applications except they are expected to perform more reliably, with limited resources and on time. One of the most common examples is the smart phone with a large number of applications and multiple concurrent processes. Therefore, it is critical that software optimizes the available limited hardware resources to improve efficiency and maintain reliability.

In the consumer electronics market authors in [57] categorize the main software activities with regard to scheduling and resource sharing.
• *Control software* which is typically implemented as periodic tasks and utilizes a small portion of the overall resources. Examples include command and control tasks that are subject to hard timing constraints and must be guaranteed offline in all operating conditions.

• *Media processing software* is typically data driven and a primary consumer of available resources (processor or memory capacity). Multimedia based applications are main examples of this type of software. These applications due to their large resource requirements and soft-real time constraints are typically offered at a level-of-service. For example, if the current available resources are not sufficient to playback a high-definition video, the quality-of-service (QoS) requirements may state that the video can then be processed at a lower resolution. In this way, instead of not being able to playback the video at the requested level of service at least some level of service is provided.

• *Interactive software* describes applications that respond to user or operation interaction. These applications are represented as *sporadic* tasks therefore these tasks timing requirements are more response time based rather than deadline based. Examples include video games, internet or photo/music browsing or on-demand programming guides for a DVD player.

The challenge is scheduling such diverse activities is to effectively manage the available resources such that each application is at least provided a minimum level of service. Additionally, it is difficult to scale theses systems because simply adding or deleting features may make the system unstable or even fail. As a result, anytime a new feature is added or an old feature removed exhaustive redesign and testing is required. It is not practical to design systems
for every possible use case. In order to eliminate excessive redesign new tools are needed to represent and manage resource requirements in dynamic systems.

In traditional legacy embedded systems the solution to these problems has been to distribute diverse applications across multiple CPUs. One CPU may be assigned the control tasks; another CPU is assigned the media processing tasks while another is assigned the interactive tasks. The idea is by physically separating the various software functionalities resource allocation and scheduling is simplified while protection is provided such that the behavior of a media processing application does not affect the behavior of a control application. The problem with these distributed applications is that they require specialized synchronization and communication mechanisms consume more power and expand their physical footprint.

Another alternative is to provide a single CPU system (with possibly multi-cores) that supports memory protection and temporal isolation where various hard real-time and soft real-time tasks could be safely combined. The result would be a much more scalable system where functional verification could be simplified by eliminating the need for extensive redesign and re-test.

2.3 Limitations with the Existing Approach

Task priority is the primary mechanism for identifying importance or criticality in real-time systems. There are a number of reasons why a single parameter is inadequate when configuring complex dynamic embedded systems. One reason is dynamic system constraints cannot be mapped to a set of priority levels.

Another problem with using priority to represent importance is the same applications could play a different role depending upon the scenario. For example, consider the AUV system
mentioned in the previous chapter. Guidance, navigational and control (GNC) tasks play significantly different roles depending on whether the vehicle is in fully autonomous or semi-autonomous. In this scenario it is much harder to group applications by their mode or functionality because the global property of system priorities would violate this grouping.

The current approach used to provide some type of adaptability in dynamic systems is to modify the task priorities at run-time. The problem is that it is difficult to predict how the system will perform at run-time when task priorities are modified dynamically. For example, increasing the priority of a task with a long execution could result in starvation of any tasks with a lower priority. Conversely, decreasing the priority of a high priority task could adversely affect the behavior of lower priority tasks by implicitly raising the priority of lower priority tasks resulting in possible overload conditions. Tasks that share resources (i.e. semaphores) present additional challenges in that the task priority can also affect the overall blocking or resource holding time. Consider a situation where a high priority task and low priority task share a resource and the resource was defined using the priority inheritance protocol in order to avoid priority inversion. Suppose the priority of the tasks sharing the resource is changed at the wrong time which could lead to serious side effects such as priority inversion [37]. These examples illustrate why traditional scheduling mechanisms are not well-suited for dynamic adaptive systems because the operating systems do not provide explicit support for quality-of-service (QoS) management.

2.4 Challenges and Future Work

In order to support dynamic task behavior the system must be able to adapt to these changes and as stated above recent research has shown that effective QoS support must be provided. Authors in [57] identified the three basic properties that operating systems need to provide in order to provide this type of adaptive support.
• Reflective. The application characteristics should be represented as a set of parameters that the scheduler could use to tune performance. Example parameters could include task deadlines, task periods, task criticality or task execution time.

• Resource aware. The application’s computational requirements should be represented so that resources could be effectively partitioned among the existing applications. The idea is that this partitioning would provide a form of temporal protection and prevent interference from other applications during overload conditions.

• Informative. Information on the current state of execution should be provided to allow for the implementation of adaptive management schemes. The difference between the actual and expected behavior of the application could then be used to adjust the system parameters to achieve better control of the overall system performance.

2.4.1 Resource Reservations

The resource reservation approach is a programming model that enables the designer to explicitly reserve the resources assigned to a specific activity at any given time. The idea is that an application or task only receives a portion of the overall system resources [37].

Presently, reservation-based scheduling if focused on the processor only but an overall system wide approach is needed that includes other resources such as main memory, disk storage or network bandwidth. The major challenge is with the constraints of embedded systems the overhead incurred by managing these reservations need to be contained.

2.4.2 Efficient Resource Reclaiming

While the reservation approach to allocating resources provides the designer more controller over how resources are allocated it is dependent upon how much of the resource is allocated to an application. If the amount of resource is under allocated then the application or
task may slow down too much affecting the nominal performance. On the other hand, if the amount of resource is over allocated then resources could be wasted which would affect the overall performance of the system.

The challenge is providing an optimal resource reservation for each application is non-trivial and requires extensive testing and code coverage analysis. The solution is to provide a reasonable resource allocation estimate then let the kernel adjust the reservations to account for wrong or imprecise reservations. The idea is similar to memory allocation in that any resource that is under-utilized gives back the unused portion to be allocated by other applications that may require more bandwidth.

It is worth observing, however, that resource reclaiming is an on-line mechanism that exploits early completions, hence it cannot always compensate for reservation errors. In the average, resource reclaiming is quite effective for compensating for small reservation errors, but it does not represent the solution for coping with large and systematic deviations in the execution behavior of computational activities.

2.4.3 Providing Real-Time Control

If an application has diverse workload requirements or varying execution behavior then feedback control theory could be used to estimate the current workload and then tune the task or reservation parameters accordingly [38]. The integration of real-time scheduling and control theory is a relatively new approach that has provided some promising research. The benefits are that feedback control can be used by the kernel to adjust the application parameters such that the resource reservations of an application could be made more adaptive to unpredictable changes.
2.4.4 Hierarchical Scheduling

Hierarchical scheduling is a framework that leverages the performance gains in modern computers to partition multiple applications. This approach is to partition the processor into a group of virtual machines where each application (subsystem) is allocated a portion of the processor. The motivation is that a single scheduling scheme may not be an ideal mapping for a diverse set of applications so different scheduling algorithms may need to be used by the same processor. However, multiple scheduling algorithms makes schedulability analysis more complicated and additional theoretical work may be required to provide schedulability guarantees. The hierarchical scheduling framework means that there is just one scheduling algorithm for each subsystem. The subsystems or applications are grouped according to their criticality or functionality and are represented as the leaves. The overall system is represented as the root of the hierarchy. Therefore, each subsystem can have its own scheduler so the system consists of a hierarchy of schedulers. In this way the root scheduler can provide guaranteed resource reservation, temporal isolation and ease of the complexity of schedulability analysis because each subsystem can be viewed as running on its own isolated processor.

2.4.5 Overload Management

In order to provide predictability in dynamic systems the incoming workload must be controlled to prevent overload conditions. When the workload exceeds the available processor bandwidth the system can quickly become unstable and cause severe performance degradation.

Computational Overload can be managed using various techniques.

- QoS levels. Some software can be performed by using different algorithms with different execution environments. For example, in some instances an application may not require precise computations so integer arithmetic may be used instead of floating
point operations. In other instances, some computations may be performed using less compute intensive algorithms resulting in varying levels of quality. In other words, the workload can be controlled by using different QoS levels.

- Adjustable timing constraints. In a real-time system the workload is controlled not only by the execution time but also by the timing constraints of a task. Therefore, the workload could be eased by relaxing the timing constraints of task. In the event of an overload condition the deadline or period of a task can be reduced or enlarged [40].

- Admission control. Another way to control the workload during an overload condition is to control the amount of tasks that are actually allowed to execute. This method alleviates an overload condition by rejecting a task or tasks that are scheduled to execute.

2.4.6 Recent Trends

In the last 30 years, embedded systems experienced an exponential growth in many application domains, both in terms of number and complexity. Surprisingly, however, such a growth in complexity was not followed by a corresponding evolution of the control software used to manage the computational resources, which is substantially similar to that adopted in the early 70s. In fact, application activities are still handled by cyclic executives or, in the best case, by fixed priority kernels. The problem is not due to a lack of alternatives, but more that nobody has been able to make a convincing case for a transition. Every attempt to raise the level of abstraction has included unacceptable penalties in terms of memory and speed. Also from an industrial perspective, the support for legacy code has often been weak.

The possibilities for embedded systems to evolve and become more reliable, while yet more complex, to some extent depend on what the next generation real-time operating systems
and implementation tools have to offer. The challenge is how to implement applications that can execute efficiently on limited resources, to meet non-functional requirements, such as timeliness.

Proper resource and quality-of-service management would enable the implementation of embedded systems that are more flexible, yet more deterministic, than it is possible today. Since such systems would be better specified, their properties would also be verified more easily. By supporting explicit resource allocation and quality-of-service functionality, the system designers would regain control over the system they are set to design.

To effectively assign system resources among applications and achieve predictability and flexibility, issues should be further investigated. At the higher abstraction level, protocols for managing quality levels and suitable architectures should be used to obtain flexible systems. At a lower level, further work on resource management algorithms, new task models, admission control, monitoring, and adaptation algorithms should be done.
3 HIERARCHICAL SCHEDULING FRAMEWORK

In this chapter we provide an overview of hierarchical scheduling for uniprocessor based platforms. The overall framework as well as the server mechanism which is the primary method for enforcing temporal isolation in hierarchical scheduled systems is presented. The schedulability analysis of the server mechanism is provided as well as the bandwidth allocation required for system schedulability. Hierarchical scheduling is presented here because the framework provides the primary building block used in this research for supporting adaptability in dynamic systems.

3.1 Hierarchical Scheduling

Hierarchical scheduling is a scheduling technique that is used to provide resource partitioning for related groups of tasks for the primary purpose of providing temporal isolation [2] between those tasks. Because of this temporal protection HSF based systems have proved to be particularly useful in the area of open systems [1] where applications can be developed, integrated and validated independently. A primary goal of hierarchical scheduling is to bind the temporal behavior of those applications whose execution times deviate considerably, allowing for the predictable operation of the various subsystems.

The basic framework of a hierarchically scheduled system [1] [3] is composed of multiple applications (subsystems) where each application could be composed of multiple tasks (see Figure 3-1). A global scheduler controls which application can use the processor while the local scheduler determines which application’s task should actually execute.
Every application is allocated a separate service manager, known as the server. Each server is allocated a CPU capacity reserve, which is assigned as a pair \((Q_i, P_i)\) where \(Q_i\) is defined as the time quantum and \(P_i\) is defined as the period. Each task gets to execute for its assigned time quantum \(Q_i\), when the task’s time quantum \(Q_i\) is exhausted the task is blocked until its next period (see Figure 3-2). In effect, the server functions as an independent processor virtually limiting the bandwidth of each application.
Each server has its own priority which is used by the global scheduler to determine which server is allocated to the processor. The local scheduler, as part of each server, determines which task should execute when the server is re-activated. In this way, temporal protection is enforced by assigning only a fraction of the resource, ensuring that each task can only consume its maximum allotment.

3.2 Server Schedulability Analysis

The schedulability of periodic tasks can be guaranteed by evaluating the interference introduced by a periodic server on periodic execution. In the worst case, such as interference is the same as the one introduced by an equivalent periodic task having a period equal to $P_s$ and a computation time equal to $Q_s$. In fact, independently of the number of tasks handled by the server, a maximum time equal to $Q_s$ is dedicated to the number of tasks during each server period. As a result, the processor utilization factor of the periodic server is $U_s = \frac{Q_s}{P_s}$ where the schedulability of a periodic task set with $n$ tasks and utilization $U_p$ can be guaranteed if

$$U_p + U_s \leq U_{lub}(n + 1).$$

If the periodic tasks (including the server) are scheduled by RM, then the schedulability test becomes:

$$\sum_{i=1}^{n} \frac{c_i}{p_i} + \frac{Q_s}{P_s} \leq (n + 1) \left[ 2^{\frac{1}{n+2}} - 1 \right].$$

Note that multiple periodic servers could be created and executed on different task sets. For example, a high-priority server could be reserved for a subset of hard real-time tasks where a lower-priority server could be used to manage soft real-time task sets. In general, in the presence of $m$ servers, a set of $n$ periodic tasks is schedulable by RM if
A more precise test can be derived by using the same technique adopted for the Liu and Layland schedulability bound by assuming that the periodic server is the highest-priority task in the system. To simplify the computation, the worst-case relations among the tasks are first determined and then the lower bound is computed against the worst-case model.

Consider a set of $n$ periodic tasks, $\tau_1, ..., \tau_n$, ordered by increasing periods and a periodic server with a highest priority. The worst-case scenario for a set of periodic tasks that fully utilize the processor is the one illustrated in Figure 3-3, where tasks are characterized by the following parameters:

$$
\begin{align*}
Q_s &= P_1 - P_s \\
C_1 &= P_2 - P_1 \\
C_2 &= P_3 - P_2 \\
C_{n-1} &= i_n - T_{n-1} \\
C_n &= P_s - Q_s - \\
\sum_{i=1}^{n-1} C_i &= 2P_s - P_n
\end{align*}
$$

Figure 3-3: Worst-case scenario for $n$ periodic tasks and a periodic server with the highest priority

$$
U_p + \sum_{j=1}^{m} U_{s_j} \leq U_{\text{ub}} (n + m).
$$
The resulting utilization is then

\[ U = \frac{Q_s}{P_s} + \frac{C_1}{P_1} + \ldots + \frac{C_n}{T_n} = \]

\[ = U_s + \frac{P_2-P_1}{P_1} + \ldots + \frac{P_n-P_{n-1}}{P_{n-1}} + \frac{2P_s-P_n}{P_n} = \]

\[ = U_s + \frac{p_2}{p_1} + \ldots + \frac{p_n}{T_{n-1}} + \left( \frac{2P_s}{P_{n-1}} \right) \frac{p_1}{p_{n-1}}. \]

Defining

\[
\begin{align*}
R_s &= \frac{P_1}{P_s} \\
R_i &= \frac{T_{i+1}}{T_i} \\
K &= \frac{2P_s}{P_1} = 2/R_s
\end{align*}
\]

And noting that

\[ R_1R_2 \ldots R_{n-1} = \frac{R_n}{R_1}, \]

Then the utilization factor may be written as

\[ U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{R_1R_2 \ldots R_{n-1}} - n. \]

Following the approach used for RM and minimize $U$ over $R_i$, $i = 1, \ldots, n - 1$.

\[
\frac{\delta U}{\delta R_i} = 1 - \frac{K}{R_i^2 \left( \prod_{j \neq i} R_j \right)}. 
\]

thus defining $P = R_1R_2 \ldots R_{n-1}$, $U$ is a minimum when

\[
\begin{align*}
R_1P &= K \\
R_2 &= K \\
R_{n-1} &= K_i
\end{align*}
\]
So that when all $R_i$ have the same value:

$$R_1 = R_2 = \ldots = R_{n-1} = K^{1/n}.$$ 

Substituting this value in $U$ we obtain

$$U_{tub} - U_s = (n - 1)K^{1/n} + \frac{K}{K^{(1-1/n)}} - n =$$

$$= nK^{1/n} - K^{1/n} + K^{1/n} - n =$$

$$n \left( K^{1/n} - 1 \right);$$

That is,

$$U_{tub} = U_s + n \left( K^{1/n} - 1 \right). \quad (3.1)$$

Now that

$$U_s = \frac{Q_s}{P_s} = \frac{P_1 - P_2}{P_s} = R_s - 1$$

We have

$$R_s = (U_s + 1).$$

So $K$ can be rewritten as

$$K = \frac{2}{R_s} = \frac{2}{U_s + 1},$$

And finally

$$U_{tub} = U_s + n \left[ \left( \frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \quad (3.2)$$
Taking the limit of Equation (2.1) as \( n \rightarrow \infty \), we find the worst-case bound as a function of \( U_s \) to be given by:

\[
\lim_{n \to \infty} U_{lub} = U_s + \ln(K) = U_s + \ln\left(\frac{2}{U_s+1}\right). \tag{3.3}
\]

Thus, given a set of \( n \) periodic tasks and a periodic server with utilization factors \( U_p \) and \( U_s \), respectively the schedulability of the periodic task set is guaranteed under RM if

\[
U_p + U_s \leq U_s + n\left(K^{1/n} - 1\right);
\]
That is if

\[ U_p \leq n \left[ \left( \frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \]  \tag{3.4}

The plot of Equation (2.3) as a function of \( U_s \) is shown in Figure xx. For comparison the RM bound is also reported in the plot. Note that the schedulability test expressed in Equation (3.4) is also valid for all servers that behave like a periodic task.

Using the Hyperbolic Bound, the guarantee test for a task set in the presence of a periodic server can be performed as follows:

\[ \prod_{i=1}^{n} (U_i + 1) \leq \frac{2}{U_s + 1}. \]

Finally, the response time of a periodic task \( \tau_i \) in the presence of a periodic server at the highest priority can be found as the smallest integer satisfying the following recurrent relation:

\[ R_i = C_i + \left\lceil \frac{R_i}{P_i} \right\rceil Q_s + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil C_j. \]

### 3.3 Server Bandwidth Allocation

Provided a set of periodic tasks what method can we use to compute the server parameters \( Q_i \) and \( T_i \) that can guarantee a feasible schedule? The first parameter that needs to be computed is the maximum server utilization \( U_s^{\text{max}} \) that guarantees the feasibility of the task set. Since the maximum response time is not easy to manipulate, because of the ceiling functions, we can derive \( U_s^{\text{max}} \) from the hyperbolic test of the equation below:

\[ \prod_{i=1}^{n} (U_i + 1) \leq \frac{2}{U_s + 1} \]  \tag{3.5}

then we can define:
And from the schedulability of the task set, from Equation 2.1 then it is defined:

\[ P \equiv \prod_{i=1}^{n} U_i + 1 \]  \hspace{1cm} (3.6)

then:

\[ P \leq \frac{2}{U_s+1} \]

therefore

\[ U_s \leq \frac{2-P}{p} \]

\[ U_s^{\max} = \frac{2-P}{p} \]  \hspace{1cm} (3.7)

Thus, the overall system utilization \( U_s \) must be less than or equal to \( U_s^{\max} \). However, for any given \( U_s \) there are an infinite number of pairs \((Q_i, P_i)\) leading to the same utilization value. So the question becomes how do we select the pair that provides the best response times. A logical solution is to assign the server the highest priority based upon the underlying scheduling algorithm. In other words, if the scheduling mechanism were Rate Monotonic then the task with the smallest period would be assigned the highest priority. The problem is just assigning \( P_s < P_1 \) may not be the best approach since a small \( P_s \) implies a smaller \( Q_1 \) which could result in much higher overhead due to the increased context switching between subsystems. Therefore, by assuming that ties are broken in favor of the server then the highest priority of the server could be achieved by setting \( P_s = P_1 \) and then \( Q_s = U_s P_s \).
This chapter provides a description for some of the primary contributions of this research which includes some novel approaches to resource synchronization in hierarchical scheduled systems. Resource sharing in semi-independent hierarchical scheduled systems has not received a significant amount of attention in the research community because it is assumed that most of the subsystems are independent (i.e. virtual machine). However, typical embedded systems are not independent but depend upon sharing resources across subsystems which is why resource sharing is a major topic of this research.

A resource is a software component that can be used by various tasks to synchronize their execution with other tasks in the system. While resources can be private, dedicated to a particular task, or public resources which are shared among tasks, this chapter focuses primarily on shared resources only. In particular, we examine shared resources which need to be protected against concurrent access, also known as exclusive resources.

### 4.1 Introduction to Resource Synchronization in Hierarchical Scheduled Systems

As mentioned in Chapter 2 the primary goal of hierarchical scheduling is to bind the temporal behavior of tasks whose execution times deviate considerably, allowing for the predictable operation of the various subsystems. Therefore, in order to provide this temporal isolation the basic HSF model assumes that each subsystem is independent, however most systems are not entirely independent and resources may need to be shared for correct behavior. In order to provide sharing in a hierarchical scheduled system resources need to be managed at the local and global level. Local resource sharing can be managed by traditional resource access
protocols such as Priority Inheritance Protocol (PIP) [4], the Priority Ceiling Protocol (PCP) [4] or the Stack Resource Policy (SRP) [5]. Global resource sharing requires resource protection at the local and global level which necessitates modifications to the traditional resource access protocols. For this work we choose the Hierarchical Stack Resource Policy (HSRP) [1], which is an extended version of SRP. A brief overview of SRP and HSRP is provided in the following subsections.

4.1.1 Terminology

Before proceeding further to discuss the SRP and HSRP protocols the terms used to describe resource sharing in a hierarchically scheduled system are provided below (also depicted in Figure 4-1).

![Figure 4-1: Task execution time with shared resources](image)

Figure 4-1: Task execution time with shared resources

The periodic task model is defined as \( \tau_i(T_i, C_i, D_i, \{c_{i,j}\}) \) where \( T_i \) represents the task period, \( C_i \) represents the WCET of the task, \( D_i \) defines the relative deadline and \( \{c_{i,j}\} \) represents the set of critical sections within each task \( \tau_i \) shared by a global resource \( R_j \). Each element in the \( \{c_{i,j}\} \) set represents the critical section execution time \( \text{CSET}_{i,j} \) where \( \text{CSET}_{i,j} < C_i \). The resource holding time \( h_{i,j} \) is defined as the duration from when the critical section is entered to the time
when it is exited. The shared resource access time is defined as the duration of the time when the resource is requested to the time the resource was released.

A subsystem is defined as a task set $T_s$ consisting of $n$ tasks such that $S_s \in S$, where $S$ represents the overall system. Each subsystem is characterized by its interface defined as $(P_s, Q_s, RHT_s)$, where $P_s$ is the subsystem period, $Q_s$ is the server budget and $RHT_s$ is the subsystem maximum resource holding time such that $RHT_s = \max\{h_i \mid \tau_i \in T_s \text{ accesses } R_j\}$. The $RHT_s$ is assumed to be less that the server capacity so that $RHT_s < Q_s$.

Each task $\tau_i$ in a subsystem $S_s$ has a preemption level defined as $\pi_i = 1/D_i$. The main idea behind preemption levels is that a task $\tau_i$ can only preempt another task $\tau_j$ if $\pi_i > \pi_j$. The purpose for distinguishing preemptions levels from priorities is in dynamic scheduling environments (e.g. EDF) priorities can change. Each subsystem $S_s$ also has a preemption level $\Pi_s = 1/P_s$, where $S_{si}$ can only preempt $S_{sj}$ if $\Pi_{si} > \Pi_{sj}$. Each globally shared resource $R_j$ has a local resource ceiling $rc_j = \max\{\pi_i \mid \tau_i \text{ accesses } R_j\}$ and one global resource ceiling defined as $RC_s$.

At the subsystem level each task set $T_s$ is scheduled according to an EDF or FPS local scheduler. At the system level each subsystem $S_s$ is scheduled according to an EDF or FPS global scheduler. Additionally, it is assumed that the system provides a mechanism so that at any time $t$ a local scheduler can determine the remaining server budget defined as $Q'_s(t)$.

### 4.1.2 Stack Resource Policy

According to the rules of the SRP protocol while a task $\tau_i$ accesses a local resource the task priority is set to the local resource ceiling $rc_j$. Another task $\tau_j$ within the same subsystem as $\tau_i$ can only preempt $\tau_i$ if $\tau_j$ has a higher priority and $\pi_j > \pi_i$. If the capacity of the subsystem
server is exhausted $Q'_s = 0$ while task $\tau_1$ is still holding the resource then task $\tau_1$ is suspended until the next server budget replenishment period. In order to illustrate the concept of SRP consider the following task set where the priority of three tasks are defined as: $\tau_1 > \tau_2 > \tau_3$. Task $\tau_1$ accesses resource $R_1$, task $\tau_2$ access resource $R_2$ and task $\tau_3$ accesses resources $R_1$ and $R_2$ (see Figure 4-2).

![Stack Resource Policy Example](image)

Figure 4-2: Stack Resource Policy Example

At time $t_0$ the system ceiling $\Pi_s = 0$ because all the resources are available. At time $t_1$ task $\tau_3$ accesses resource $R_1$ and since $R_1$ will also be requested by $t_1$ the system ceiling is set to $\Pi_s = 3$ and subsequently the resource ceiling is set to $rc_j = 3$. Notice that at time $t_2$ even though task $\tau_2$ has a higher priority than $\tau_3$ it is not allowed to preempt $\tau_3$ because task $\tau_2$ preemption level $\pi_2 = 2$ is less than the system ceiling $\Pi_s = 3$. At time $t_7$ task $\tau_2$ is allowed to run since it is the highest priority ready task and its preemption level $\pi_s > \Pi_s$. At time $t_9$ task $\tau_2$ accesses resource $R_2$ and since $\tau_2$ is the highest priority task that could access $R_2$ the system ceiling is set to $\Pi_s = 2$. 


4.1.3 Hierarchical Stack Resource Policy

HSRP extends SRP to work in an HSF. The same reasoning that is used by SRP for the preemption policy is applied to HSRP except from a global scheduling point of view. According to HSRP the global system resource ceiling $RC_s$ is set to the highest priority of any server in the subsystem that accesses a global resource $R_j$. When a task accesses a global resource the priority of the task’s server is set to the global resource ceiling $RC_s$. Additionally, while a task is holding a global resource the priority of the task is increased to the highest priority task in the subsystem.

4.1.4 Server Budget Exhaustion

As mentioned previously global resource sharing requires that a resource be protected at both the local and global level which means a task that locks a global resource will also cause its server to lock the resource. However, there is an additional complication when a task locks a global resource. What happens when a task has a global resource locked but the server’s budget for that task has been depleted? Consider the following scenario, illustrated in Figure 4-3, a high priority task $Task_1$ shares a mutex with a lower priority task $Task_2$ which is managed by a fixed priority periodic server. Given a server budget of $Q_i = 4$ and a period of $T_i = 10$, at time $t_3$, task $Task_1$ preempts $Task_2$ then is blocked on the critical resource. However, when $Task_2$ resumes execution its server budget is not enough to finish the task requiring $Task_2$ to wait until its budget is replenished, thereby creating an additional delay for task $Task_1$. 
Researchers have proposed several solutions to the problem of added delay in critical sections due to server budget exhaustion. One such approach called budget check checks to see if there is sufficient server budget before allowing a task to enter a critical section. If the budget is insufficient the task is not granted access to the resource until the next budget replenishment.

The other approach allows the task to enter a critical section without checking for a sufficient budget. As a result, if the budget is exhausted while still inside the critical section the task is just allowed to continue and consume extra budget until the end of the critical section. There are two slight variations to the protocol on how they handle budget overruns. One variation consumes the extra budget at the expense of other tasks. The result being that other tasks in the subsystem may not receive their full budget allotment. The other method does “payback” to other tasks in the subsystem by taking away a portion of the full budget allotment, of the task that overran, during subsequent replenishment periods.

The problem of budget exhaustion can be amplified during periods of overload as it could further increase the time a critical task would have to wait for the resource. Overload conditions can result because tasks execute longer than expected. Hierarchical scheduling is a general technique that can be used to limit the effects of overruns in tasks with these variable execution
times. However, in the interest of timing guarantees there are distinctions between hard and soft tasks in a hierarchical scheduled system. A hard reservation allows a task to execute at most $Q_i$ (budget) units of time for every $P_i$ (period), whereas a soft reservation allows the task to execute for at least $Q_i$ time units for every $P_i$. This way a soft real-time task can execute more if there is some idle time available. The issue with this is when a hard and soft real-time task share a global resource budget overrun allows the soft real-time task to continue affecting the budget for the hard real-time task. This is one reason hierarchical scheduled systems are generally considered only for soft real-time systems, such as video processing. As an example, consider the same task model mentioned in the previous section and the budget overrun depicted in Figure 4-4.

![Figure 4-4: Server budget inside a critical section with overrun (no payback)](image)

During an overload condition if a served task is allowed to continue while still inside its critical section it can violate the temporal isolation between subsystems. Therefore, our proposal is that current resource sharing algorithms (in hierarchical scheduled systems) are insufficient for hard real-time tasks when a resource is shared across subsystems, specifically during periods of overload.
4.2 Related Work in Resource Synchronization for Hierarchically Scheduled Systems

The schedulability of HSF has been analyzed using fixed-priority global scheduling [6] and EDF bound global scheduling [7]. Initially, HSF designed systems were meant to be independent but researchers realized this approach was not practical as many embedded systems are semi-independent via the sharing of global resources. Research on the HSF was extended to perform schedulability analysis of semi-independent real-time components [4] [5]. The main focus of this work was to reduce the resource holding times that were being incurred during budget expiration.

The SIRAP [8] protocol was developed for fixed-priority preemptive scheduling while the BROE [9] protocol was developed for dynamic-priority scheduling. Their work used a form of budget check to determine if there was enough budget left to enter the critical section. If the remaining budget was deficient to complete the critical section the task was blocked from locking the resource until the next budget replenishment. The limitation with this approach is that the critical section execution time is based upon worst-case analysis. This could lead to resource underutilization due to conservative WCET estimations. Additionally, a priori knowledge of the WCET for a critical section is required which is often difficult to evaluate in applications with variable execution times.

Hierarchical scheduling with resource sharing HSRP [1] and later extended to OPEN-HSRP [8] utilized the budget overrun approach to reduce the resource holding times during budget expiration. Two variations to budget overrun were compared, budget overrun with and without payback. While this approach does provide better flexibility for applications with variable execution times there are some drawbacks. Even though a task is allowed to overrun its
budget there still has to be a limit placed upon the maximum overrun time. In order to prevent unbounded blocking a task is forcefully preempted if it is still holding the resource during the next budget replenishment. This leads to limitations being placed upon the types of shared resources used to those that can safely be aborted to relatively short critical section execution times. Another consideration is because a task can overrun its budget the strict temporal isolation between subsystems could be violated. It is for these reasons that HSRP based systems are generally used for soft real-time systems.

Other recently published work, known as RRP [11], took a different approach to the problem, of resource sharing in hierarchical scheduled systems. Instead of performing a budget check the task was allowed to enter the critical section and unlike HSRP if the budget had expired the task was simply preempted and rolled back. The RRP protocol improved the average case response times and task schedulability as compared to SIRAP and the OPEN-HSRP protocols. However, the limitation with RRP it that can only be used with shared resources that can be safely rolled back (i.e. databases).

4.3 Resource Synchronization in HSF using Preemption

This section provides a description of the Resource Access Control Protocol with Preemption (RACPwP) which is one approach to more effectively manage resource synchronization in mixed-criticality hierarchical scheduled systems.

RACPwP does not rely on WCET analysis of critical section executions times so the protocol is allowed to be more aggressive during task admission which provides improved task schedulability over SIRAP. Given that RACPwP does not use overrun mechanisms higher priority task response time is improved and temporal isolation is strictly enforced as compared to the OPEN-HSRP. Finally, because RACPwP utilizes preemptable critical sections the type of
shared resources that can be used is expanded to include other shared resources and not just
databases as in RRP.

4.3.1 Protocol Description

Similar to other hierarchical scheduling frameworks (SIRAP and OPEN-HSRP) the
RACPwP protocol is based upon a hierarchical scheduling framework. Unlike SIRAP which
performs a budget check before entering a critical section RACPwP always grants access to a
global resource. If the task’s subsystem budget is depleted while still holding a lock the task is
not allowed to continue, as it is in budget overrun mechanisms (e.g. OPEN-HSRP), but instead is
preempted.

Comparable to RRP our method utilizes check pointing and rollback to recover from a
forced preemption but RACPwP incorporates a new technique known as Preemptable Critical
Sections (PCS). The benefit of this approach is it lifts the restriction RRP has by only being able
to handle resources that can be aborted or aborted and rolled back (e.g. database applications).
The protocol is defined as follows:

- Tasks are scheduled based upon their active priorities. Tasks with the same priority are
  executed in a first come first served basis.
- When a task $\tau_i$ requests a local resource and the resource is available the task’s priority is
  raised to the local ceiling priority of the resource $C_{R_j}$.
- If task $\tau_i$ requests a local resource that is locked then $T_i$ is blocked for the duration of the
  longest critical section among the tasks that access resource $R_j$.
- If the subsystem $S_i$ capacity is exhausted while task $\tau_i$ has resource $R_j$ locked then the server
  suspends $\tau_i$.
- If a task $\tau_i$ requests a global resource and the resource is available then the subsystem server’s
  $S_i$ priority is raised to the global ceiling priority of the resource.
- If task $\tau_i$ requests a global resource that is locked then $\tau_i$ is blocked for $B_i$ which is defined as
  the longest time a task in the same application can execute.
• If server $S_i$ capacity is exhausted while $\tau_i$ still has the resource locked then the task is preempted for a maximum duration of its replenishment period.

### 4.3.2 Preemptable Critical Sections

In order to provide a safe mechanism for forceful preemption we introduce a new programming construct for resource synchronization known as preemptable critical sections (PCS). PCS utilizes a form of software transactional memory (STM) to restart a transaction that has been preempted by a higher priority task. After the higher priority task has released the resource the lower-priority task is rolled back and restarted.

The main benefit of this approach is that a higher-priority task gets to execute quickly. In fact, the worst case blocking time is equal to the remaining budget of the lower priority task that is sharing the global resource. Another benefit of this approach includes the elimination of the increased blocking times incurred when a lower-priority task is allowed to overrun its budget as used in the OPEN-HSRP protocol. As a result, RACPwP provides improved response times of hard real-time tasks while relaxing the restriction placed upon critical section execution times by hierarchical scheduled systems.

In order to illustrate the RACPwP protocol, consider the scenario presented previously and depicted in Figure 4-5. At time $t = 2$, $Task_2$ requests and is granted access to the critical section. At time $t = 3$ the hard real-time task $Task_1$ preempts $Task_2$ and executes. At time $t = 4$, $Task_1$ requests the shared resource locked by $Task_2$ and is blocked. At time $t = 5$, the budget for $Task_2$ has expired. $Task_2$ is pre-empted and $Task_1$ is allowed to lock the resource and continue. Finally at time $t = 11$, Task1 completes and $Task_2$ is allowed to once again lock the resource and enter its critical section.
Figure 4-5: Example Resource Access Control Protocol with Preemption

Figure 4-6 provides an example function that simulates a bank balance transfer function implemented using traditional locking mechanisms (i.e. semaphores). Figure 4-7 provides the same function implemented using a preemptable critical section.

```
int transfer () {
    1. sem_wait ();
    2. value = source.balance
    3. value = value – amount
    4. source.balance = value;
    5. value = destination.balance
    6. value = value + amount
    7. destination.balance = value
    8. sem_give ()
}
```

Figure 4-6: Traditional Locking Mechanism

```
int transfer () {
    1. PCS_START ();
    2. value = PCS_LOAD(source.balance)
    3. value = value – amount
    4. PCS_STORE(source.balance)
    5. value = PCS_LOAD(destination.balance)
    6. value = value + amount
    7. PCS_STORE(destination.balance)
    8. PCS_COMMIT
}
```

Figure 4-7: Preemptable Critical Section
In Figure 4-6 the semaphore is acquired at line 1 and for the remaining 6 instructions cannot be preempted. Regardless of the priority the task cannot be preempted until the semaphore is released in line 8. In Figure 4-7 the code snippet illustrates the PCS mechanism which is implemented as a collection of macros. The \texttt{PCS\_START} macro at line 1 performs the initial checkpoint required if the function is preempted and requires restarting. The other macros \texttt{PCS\_STORE} and \texttt{PCS\_LOAD} perform the memory access transactions and the \texttt{PCS\_COMMIT} macro at line 8 commits the transactions to memory. The benefit of these macros is unlike traditional synchronization mechanisms the transfer function can be preempted at any point of its execution (except during an atomic PCS operation). Therefore if a task has exceeded its budget but still inside the critical section (i.e. a \texttt{PCS\_COMMIT} has not been performed) then RACPwP can safely preempt the task. The task is then blocked until the next budget replenishment and allowed to restart again.

It is important to note that there are some limitations associated with using PCS. One is the added computational overhead that software transactional memory imposes of the system. The other limitation is traditional I/O should not be executed within a transaction.

### 4.3.3 Performance Analysis

This section provides the performance analysis of RACPwP as part of a hierarchical scheduled system. The performance is evaluated using worst-case response time analysis. Using the method provided by authors in [12] the worst-case response time of a task \( T_i \) served by a subsystem \( S_i \) occurs during one of the following scenarios:

- The subsystem’s budget is exhausted as soon as the lower priority tasks begin to run and if the task is inside a critical section it is preempted.
- The task $T_i$ and all other higher priority tasks in the application arrive right after the subsystem’s budget is exhausted.
- The subsystem’s budget is replenished but the execution is delayed for as long as possible due to the interference from other higher priority subsystems.

Based upon the scenarios provided above the worst-case response time of a task can be computed by identifying the interval of time where tasks at priority level $i$ or higher can execute.

This interval of time or execution window $w$ is determined by three components:

1. The execution of task $T_i$ along with all higher priority tasks at the $i^{th}$ priority level.
2. The replenishment periods of any complete servers.
3. Interference from higher priority servers (tasks running in higher priority subsystems).
Therefore, the worst-case response time of a task $T_{si}$ can be calculated using the equation:

$$w_{si} = w_{si}^n + J_s \quad (3)$$

where $w_{si}^n$ can be determined by a recurrence function and $J_s$ is the release jitter of task $T_{si}$.

$$w_{si}^{n+1} = L(w_{si}^n) + G(w_{si}^n) + I(w_{si}^n) \quad (4)$$

The worst-case response time analysis was extended by authors in [4] to include resource sharing across subsystems. The load $L(w_{si}^n)$ at the $i$th priority level is expanded by one term to including the affects of local and global blocking factors and is defined as:

$$L(w_{si}^n) = B_{si} + C_{si} + \sum_{j \in \text{hp}(i)} \left[ \frac{w_{sj}^{n+1}}{T_i} \right] C_{sj} \quad (5)$$

where $B_{si}$ is the blocking factor due to local and global resource access. The replenishment period $G(w_{si}^n)$ extended to include global blocking factors is defined as:

$$G(w_{si}^n) = \left( \left\lfloor \frac{L(w_{si}^n)}{C_s} \right\rfloor - 1 \right) (T_s - C_s) + B_s \quad (6)$$

where $B_s$ represents the longest time that a server $S_i$ could be blocked from executing by a lower priority server. The interference $I(w_{si}^n)$ from any higher priority servers is defined as:

$$I(w_{si}^n) = \sum_{x \in \text{hp}(s)} \left[ \max_{0 \leq w_{si}^n} \left( \left\lfloor \frac{L(w_{si}^n)}{C_s} \right\rfloor - 1 \right) T_x \right] (C_x + B_{so}) \quad (7)$$

where $B_{so}$ represents the server overrun time which is the longest time a server $S_i$ may execute. Additionally equation (10) represents a server overrun with no payback. In order to analyze budget overrun with payback $I(w_{si}^n)$ is defined as follows:
\[ I(w^n_{sl}) = \sum_{\forall X \in h_p(S)} B_{xo} + \sum_{\forall X \subseteq I} \left[ \max \left(0, \frac{\left( \left\lfloor \frac{I(w^n_{sl})}{C_s} \right\rfloor - 1 \right) T_x}{T_x} \right) C_x \right] \quad (8) \]

(Note that for RACPwP the server overrun term \( B_s \) in equation (6) is zero since the task would be preempted if the server budget expired while holding a lock on a global resource).

### 4.3.3.1 Results

In the following subsection, we provide a simple example for evaluation purposes between RACPwP and other protocols that use budget overrun mechanisms. For our example, we compare the server and task worst-case response times for RACPwP, OPEN-HSRP with budget payback and OPEN-HSRP without budget payback. The server response time for OPEN-HSRP with budget payback is calculated based upon the following:

\[ w^{n+1}_s = C_s + B_s + \sum_{\forall X \in h_p(S)} B_{xo} + \sum_{\forall X \in h_p(S)} \left\lfloor \frac{w^n_{sl}}{T_x} \right\rfloor C_x \quad (9) \]

The server response time for OPEN-HSRP without budget payback is calculated based upon the following:

\[ w^{n+1}_s = C_s + B_{so} + \sum_{\forall X \in h_p(S)} \left\lfloor \frac{w^n_{sl}}{T_x} \right\rfloor (C_x + B_{xo}) \quad (10) \]

The recurrence functions for equations (9) and (10) begin with \( w^0_s = 0 \) and terminate when \( w^{n+1}_s = w^n_s \) which is the worst-case response time of the server. If \( w^{n+1}_s > T_s \) then the server is not schedulable and therefore not considered. The simulated systems are composed of three separate subsystems each scheduled by the global preemptive periodic server. A global resource is shared between each subsystem \( S_i \) and the critical section execution time is represented as \( CSET_i \). The \( J_i \) term represents the subsystem server jitter.

In order to evaluate the worst case server response times we used 100 simulation runs that used a uniform random number generator to vary the server capacity, server budget and the
critical section execution times. The server parameters ranges that were used to generate the
server parameters are provided in the Table 4-1.

<table>
<thead>
<tr>
<th></th>
<th>$S_i$</th>
<th>$C_i$</th>
<th>$T_i$</th>
<th>$J_i$</th>
<th>$CSET_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>[50,500]</td>
<td>[200,2000]</td>
<td>[150,1500]</td>
<td>[35,200]</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>[1250,2500]</td>
<td>[5000,10000]</td>
<td>[3750,7500]</td>
<td>[35,200]</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>[3000,5000]</td>
<td>[12000,20000]</td>
<td>[9000,15000]</td>
<td>[35,200]</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-2 provides the worst-case response times for RACPwP, OPEN-HSRP without budget
payback (HSRPnP) and OPEN-HSRP with budget payback (HSRPwP).

<table>
<thead>
<tr>
<th></th>
<th>$S_i$</th>
<th>RACPwP</th>
<th>HSRPnP</th>
<th>HSRPwP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>390</td>
<td>390</td>
<td>390</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>2360</td>
<td>3400</td>
<td>2900</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>5550</td>
<td>12150</td>
<td>10012</td>
<td></td>
</tr>
</tbody>
</table>

As shown by the schedulability analysis given in Table 4-2 server response times are
improved with RACPwP since does not perform any budget overrun. Server response times are
practical identical for subsystem $S_1$ since the highest priority server is not subject to overruns.
However, notice that RACPwP does significantly improve server response times for subsystems
$S_2$ and $S_3$ which is not affected by the overrun mechanism.

The next step is to evaluate the worst-case task response times of RACPwP as compared
to OPEN-HSRP with and without budget payback. The recurrence function for worst-case task
response times begins with $w_i^0 = 0$ and ends when $w_i^{n+1} = w_i^n$ where the worst-case response
time is $w_i^n + J_i$. 

45
The task is not schedulable if \( w_i^{n+1} > D_i - J_i \) in which case it is not considered for analysis. Subsystem \( S_2 \) was chosen to execute the tasks as it is the mid-level priority subsystem. For this example, a global shared resource \( CSET_i \) is shared among tasks as well as a local resource. The local resource critical section execution time is represented by \( CSET_{si} \). Similar to the server parameters a random number generator was used to vary the task worst case execution time, the task period and deadline. The local resource execution was also varied. The task parameter ranges that were used to vary the task parameters are defined in Table 4-3.

### Table 4-3: Task parameters

<table>
<thead>
<tr>
<th>( \tau_i )</th>
<th>( Ci )</th>
<th>( Ti )</th>
<th>( Di )</th>
<th>( CSET_{si} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>[1180,2375]</td>
<td>[12500,25000]</td>
<td>[12500,25000]</td>
<td>[150,350]</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>[3150,4500]</td>
<td>[35000,50000]</td>
<td>[35000,50000]</td>
<td>[150,350]</td>
</tr>
</tbody>
</table>

Table 4-3 represents the worst-case response times for all tasks in subsystem \( S_2 \). The subsystem \( S_2 \) was chosen since it’s the mid-level priority subsystem and would best illustrate the effects of our protocol on the overall system. The task worst-case response times are calculated according to the recurrence function (4).

### Table 4-4: Task worst case response times

<table>
<thead>
<tr>
<th>( \tau_i )</th>
<th>( RACPwP )</th>
<th>( HSRPnP )</th>
<th>( HSRPwP )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>7665</td>
<td>8860</td>
<td>9388</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>28900</td>
<td>23800</td>
<td>26200</td>
</tr>
</tbody>
</table>

As shown in Table 4-4 it is evident that shared resource access can have a significant impact on the overall task load. The result being that lower priority tasks may get preempted and have to be restarted. The result is illustrated in the increased response times. The lower priority tasks pays the steepest price as it may suffer from multiple preemptions incurred by the higher
priority tasks. Notice how RACPwP provides improved response times over HSRPnP and HSRPwP for higher priority tasks but lower priority tasks may experience degraded response times. This is the inherent tradeoff with RACPwP. Higher priority tasks which are generally hard real-time benefit from the lower response times, which provides improved determinism. However lower priority tasks which are typically soft real-time are more tolerant of the increased response times.

4.4 Adaptive Resource Synchronization in HSF

This section provides an overview of the adaptive resource synchronization policy which is a resource access protocol that synchronizes access to shared resources in a hierarchically scheduled system. We only consider mutually exclusive resources. Namely, shared memory but could also include other shared resources (e.g. memory-mapped areas, device registers, peripheral devices …) as well. The access to these resources are performed as part of a critical section and protected by a semaphore.

4.4.1 Protocol Description

Similar to the SIRAP and OPEN-HSRP protocols ARAP utilizes a two-level hierarchy for resource management. Resources that are shared within a subsystem are managed with SRP and resources that are shared across subsystems are managed with an extended version of HSRP. The overall sequence of actions for ARAP is provided in the flowchart depicted in Figure 4-8.
4.4.1.1 Budget Exhaustion

The primary difference between the various resource access protocols in an HSF is how the budget exhaustion of a subsystem’s server is handled. As mentioned in previous section the SIRAP protocol performs a budget check while OPEN-HSRP permits budget overflow. Similar to the SIRAP protocol our method also performs a form of budget check. However, instead of
using a static a-priori calculation of the CSET_t, ARAP incorporates feedback to estimate the next resource holding time h_t,j for that critical section execution instance.

Incorporating feedback from previous resource holding times to predict future usage is a valid approach substantiated by previous research used in WCET analysis. Authors in [13] demonstrated that methods which invoke a past behavior approach are 2.3 times more effective for predicting execution times than other methods such as task profiling or machine benchmarking. Similar work has been done by researchers [14] using standard benchmarks, such as gunzip and matrix multiply, and they reported approximately a 95% correlation between past performance. Additionally, other work was done [16] that used past behavior of decoding times for MPEG-compressed video to estimate the decoding time required for the next frame with about a 75% accuracy rate.

The primary benefit of incorporating feedback is that the system can dynamically adapt to changes in CSET specifically during periods of transient overload. In this situation the SIRAP protocol tends to be too conservative because with SIRAP the overload condition would have to be factored into the CSET calculation resulting in task under utilization. While the OPEN-HSRP protocol can better adapt to overload conditions the overrun mechanism could adversely affect the response times of higher priority tasks. Therefore, our approach leads to a more robust and better utilized system with a higher degree of determinism.

4.4.1.2 Feedback Mechanism

Our feedback architecture is implemented as part of the kernel (see Figure 4-9) and consists of a PID (Proportional-Integral-Derivative) controller which is used to estimate the execution time for a task executing inside a critical section. The output or observed value of the
PID is the estimated error ratio (ER) which is defined as the ratio between the actual measured critical section execution time and the previous window of past error ratios. The semaphore request mechanism is used as the actuator of the system determining whether a task is granted access to the critical section. The control action is performed by either allowing the task to acquire the semaphore or to block the task waiting on the semaphore. If a task \( \tau_i \) requests a semaphore (e.g. \textit{srp\_wait ()}) it has to pass the budget check test to acquire the semaphore. The budget check test uses information from the PID controller as well as information from the scheduler to determine if there is enough remaining budget to complete the critical section based upon the projected critical section execution time. To apply the feedback control ARAP uses the PID controller to compute \( \Delta \text{CSET} \) which is used to project the next critical section execution time based upon the ER. Using the basic form a general PID controller defined as:

\[
\Delta \text{CSET}(t) = -C_p \text{ER}(t) - C_i \sum_{IW} \text{ER}(t) - C_D \frac{\text{ER}(t-DW)-\text{ER}(t)}{DW} \tag{11}
\]

where \( C_p, C_i \) and \( C_D \) are the PID control parameters, \( IW \) is the integration window and \( DW \) is the size of the differentiation interval.

Figure 4-9: Feedback architecture

The idea of using a PID controller to help manage a task’s access to a critical section has never been used as a means of resource synchronization in a hierarchically scheduled system.
However, the approach proved accurate in estimating and controlling the amount of time a task would spend in the critical section. As verification tool we used a ground-based satellite command and control embedded system to measure the correlation between the actual execution time and execution times projected by the PID controller. This particular use case was chosen because satellite telemetry can vary significantly. Bit rates can range considerably and the amount of processing required to process a telemetry frame can vary significantly depending upon how densely a telemetry frame is populated. Figure 4-10a represents the measured task execution times as compared to the task execution times projected by the PID controller. Figure 4-10b represents the estimation error between actual and projected execution times. A negative value represents and under-estimation while a positive value represents an over-estimation. Notice most projections are within 10% of the actual execution times.

Figure 4-10: Critical section execution times - projected vs. actual
4.4.1.3 Implementation Considerations

The implementation of ARAP is similar to SIRAP but the locking operations (srp\_wait, and srp\_signal) are modified to utilize the feedback mechanism. A description of the required locking modifications is provided in the subsequent text.

The locking operation is performed by the srp\_wait function so when a task tries to acquire a resource the local scheduler performs a check to determine if there is enough budget to complete the critical section. At the semaphore request time $t$ let the function $calcRHT()$, which utilizes the PID controller, calculate the projected resource holding time so that $h_{i,j} = calcRHT()$. At the same time let the function $getCurBudget()$ get the subsystem server’s remaining budget such that $Q'_s(t) = getCurBudget(t)$. If the task’s projected resource holding time $h_{i,j} < Q'_s(t)$ then the task is allowed to lock the resource and execution continues according

![Figure 4-11: PID critical execution time estimation error](image)
to the rules of HSRP. On the other hand, if $h_{ij} > Q_c$ then the task is not allowed to lock the resource until the next subsystem budget replenishment, this is known as self-blocking. Note that at this time the system ceiling will be equal to the global resource ceiling $\Pi_s = RC_s$, which means that even if a ready task does not use the global resource it could be still be blocked if the task’s preemption level is less than the system ceiling $\pi_i < \Pi_s$. This preemption test ensures that during the next budget replenishment period there will be enough budget for the task to execute inside the critical section.

The release operation is performed by the $srp\_signal$ function which signifies the completion of the critical section. The time that is spent in the critical section is used as the feedback to the PID controller. At the semaphore release time $t'$ let the function $recordRHT(h_{ij}(t))$ record the actual time spent in the critical section.

### 4.4.2 Performance Analysis

This section provides the background for the performance analysis of ARAP as part of a hierarchically scheduled system. Given that both ARAP and SIRAP perform a budget check before locking a global resource the analysis performed by authors [8] for SIRAP can be applied directly to ARAP as well.

#### 4.4.2.1 Local Performance Analysis

According to the authors in [7] each subsystem $S_s \in S$ is schedulable if

$$\forall \tau_i \quad 0 < \exists \epsilon \leq D_i \quad rbf(i, t) \leq sbf_T(t)$$

(12)

where $sbf_T(t)$ is the supply bound function used by authors [8] to calculate the minimum CPU allocations required during an interval of time. Authors in [16] presented a periodic processor
model to characterize the allocations defined by what they called the virtual processor model represented as $T(P, Q)$. The supply bound function (see Figure 4-11) of the virtual processor model is defined as:

$$sbf_T(t) = \begin{cases} t - (k + 1)(P - Q), & \text{if } t \in W^{(k)} \\ (k - 1)Q, & \text{otherwise} \end{cases}$$  \hspace{1cm} (13)

where $k = \max([t - (P - Q)/P], 1)$ and $W^{(k)}$ is defined as the interval $[(k + 1)P - 2Q, (k + 1)P - Q]$.

The request bound function of a task $\tau_i$ is defined as:

$$rbf_{fp} = C_i + I_S(i) + I_H(i, t) + I_L(i)$$  \hspace{1cm} (14)

where $C_i$ is the WCET of task $\tau_i$, $I_S(i)$ is the maximum self-blocking for task $\tau_i$, $I_H(i, t)$ is the interference from tasks with a higher priority than task $\tau_i$ and $I_L(i)$ is the maximum interference by tasks with lower priority than task $\tau_i$ which share a global resource, such that:

$$I_S(i) = \sum_{k=1}^{o} h_{i,k}$$  \hspace{1cm} (15)

$$I_H(i, t) = \sum_{j=1}^{i-1} \left[ \frac{t}{T_j} \right] (C_j + \sum_{k=1}^{o} h_{j,k})$$  \hspace{1cm} (16)

$$I_H(i, t) = \sum_{j=1}^{i-1} \left[ \frac{t}{T_j} \right] (C_j + \sum_{k=1}^{o} h_{j,k})$$  \hspace{1cm} (17)
The resource holding time $h_{i,j}$ of a task $\tau_i$ is defined as the maximum critical section execution $c_{i,j}$ plus the interference from the tasks with a higher preemption level than the ceiling of the resource during the CSET of $c_{i,j}$. Such that $h_{i,j}$ is computed using $W_{i,j}(t)$ as follows:

$$W_{i,j}(t) = c_{i,j} + \sum_{l=\text{ceil}(c_{i,j})+1}^{u} \left\lfloor \frac{t}{T_l} \right\rfloor C_l$$  \hspace{1cm} (19)

where $\text{ceil}(c_{i,j})$ is the ceiling of the resource accessed within the critical section $c_{i,j}$ and $C_l, T_l$ are the worst-case execution time and period of the task that has a higher preemption level than $\text{ceil}(c_{i,j})$. The value $u$ represents the maximum resource $rc_j$ within a subsystem.

4.4.2.2 Global Performance Analysis

For global scheduling analysis the virtual processor model can be extended to a global model $T_s(P_s, Q_s)$ where multiple subsystems $S_s$ can be verified according to equation (3). Therefore, the schedulability test for a fixed priority global scheduler is defined as:

$$W_{i+1} = Q_i + B_k + \sum_{j=1}^{i-1} \left\lfloor \frac{W_j}{\pi_j} \right\rfloor C_j$$  \hspace{1cm} (20)

where $B_k$ of subsystem $S_k$ is the maximum resource holding time with a preemption level less than $\pi_k$.

4.4.3 Performance Results

In this section we compare the performance of ARAP, SIRAP and OPEN-HSRP in terms of the overall system load which is the amount of CPU utilization required to guarantee subsystem schedulability. Schedulability analysis for HSRP was performed by authors in [1] which is very similar to the SIRAP analysis which excludes self-blocking but has to consider the overrun mechanisms.
Similar to SIRAP the local schedulability analysis for OPEN-HSRP is extended from equation (12) as follows:

$$\forall \tau_i \exists t: 0 < t \leq D_i, rbf_{fp}(i, t) + b_i \leq sbf(t)$$

where $b_i$ is the maximum blocking time when $\tau_i$ is blocked by a lower priority task. The supply bound function $sbf(t)$ is defined by equation (13) and the request bound function $rbf_{fp}(i, t)$ is defined as follows:

$$rbf_{fp}(i, t) = C_i + \sum_{k \in HP(i)} \left[ \frac{t}{T_k} \right] C_k$$

where $HP(i)$ is the set of tasks with priorities higher than $\tau_i$. The global schedulability is defined as:

$$\forall S_s \exists t: 0 < t \leq P_s, LBF_s(t) \leq t$$

where the load bound function $LBF_s(t)$ is defined as follows:

$$LBF_s = RBF_s + B_s$$

where

$$RBF_s(t) = Q_s + \sum_{k \in HPS(s)} \left[ \frac{t}{P_k} \right] Q_k$$

where $HPS(s)$ is the set of subsystems with a higher priority than subsystem $S_s$ and $B_s$ is the maximum time that $S_s$ is blocked by lower priority subsystems.

The performance of ARAP is evaluated in terms of the added cost required to ensure schedulability analysis. In other words what is the minimum value of the request bound function $rbf_{fp}(t)$ that would guarantee schedulability. For a synthetic workload we generated random variances of a hierarchical system consisting of 3 separate subsystems such that $S_3 < S_2 < S_1$. Each subsystem $S_n$ consisted of 3 tasks with a global resource being shared between a single task in each subsystem. Each subsystem has a total utilization of 15%. Task periods ranged between 100 and 1000. Figure 4-12 represents the overall system utilization for the 3 task sets required for task schedulability as defined by equation (12).
The resource holding time in Figure 4-12 represents the worst-case critical section execution time of CSET. The synthetic workload generator would vary the critical section execution time using a uniform random generator between 1 and maximum CSET. Notice that SIRAP has the highest system utilization or in other words requires the most CPU utilization in order to provide subsystem schedulability guarantees. This is due to the fact that SIRAP has to use static analysis of critical section execution time which could lead to over estimation for the request bound function. OPEN-HSRP performs slightly better than ARAP in terms of system utilization because ARAP does perform self-blocking.

The data in Figure 3-13 represents the task acceptance rate according to equation (3). We used the same task set as that which was used in Figure 4-12 and varied the critical section execution times but we varied the number of subsystems from 1 to 10. For just one subsystem all the protocols were comparable with no task rejection which is reasonable since there is no global resource sharing. However, as the number of subsystems increase the task acceptance rate drops considerably. Notice how the task acceptance rate for SIRAP is lower than both OPEN-HSRP
and ARAP as soon as the number of subsystems increase, which indicates global resource sharing. This behavior is logical because SIRAP has to use the conservative estimate of worst-case execution time for the critical section. OPEN-HSRP and ARAP are more comparable since both protocols do not rely on worst-case execution time estimates. However, notice that in some instances OPEN-HSRP outperforms ARAP this is due to the overrun mechanisms of OPEN-HSRP while ARAP may self-block a task due to budget constraints the OPEN-HSRP allows the task to just continue.

![Figure 4-14: Task acceptance rate](image)

### 4.4.4 Simulation Environment

This section describes the simulation environment we used to further analyze ARAP which was implemented as a simulation component within RTSIM [17]. RTSIM (Real-Time System Simulator) is a task scheduler simulation and is used primarily for simulating real-time control systems. In order to implement ARAP in RTSIM we extended the existing resource manager class to include the feedback mechanisms. Figure 4-15 presents the overall architecture of how ARAP was integrated with RTSIM.
4.4.4.1 Modeling Transient Overload

A Heaviside step function (26) was used to model the transient overload condition. Heaviside functions are used extensively in control theory to represent different loads. The discontinuous nature of this function maps nicely to an overload situation where we can model periods of nominal, ramp-up and ramp-down behaviors. The Heaviside step function is defined as follows:

$$H(s) = \begin{cases} 
    urf(BCET, ACET), & for \ 0 \leq t < rs \\
    rup(H_{(sk-1)}), & for \ rs \leq t < \frac{rd}{2}, \ H_{(sk-1)} \leq WCET \\
    rdn(H_{(sk-1)}), & for \ \frac{rd}{2} \leq t < rd, \ H_{(sk-1)} \geq ACET \\
    urf(BCET, ACET), & for \ t > rd
\end{cases} \tag{26}$$

Where $urf$ is a uniform random function, $rup$ is a ramp up function, and $rdn$ is a ramp down function. $BCET$, $ACET$ and $WCET$ are best, average and worst case execution times respectively, $rs$ is ramp start, $rd$ is ramp duration. Nominal functionality was considered in the range between $BCET$ and $ACET$. 

Figure 4-15: ARAP architecture in RTSIM with sample task set

59
4.4.4.2 PID Tuning

In order to most effectively use the feedback mechanism we need to determine the values for the proportional, integral and derivative controller parameters. How to best determine the PID parameters, known as PID tuning, in a control system has been intensely studied so for our design we incorporated the Ziegler-Nichols [18] tuning method. Ziegler and Nichols conducted numerous experiments and proposed rules for determining the values for CP, CI and CD based upon the transient ramp response of the plant or output. Since our system manages transient overloads their method which utilizes the same type of ramp response seems like an ideal match. Their response value which is in the form of an S-shaped curve is characterized by two constants the delay time \( L \) and the time constant \( T \). Using these two parameters we can set the values of the \( C_P \), \( C_I \) and \( C_D \) according to the values listed in the Table 4-5.

<table>
<thead>
<tr>
<th>Controller</th>
<th>( C_P )</th>
<th>( C_I )</th>
<th>( C_D )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.2 ( T/L )</td>
<td>0.6 ( T/L^2 )</td>
<td>0.6 ( T )</td>
</tr>
</tbody>
</table>

4.4.4.3 Example Task Set

The example task set consisted of a total of five periodic tasks, two hard real-time and three soft real-time tasks but only two tasks \( T_3 \) and \( T_5 \) shared a critical region and therefore were synchronized by a semaphore. The hard real-time tasks were directly scheduled by the global EDF scheduler while the three soft real-time tasks were managed by a periodic server. Initial task schedulability was based upon the worst-case execution time of the hard real-time tasks and average case execution time for the soft real-time tasks.
For the two tasks \( T_2 \) and \( T_3 \), that shared a semaphore, execution times were modeled to exceed their nominal rates. The Tasks \( T_4 \) and \( T_5 \) which represent hard real-time tasks were allowed to execute up to their predefined \( WCET \) while Task \( T_3 \) was modeled to exceed its bandwidth, thereby generating a transient overload condition. The other soft real-time tasks were simulated at their nominal rates which were defined as a random uniform distribution between \( BCET \) and \( ACET \). (See Table 4-6)

Table 4-6: Example Task Set

<table>
<thead>
<tr>
<th>Task</th>
<th>Resource Utilization</th>
<th>Bandwidth Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>( U_{s1} )</td>
<td>0.05</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>( U_{s2} )</td>
<td>0.15</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>( U_{s3} )</td>
<td>0.30</td>
</tr>
<tr>
<td>( T_4 )</td>
<td></td>
<td>0.50</td>
</tr>
<tr>
<td>( T_5 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order to confirm that an overload condition was present RTSIM was modified to record the estimated instantaneous load \( \rho(t) \) proposed by [19]. According to this method the load is computed in all intervals from the current time \( t \) to each deadline \( [d_i] \). For each interval \([t, d_1], [t, d_2], ... [t, d_n]\) the partial load \( \rho(t) \) is defined as:

\[
\rho_i(t) = \frac{\Sigma_{k: d_k \leq d_i} c_k(t)}{(d_i-t)}
\]  

(27)

where \( c_k(t) \) represents the tasks remaining execution time with a deadline less than or equal to \( d_i \). Therefore the total at time \( t \) is: \( \rho(t) = \max_i \rho_i(t) \). Figure 4-15 displays the recorded instantaneous load defined by equation (27) of a sample simulation as verification that the system was experiencing overload conditions.

61
In order to compare our resource allocation protocol to other approaches that manage the reservation server’s budget we extended the RTSIM simulator to support resource managers that support both budget check (SIRAP) and budget overrun mechanisms (OPEN-HSRP-no payback). One note; is that in our scenario due to the increased level of simulated processing within the critical section during overload conditions the WCET time of the soft real-time exceeded the total server budget so it was denied access to the resource due to the conservative nature of the SIRAP protocol. This event was observed during simulations, for this reason the SIRAP protocol was not included in the results.

We ran simulations using the task set defined in Table 4-2 with the RTSIM simulator to analyze how well the adaptive resource access control protocol performed against standard EDF scheduling with first-come-first-serve resource allocation (no PIP) as well EDF using resource sharing with budget overrun (OPEN-HSRP). We executed sample runs modeling transient overload conditions at 0%, 5%, 10%, 15%, 20% and 25% respectively. We stopped at 25% transient overload because the performance started to degrade so severely that the simulation run
times had to be increased to enable all the scheduled tasks to actually finish. The Figures 4-17 and 4-18 separate out the miss rates between the hard and soft real time tasks.

Figure 4-17: Hard real-time task miss rate

Figure 4-18: Soft real-time task miss rate
EDF-FCFS represents EDF scheduling with standard first-come-first-serve resource management (no PIP), EDF-HSRP represents EDF scheduling with OPEN-HSRP (no payback) resource management and EDF-ARAP represents EDF scheduling with PID feedback control. Depicted in Figure 4-17 the adaptive resource access control protocol outperforms the other methods while it sacrifices some deadline misses in soft real-time tasks to guard against deadline misses in hard real-time tasks. Notice that in Figure 4-17 both scheduling mechanisms EDF-FCFS and EDF-HSRP exhibit hard real-time task misses. Even though EDF-HSRP manages the soft real-time tasks with a periodic server hard real-time task misses are realized because the overrun mechanism allows the task to continue, even though its server budget has been exhausted. In Figure 4-18 EDF-HSRP outperforms EDF-ARAP because the overrun mechanism allows the task to continue, even if it holds a semaphore, at the expense of hard real-time deadline misses.
5 HIERARCHICAL SCHEDULING FOR PERIODIC TASKS IN SYMMETRIC MULTIPROCESSING

In this chapter we present a new hierarchical scheduling framework for periodic tasks in symmetric multiprocessor (SMP) platforms. Currently, there is very little research in the area applying hierarchical scheduling to multi-core platforms. Unfortunately, what work has been done focused mainly on soft-real time applications, such as media processing. For this work, we focus on applying hierarchical scheduling to SMP-based platforms for both hard and soft real-time.

Partitioned and global scheduling are the two main approaches used by SMP based systems where global scheduling is recommended for overall performance and partitioned scheduling is recommended for hard real-time performance. Our approach combines both the global and partitioned approaches of traditional SMP-based schedulers to provide hard real-time performance guarantees for critical tasks and improved response times for soft real-time tasks. Implemented as part of VxWorks, the results are confirmed using a real-time benchmark application, where response times were improved for soft real-time tasks while still providing hard real-time performance.

5.1 Introduction to Real-Time Task Scheduling in Hierarchical Scheduled SMP-Based Systems

The next generation embedded systems are working to consolidate large complex workloads onto multi-core platforms with mixed real-time applications. The existing architecture typically uses distributed uniprocessors connected over a common backplane where one processor may be assigned a soft real-time (SRT) task set and another processor a hard real-time
(HRT) task set. The problem with this approach is it limits the computational throughput and increases costs as compared to multi-core platforms. It is for these reasons; designers are looking to re-host these new complex workloads onto multi-core platforms to reduce the size, weight and power (SWaP) requirements of traditional distributed systems.

Therefore, in this paper we look into symmetric multiprocessing (SMP) because most multi-core systems use SMP architecture. Briefly, SMP is a computing framework that manages the processing of tasks across multiple homogeneous processors or cores\(^2\) that share a common operating system, memory and I/O data path. One major challenge for SMP in mixed real-time scheduling is to effectively balance the competing needs of HRT and SRT tasks, such as temporal isolation, resource allocation or fault mitigation.

There are two main scheduling approaches for a SMP-based system: partitioned and global scheduling. Partitioned scheduling binds a task to a specific processor or core while global scheduling allows a task to migrate across multiple cores. Researchers have studied the schedulability of both approaches and have concluded that no single method dominates the other for all task sets [20]. Global scheduling provides better average case response times by performing load-balancing across multiple cores. However, the superior average case performance of global scheduling is not easily extended to hard real-time performance guarantees. For example, when performing load-balancing a global scheduler may migrate a task to another core and as a result invalidate the local cache. This invalidation process proves costly and can severely impact the determinism of the affected task.

On the other hand, partitioned scheduling statically assigns tasks to a specific core which can control task migration. Also known as CPU affinity, the idea is the designer can specify

\(^2\) Note that core and processor will be used interchangeably to indicate the basic computation unit of the CPU.
which tasks to run on a specific core then the scheduler obeys the order and only runs those tasks on the specified core. It also makes logical sense to bind all the tasks that access the same data to the same core(s) in this way they do not contend over data and ensure the task receives the full attention of the processor. However, when tasks are statically assigned to specific cores an unbalanced load distribution is likely to occur leading to a less than optimal utilization of the overall system.

Another concern involves the diversity and complexity of the various computational workloads in these next generation systems. Processing and criticality requirements may vary significantly where different operating modes could have vastly different workloads. In addition to the computational variations, mission critical type systems must perform continuously in harsh environments where they are expected to perform at least a subset of some critical functions under an overloaded or fault condition. The occurrence of an overload or fault must not hinder the overall survivability of the embedded system. Consequently, what is needed may be a more collective type of resource allocation where tasks are assigned resources according to their functionality requirements. In this way, applications can be grouped by service classes based upon their processing and criticality constraints.

Unfortunately, traditional SMP-based schedulers are not suitable to this type of collective resource allocation because they perform fine-grained scheduling at the task level. Since, these schedulers do not differentiate between tasks of different applications system-wide performance may not be the ideal metric for application specific requirements. Additionally, HRT and SRT tasks have competing objectives. HRT tasks require strict timing constraints where deadline misses are not tolerated. While SRT tasks can accept some deadlines misses but place a greater premium on task response time.
To solve these issues in this paper we present a new multi-core hierarchical scheduling framework (HSP) for periodic tasks in SMP-based systems. Our objective is to provide a hierarchical scheduling mechanism that can more effectively adapt to execution time variations in mixed real-time environments. Traditionally, the approach to scheduling mixed real-time applications has been to provide conservative WCET values to ensure the timing correctness of the HRT tasks. The problem with this approach is it usually leads to underutilized resources and poor response times because the actual WCET value of a task is rarely realized. As a result we look to exploit this underutilization by utilizing both the partitioned and non-partitioned scheduling mechanisms of a SMP-based system.

The benefits of this new scheduler are: (1) Better determinism for hard real-time tasks and improved response times for soft-real time tasks as compared to the global and partitioned scheduling methods of traditional SMP-based schedulers. (2) An application based resource allocation scheme which enhances scalability by reducing excessive interprocessor communication, bus contention and synchronization overhead. (3) A scheduling mechanism which provides for improved resource utilization and task acceptance rates. (4) Temporal isolation for hard real-time tasks where lower priority tasks cannot affect the timing behavior during overload or fault conditions.

5.1.1 Terminology

We consider a periodic task model defined as $\tau_i \left(T_i, C_i^{Lo}, C_i^{Hi}, D_i \right)$, where $T_i$ is defined as the task period, $C_i^{Lo}$ and $C_i^{Hi}$ are defined as the average case execution time (ACET) and the worst case execution time (WCET) respectively and finally $D_i$ is defined as the relative deadline. It is assumed that each task $\tau_i$ is a constrained task such that $C_i^{Hi} \leq D_i \leq T_i$. Each task $\tau_i$ must
receive $C_i^{Hi}$ within $D_i$ or it is considered late. It is also assumed that $C_i^{Hi}$ processor units are assigned to a task in a non-concurrent manner.

A subsystem (i.e. application) consists of a task set defined as a collection of periodic tasks $T_s = \{\tau_1, \tau_2, \ldots, \tau_n\}$. A system $S$ consists of $n$ homogenous processors while a subsystem consists of $m$ processors such that $1 \leq m \leq n$. Each subsystem is characterized by a multiprocessor resource model [21] which specifies the resource supply provided to the subsystem (also known as a clustering). The multiprocessor periodic resource (MPR) model is defined as $(P_s, Q_s, m_i)$, where $Q_s$ provides the resource budget over $P_s$ time units to a subsystem consisting of $m_i$ processors. Therefore, a schedulable subsystem must meet the condition $Q_s \leq mP_s$.

In uniprocessor scheduling the supply bound function (sbf) is used to bound the supply required for schedulability of the subsystem. Authors in [21] extended this approach for hierarchical multiprocessor frameworks for deriving schedulability conditions of the subsystem. Therefore, the supply bound function for a multi-core subsystem sbf$_s$ is defined as:

$$sbf_s(t) = \begin{cases} kQ_s + \max\{0, \left\lfloor \frac{t - (P_s - \frac{Q_s}{m})}{P_s} \right\rfloor Q_s, P_s \right\rfloor + Q_s \right\rfloor & t \geq P_s - \left\lfloor \frac{Q_s}{m} \right\rfloor \\ 0, Otherwise \end{cases}$$

where, $k = \left\lfloor \frac{t - (P_s - \frac{Q_s}{m})}{P_s} \right\rfloor$ and $I = t - 2P_s + \left\lfloor \frac{Q_s}{m} \right\rfloor$. Additionally, a lower bound of the sbf$_s$ has been derived for improved schedulability. The lower bound supply lsb$_s$ function is defined as:

$$lsb_s(t) = \frac{Q_s}{P_s} \left( t - 2 \left( P_s - \frac{Q_s}{m} \right) \right)$$
The schedule for a subsystem that generates the resource supply in a time interval of \([0, t]\) is shown in Figure 1 along with the linear lower bound function. In Figure 5-1 we define \(\alpha = \left\lfloor \frac{Q_s}{m} \right\rfloor\) and \(\beta = Q_s - m\alpha\).

![Figure 5-1: Supply bound and lower supply function for a subsystem](image)

The MPR model presented by authors in [21] presents a framework that allows a subsystem exclusive access over a share of the multi-core platform. This share is then guaranteed by the sbf to provide a minimum resource supply to a subsystem. Therefore, HSP can utilize the MPR model to provide temporal isolation and schedulability guarantees between subsystems.

### 5.1.2 Hierarchical Scheduling in a Symmetric Multiprocessor Platform

The basic framework of a hierarchically scheduled system for a uniprocessor platform is composed of multiple applications (subsystems) where each subsystem can be composed of a single or multiple tasks (see Chapter 2). A global scheduler controls which subsystem is allocated the processor while the local scheduler determines which subsystem’s task should actually execute.
This two-level hierarchical scheduling approach is general enough in that it can be extended to a multiprocessor platform. In this case the scheduling of tasks within a subsystem, across \( m \) processors can be performed by the subsystem (local) scheduler while the scheduling of subsystems across the multiprocessor platform is performed by the system (global) scheduler. For example, consider a system where the overall utilization for each subsystem is \( \sum_{i=1}^{n} \frac{C_i}{T_i} \) and \( S_1 = 1.3, S_2 = 0.133 \) and \( S_3 = 1.122 \) then the overall budget is 2.5 and \( m = 3 \), then the global scheduler will provide two units of resource from two processors and the remaining 0.5 units will be provided by the third processor.

### 5.2 Related Work in Hierarchical Scheduling for SMP-Based Environments

A considerable amount of research has also been performed with hierarchical scheduling in a uniprocessor environment [3][12][22]. There has also been a fair amount of work in investigating how resources are shared across subsystems in an HSF [1][9][8]. However, there has not been a lot of work performed in actually applying a hierarchical scheduler to a multi-core environment. This lack of research is due in part to the fact that existing hierarchical scheduling algorithms are not easily extendable to multi-core environments. A couple reasons is that existing algorithms do not incorporate the inherent parallelism of a multi-core system and unfairness or task starvation can result if applied in a naïve manner.

Authors in [23] have presented a hierarchical multiprocessor algorithm known as H-SMP which was designed for a SMP-based platform. Their approach is to take a task set (i.e. an application) and assign it to the various cores in the subsystem based upon the application’s level of parallelism and service requirements. Applications with higher service requirements would be allocated a higher bandwidth partition. For example, applications with soft real-time
requirements would be receive a higher service level than applications with a best-effort type of service requirement. The primary limitation of this approach is that the CPU partitioning is done statically based upon a priori simulated workloads which may not represent real-world applications. In particular this static bandwidth partitioning may not achieve the best CPU partitioning for a dynamically changing workload. Another drawback is there is no explicit notion of criticality for adaptability to changing computational environments. In other words, tasks are assigned fixed budgets based upon their pre-determined WCET values where overly conservative WCET estimates could lead to system under utilization or higher task rejection rates.

Additional work was done by authors [24][25] to provide a mixed-criticality scheduling framework for real-time operating systems (RTOS). Their approach was to use hierarchical scheduling to temporally isolate tasks of different criticality levels. A different scheduling algorithm was assigned to each criticality level. For example, tasks with the highest criticality were assigned a cyclic executive scheduler while less critical tasks were assigned other schedulers like earliest deadline first (EDF). Temporal isolation is enforced by a server with a specific budget which is statically assigned to each critically level.

There has also been some work done [26][27][28] in semi-partitioned scheduling in multiprocessors. The idea is that some tasks are assigned according to the partitioned scheduling approach while other tasks are assigned by global scheduling and therefore allowed to migrate. In order to determine how tasks are assigned the authors took a look at the task workload and then tried to assign that tasks to processors accordingly. For example, tasks with a high workload (i.e. high utilization factor $U_i = \frac{C_i}{T_i}$) would be partitioned while tasks with a low workload would be scheduled globally. Other approaches have looked at how to assign tasks to reduce cache
misses [29] by using partitioned scheduling for the task most likely to generate a high number of cache invalidations. The main limitation with these approaches are that the processor assignments are done a priori with no real notion of criticality for HRT or SRT tasks to adapt to computational changes, such as task overloads.

In our work we take an adaptive approach where non-critical resources are assigned dynamically based upon environmental changes. Instead of static partitioning tasks are allocated based upon a feedback mechanism that the scheduler uses to adjust resource allocation to more effectively adapt to diverse computational workloads at run time. In order to support a service level requirement approach like H-SMP tasks are guaranteed a certain budget but are allowed to share any unused budget by employing capacity sharing mechanisms. A type of capacity sharing algorithm, known as slack stealing [30] is used which allows a lower-priority task to share the bandwidth of a higher priority task. In this way critical functions can be guaranteed a certain level of service but any unused resource can then be re-allocated to task with a lower service level thereby improving the performance, such as reduced response times, of the lower priority task.

5.3 HSP Algorithm Description

This section provides an overview of the HSP scheduling framework which is used to more effectively manage HRT and SRT tasks on a symmetric multiprocessing platform. Our approach employs a two-level hierarchical scheduled framework (see Figure 5-2) to provide resource partitioning and temporal isolation between subsystems. Additionally, HSP utilizes elements of both the partitioned and global scheduling approaches to maximize the benefits of both scheduling mechanisms.
However, unlike uniprocessor based hierarchical scheduling SMP-based hierarchical scheduling needs to contend with tasks that can be stationary or migratory. In order to account for this added complication SMP-based hierarchical scheduling requires enhanced functionality which includes: processor assignment, task set schedulability analysis and run-time scheduling. Processor assignment is the algorithm that determines how an application is assigned to the various processors allocated by the subsystem. The tasks that comprise an application are assigned to processors based upon a combination of mixed-criticality scheduling and semi-partitioned scheduling. The schedulability analysis determines whether the HRT/SRT task set is schedulable on a specific processor. Run-time scheduling determines when tasks execute as well as manage when a task should migrate to another idle core in the subsystem.

### 5.3.1 Processor Assignment

HSP like other traditional partitioned scheduling approaches assigns each task to a particular processor based upon some type of bin-packing heuristics. HRT tasks with strict
timing constraints are assigned to a specific core first according to the chosen heuristic and if the schedulability condition can be satisfied for that core. In this way HRT tasks can get the full attention of the processor and improve the deterministic behavior of the task. Consider Table 5-1 that defines a task set for the example Subsystem1 depicted in Figure 5-2.

Table 5-1: Example subsystem task set

<table>
<thead>
<tr>
<th>Task</th>
<th>Core</th>
<th>$T_i$</th>
<th>$C_i^{lo}$</th>
<th>$C_i^{Hi}$</th>
<th>$D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>p</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>p</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>p</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>p</td>
<td>20</td>
<td>2</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>g</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>$\tau_6$</td>
<td>g</td>
<td>20</td>
<td>2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>$\tau_7$</td>
<td>g</td>
<td>25</td>
<td>2</td>
<td>5</td>
<td>25</td>
</tr>
</tbody>
</table>

According to Table 5-1 tasks that are partitioned (p) are considered HRT tasks are statically assigned to a specific core and not allowed to migrate. Tasks that are global (g) are considered SRT tasks and allowed to migrate across cores in the subsystem. This is similar to mixed-criticality scheduling that assigns highly critical tasks to specific cores but allows less critical tasks to migrate.

For the purpose of schedulability guarantees the HRT tasks are allocated a budget, by the hierarchical scheduler, equal to the task’s WCET value($C_i^{Hi}$), in this way tasks are guaranteed a fixed processing time by the subsystem’s local scheduler. The HRT tasks are assigned to a core based upon the next-fit bin-packing heuristic and since the rate monotonic (RM) algorithm is optimal for fixed priority scheduling it is used as the determination of schedulability for partitioned tasks (see Algorithm 5-1). Therefore, the maximum utilization $U_{si}$ for a core in a subsystem as defined by RM is:
\[ U_{st} = \sum_{i=1}^{n} \frac{c_{ti}}{t_{i}} \leq n \left(2^{1/n} - 1\right) \]  

(3)

**Algorithm 5-1: HRT Task Assignment Algorithm**

<table>
<thead>
<tr>
<th>Algorithm 1 HSP HRT task processor assignment algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> The HRT/SRT task set ( T_s ) and the processors ( m ) assigned to the subsystem ( S_t )</td>
</tr>
<tr>
<td><strong>Output:</strong> On each processor ( p_i ) a executable (or not schedulable) task set.</td>
</tr>
<tr>
<td>1: FOR each ( \tau_i \in T_s )</td>
</tr>
<tr>
<td>2: IF ( \tau_i ) is not a HRT task then</td>
</tr>
<tr>
<td>3: continue</td>
</tr>
<tr>
<td>5: ( u_i = C_i/T_i )</td>
</tr>
<tr>
<td>4: Assign ( \tau_i ) to processor ( p_i ) based upon ( u_i ) and next-fit bin packing heuristic</td>
</tr>
<tr>
<td>5: Let ( p_i^{bn} ) be the set of HRT tasks assigned to processor ( p_i )</td>
</tr>
<tr>
<td>6: ENDIF</td>
</tr>
<tr>
<td>7: Execute HSP task-splitting algorithm on processors in subsystem ( S_t )</td>
</tr>
</tbody>
</table>

From the example task set shown in Table 5-1 and the multi-core system depicted in Figure 5-2 the HRT tasks would be assigned a particular core as illustrated in Figure 5-3.

![Figure 5-3: Partitioned task core assignments](image)

After the HRT tasks are assigned to their respective cores the SRT tasks are assigned based upon the remaining resource capacity. If the SRT task does not fit onto a particular core to support the full execution capacity then the task is split across cores in the subsystem. Task splitting is based upon semi-partitioned scheduling which is defined as a task \( \tau_i \) that is executed
on $l_i$ processors where $l_i \geq 2$. There are $l_i$ subtasks denoted by $\tau_{i1}, \tau_{i2}, \ldots, \tau_{il_i}$, which are synchronized where no subtasks can run in parallel and each subtask $\tau_{il_i}$ has a computation time $C_{il_i}$ such that $C_i = \sum_{j=1}^{l_i} C_{il_i}$. The algorithm for splitting a task $\tau_i$ is provided in Algorithm 2.

Consider the example provided below of how a task may be split across more than one processor.

![Figure 5-4: Split task across two processors](image)

**Algorithm 5-2: HRT Task Assignment Algorithm**

```plaintext
Algorithm 2 HSP task-splitting processor assignment algorithm

**Input:** The HRT/SRT task set $T_i$ and the processors $m$ assigned to the subsystem $S_i$

**Output:** On each processor $p$, a executable (or not schedulable) task set.

1. FOR each $\tau_i \in T_i$
2.   IF $\tau_i$ is not a SRT task then
3.     continue
4.   find processor $m^*$ with maximum slack potential
5.   return unschedulable if $U_{m*} \leq U_{si}$
6.   IF $U_{m*} + C_i/T_i \leq U_{si}$ then
7.     assign subtask $\tau_i^k$ to processor $m^*$
8.   ELSE
9.     split task $\tau_i$ again where $C_i^{l_i-1} \leftarrow C_i^k - (U_{si} - U_{m*})T_i$
10.   $C_i^k \leftarrow (U_{si} - U_{m*})T_i$
11.   assign subtask $\tau_i^k$ to processor $m^*$, where $U_{m*} \leq U_{si}$
12. ENDIF
```

77
While Figure 5-4 illustrates how a split task could be split it does not describe the criteria used to assign the split tasks to the various processors in the subsystem. Traditional approaches have been to assign each share to processors with subsequent indexes so that $\tau_1^1$ would be assigned to $P_1$ and $\tau_1^2$ would be assigned to $P_2$. With semi-partitioned scheduling most tasks are assigned to a particular processor to reduce overhead while the remaining tasks are split to improve schedulability. The problem with this approach is there is no real notion of criticality and tasks are assigned to a processor based upon their respective WCET values which are typically conservative. Our approach with HSP is different in that task criticality is considered by assigning HRT tasks first ensuring that the tasks will be fixed to a particular processor thereby reducing runtime overhead. The schedulability is maintained for the SRT tasks by performing task-splitting and task response times are improved by taking advantage of the potential unused processing capacity, also known as slack. This slack potential is then used by HSP for processor assignment of SRT tasks. SRT tasks whether they requiring splitting or not are then assigned to available cores based upon the maximum slack potential for that core. Note that this slack potential is determined not by the WCET of a HRT task but rather by their average execution time denoted by $C_t^{LO}$. In this way the maximum potential can be identified which represents a much less conservative calculation for improving task response times. The set of algorithms for identifying slack and taking advantage of it is known as slack stealing. A brief overview of slack stealing is provided in the subsection below; for more detail readers are encouraged to review the references.

5.3.1.1 Slack Stealing

According to Equation (3) a task set that meets the criteria will always make its deadlines. The problem is this criterion is based upon WCET values which are usually
conservative calculations and there tends to be a large gap between the WCET value and the actual processing time of a HRT task. This gap, known as slack, presents an opportunity to minimize the response times of a SRT task. Authors in [30][31] describe how the slack is found by mapping out the processor schedule of the HRT tasks over their hyperperiod in a task mapping table. The table is then examined to determine the slack present between the deadline and the next invocation of the task. In turn, this table is then examined by HSP to help identify the core(s) with the maximum slack time potential for SRT task processor assignment.

5.3.2 Task Scheduling

The local scheduler of a subsystem in HSP is responsible for scheduling of tasks on the various cores of the subsystem. Scheduling for the HRT tasks are straightforward in that traditional scheduling mechanisms, such as RM, where the priorities of each task are assigned so that: \(\tau_4 < \tau_3 < \tau_2 < \tau_1\). Similar to HRT tasks priorities are assigned according to the RM except SRT tasks always have a lower priority than HRT tasks, such that SRT < HRT, except during slack stealing periods. During periods of slack stealing the SRT task is temporarily promoted to the same priority level as the HRT task that finished with some available slack time. In this way another HRT task of lower priority cannot preempt a SRT task while it is stealing the slack of another HRT task.

During run-time after a HRT task completes the local scheduler looks to exploit the slack time of an HRT tasks to improve a SRT task’s response time. The run-time slack of a HRT task \(\tau_i\) is based upon the budget \((C_i^{Hi})\) of task \(\tau_i\) provided by the subsystem’s \(S_i\) local scheduler. The task’s budget for the subsystem’s \(S_i\) local scheduler of a HRT task along with the feedback from the task provides the information needed to determine if there is any potential slack available to the SRT tasks. In order to calculate the slack at some arbitrary time \(t\) we look at the unused
server budget of an HRT task in the interval $[t, t + D_i(t)]$. Therefore, the slack is determined by
the length of that interval less than the actual unused budget available from all of the HRT tasks
that fall into that interval. The slack is defined as $s_i(t) = \sum_{j \in \text{hrt}(i)} (Q_j - c_j)$ that is available to
any SRT task at some arbitrary time $t$ and $c_j$ is the actual processing time of the HRT task. As an
example consider the example task set in Table 5-1. Figure 5-4 represents the tasks scheduled on
the first core while Figure 5-5 represents the tasks scheduled on the second core. The up arrow
represents task start time and the down arrow represents the task completion time.

The HRT task set is statically assigned to a core and based upon the next-fit bin-packing
heuristic tasks $\tau_1$ and $\tau_3$ are assigned core 1 while tasks $\tau_2$ and $\tau_4$ are assigned core 2. The
highest priority SRT task $\tau_5$ if the first task scheduled to run on either core when there is
available processing or slack time. At time $t_1$ task $\tau_5$ is allowed to run by stealing the slack from task $\tau_1$ but at time $t_2$ is preempted by the HRT task $\tau_3$. Task $\tau_5$ is then allowed to steal slack from task $\tau_3$ at time $t_4$ and from task $\tau_1$ then complete execution by time $t_7$.

5.4 Schedulability Analysis

With the HSP all tasks execute up to their worst case execution time $C_i^{Hi}$ but the local scheduler prevents the tasks from executing any further. If a task executes further than $C_i^{Hi}$ it is considered in fault and aborted or considered overloaded and descheduled until it is safe to be executed again. This section presents the response time analysis for HSP as it relates to partitioned and non-partitioned scheduling.

As mentioned in Section 5.1 the tasks are scheduled by a fixed priority preemptive scheduler and the task priorities are assigned according to the RM algorithm. Priority ($p$) is derived from the deadlines of the tasks, such that for any two tasks $\tau_i$ and $\tau_j$ their deadlines $D_i < D_j \Rightarrow p_i > p_j$. To test for schedulability, the standard Response Time Analysis (RTA) [19] [20] for uniprocessor scheduling can be extended to HSP. RTA first computes the worst-case completion time for each task (i.e. response time $R_i$) and then compares that value to the task deadline, such that $R_i \leq D_i$ for task $\tau_i$. The response time value is calculated using recurrence relations:

$$R_i = C_i + \sum_{\tau_j \in hp(i)} \left[ \frac{R_j}{T_j} \right] C_j$$

(4)

where $hp(i)$ defines the set of tasks with a higher priority than the task $\tau_i$. The general response time Equation (4) can then be applied to mixed critically systems [12] where the LO-criticality and HI-criticality mode schedulability can be verified. HSP can then adapt this
analysis and apply it to HRT tasks which are considered HI-criticality and SRT tasks which are considered LO-criticality. Standard RTA for a uniprocessor can be applied for SRT tasks as follows:

\[ R_i^{Lo} = C_i + \sum_{j \in hp(i)} \left( \frac{R_j^{Lo}}{T_j} \right) C_j^{Lo} \]  \hspace{1cm} (5)

where \( hp(i) \) is the set of SRT tasks with a higher priority than task \( \tau_i \). The same analysis can also be applied to HRT tasks as follows:

\[ R_i^{Hi} = C_i + \sum_{j \in hpH(i)} \left( \frac{R_j^{Hi}}{T_j} \right) C_j^{Hi} \]  \hspace{1cm} (6)

where \( hpH(i) \) is the set of HRT tasks with a higher priority than task \( \tau_i \). For uniprocessor based systems the schedulability test is determined by calculating the response times of all tasks in an interval starting with a critical instant (case where all tasks experience their WCET) and comparing that to the task deadlines. However it has been shown [20] that it is a NP-hard problem when analyzing globally scheduled periodic tasks. The issue is that it is not easy to find a “representative” interval to represent the start of the critical instant. As a result, in a multicore system only sufficient results can be determined in a reasonable amount of time. Authors in [22] provide a sufficient RTA-based approach for schedulability tests for global scheduled multicore systems. The test is based upon the RTA test of Equation (4) and operates as follows:

\[ R_i^{max} \leftarrow C_i + \frac{1}{m} \sum_{j \in hp(i)} \left( \frac{R_j^{max}}{T_j} \right) C_j + C_j \]  \hspace{1cm} (7)

The schedulability analysis for semi-partitioned systems can then be derived by combing equation (4) and equation (7). To determine the schedulability for SRT and HRT tasks using average case execution time:

\[ R_i^{Lo} \leftarrow C_i^{Lo} + \frac{1}{m} \left( \text{SRT}(\tau_i^{Lo}) + \text{HRT}(\tau_i^{Lo}) \right) \]  \hspace{1cm} (8)
where SRT\(\left(\tau^L_i\right)\) represents the SRT task set average execution times such that:

\[
SRT\left(\tau^L_i\right) = \sum_{t_j \in hp(i)} \left[ \frac{R^L_i}{T_{j}} \right] C^L_{j} + C^L_{i}
\]  
(9)

And HRT\(\left(\tau^L_i\right)\) represents the HRT task set average execution times such that:

\[
HRT\left(\tau^L_i\right) = \sum_{t_j \in hpH(i)} \left[ \frac{R^L_i}{T_{j}} \right] C^L_{j}
\]  
(10)

where hpH\((i)\) is the set of HRT tasks that are assigned to processor \(P_i\). Additionally, to determine the schedulability for SRT and HRT tasks using worst case execution time:

\[
R^H_i \leftarrow C^H_i + \frac{1}{m} \left( SRT\left(\tau^H_i\right) + HRT\left(\tau^H_i\right) \right)
\]  
(11)

\[
SRT\left(\tau^H_i\right) = \sum_{t_j \in hp(i)} \left[ \frac{R^H_i}{T_{j}} \right] C^H_{j} + C^H_{i}
\]  
(12)

\[
HRT\left(\tau^H_i\right) = \sum_{t_j \in hpH(i)} \left[ \frac{R^H_i}{T_{j}} \right] C^H_{j}
\]  
(13)

Consider the task set represented by Table 5-1 in Section 5.1 the schedulability analysis for both SRT and HRT would be as follows.

**Table 5-2: Example task set with response times**

<table>
<thead>
<tr>
<th>Task</th>
<th>Core</th>
<th>(T_i)</th>
<th>(C^L_i)</th>
<th>(C^H_i)</th>
<th>(D_i)</th>
<th>(R^L_i)</th>
<th>(R^H_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_1)</td>
<td>p</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>p</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(\tau_3)</td>
<td>p</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>(\tau_4)</td>
<td>p</td>
<td>20</td>
<td>2</td>
<td>5</td>
<td>20</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>(\tau_5)</td>
<td>g</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>(\tau_6)</td>
<td>g</td>
<td>20</td>
<td>2</td>
<td>4</td>
<td>20</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>(\tau_7)</td>
<td>g</td>
<td>20</td>
<td>2</td>
<td>5</td>
<td>25</td>
<td>8</td>
<td>58</td>
</tr>
</tbody>
</table>

83
5.5 Performance Analysis

For the purpose of comparisons, we used a combined SRT/HRT periodic task set that comprised a single subsystem (i.e. application) and spanned up to $m$ cores, where $m = 2, 4, 8$. Task periods ($p_i$) were chosen using a uniform random distribution from the list \{0.25Hz, 0.5Hz, 1Hz, 2 Hz, 4Hz, 5Hz, 8Hz, 10Hz, 20Hz, 25Hz, 50Hz, 100Hz, 200Hz\}. The list was created to represent some typical rates of periodic tasks. Overall system utilization ($u_{sys}$) for each processor ranged from $[0.50, 1.00]$ in increments of 0.05. Individual task utilization ($u_i$) was randomly generated with an expected value of 0.20 and a standard deviation of 0.15. The number of tasks in the set were determined by the summation of the individual tasks where $\sum_{i=0}^{n} u_i = u_{sys}$. The execution time ($c_i$) was calculated based upon the task period and task utilization such that $c_i = p_i * u_i$. The HRT/SRT tasks were randomly divided from the generated task set with an expected value of $\frac{n}{2}$ and a standard deviation of $n-2$.

HSP was compared against four other semi-partitioning algorithms used in mixed-criticality systems, DU-RM, DU-Audsley [34], DC-RM and DC-Audsley. Each algorithm, including HSP utilizes the next-fit bin packing heuristic but differ on processor and priority assignment. The DU-RM algorithm decreasingly assigns tasks based upon the task utilization and determines feasibility based upon the RM scheduler. In other words the task with the highest utilization factor is assigned to the first available processor. DU-Audsley is similar to DU-RM except Audsley’s priority assignment is optimal for a given processor but the complexity is much higher than RM assignment. The DC-RM algorithm performs processor assignment based upon the decreasing criticality of a task so HRT tasks would be assigned to a processor before a SRT task. DC-Audsley also performs processor assignment based upon the task criticality but its priority assignment is different than DC-RM. Our approach with like DC-RM and DC-Audsley
assigns a task based upon criticality but differs in that if there is not enough available utilization HSP will spilt tasks across any available processors. This has the potential to significantly improve schedulability.

For the simulations we generated 10,000 task sets from the parameters described in the previous paragraph. The task sets were determined to be schedulable if every task in the set was successfully assigned to the group of cores defined by the subsystem $S_i$. The performance criteria for the processor assignment algorithm was determined by the success ratio of the number of tasks scheduled by the number of submitted tasks accepted, defined as follows:

\[
\frac{\text{number of scheduable task sets}}{\text{number of attempted task sets}}
\]

The overall subsystem utilization was determined by $u_{sys} \times m$, so that 1.0, 2.0 and 4.0 represents 50% utilization for $m = 2, 4, 8$ respectively. The data in Figures 5-8, 5-9 and 5-10 illustrates the results from $u_{sys} = [0.5, 1.0]$ where HSP clearly provides better schedulability than the other processor assignment algorithms. Note that the other algorithms start to report failure around 0.5 to 0.7 while HSP does not start to report failure until close to 0.7 to 0.8. This coincides with other work [32][33] that states maximum schedulability for RM or DM is about 88% for uniprocessors. Also notice that HSP outperforms the other algorithms as the number of cores increase because this provides HSP the opportunity to share more of the computation across the various cores in the subsystem.
Figure 5-7: Task set schedule simulation (2 cores)

Figure 5-8: Task set schedule simulation (4 cores)
5.5.1 Hardware Platform

HSP was implemented as described in the previous section with VxWorks 6.9 on a Freescale T4240: QorIQ 12 core (24 virtual-core) communications processor.

For evaluation purposes we ported the SNU Real-Time Benchmark Suite [36] and compared response times and overall system utilization using partitioned, non-partitioned (global) and hierarchical scheduling. The SNU real-time benchmark suite contains small C programs used for worst-case execution time analysis. This benchmark was chosen because it is completely structured (no unconditional jumps, no loop body exits, no switch or do-while statements and no library calls or specific systems calls. The programs are mostly numeric and DSP algorithms.

In order to represent the periodic task model of an embedded system a subset of the programs in the benchmark suite were chosen and assigned arbitrary task rates (see Table 5-3).
Table 5-3: Simulated Periodic Task Set

<table>
<thead>
<tr>
<th>C Program</th>
<th>Task</th>
<th>Rate</th>
<th>$C_{i}^{lo}$</th>
<th>$C_{i}^{hi}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmul</td>
<td>$\tau_1$</td>
<td>50Hz</td>
<td>1.7ms</td>
<td>5.1ms</td>
</tr>
<tr>
<td>fft1</td>
<td>$\tau_2$</td>
<td>40Hz</td>
<td>2.7ms</td>
<td>5.4ms</td>
</tr>
<tr>
<td>fir</td>
<td>$\tau_3$</td>
<td>20Hz</td>
<td>10.4ms</td>
<td>20.8ms</td>
</tr>
<tr>
<td>lms</td>
<td>$\tau_4$</td>
<td>10Hz</td>
<td>12.6ms</td>
<td>25.2ms</td>
</tr>
<tr>
<td>ludcmp</td>
<td>$\tau_5$</td>
<td>40Hz</td>
<td>6.8ms</td>
<td>13.6ms</td>
</tr>
<tr>
<td>minver</td>
<td>$\tau_6$</td>
<td>10Hz</td>
<td>3.5ms</td>
<td>10.5ms</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>$\tau_7$</td>
<td>5Hz</td>
<td>2.2ms</td>
<td>11.0ms</td>
</tr>
</tbody>
</table>

The tasks sets were assigned as HRT = \{\tau_1, \tau_2, \tau_3, \tau_4\} and SRT = \{\tau_5, \tau_6, \tau_7\}. The HRT/SRT task sets comprised a single subsystem $S_1$ which was allocated two cores in the hierarchical system. The HRT/SRT task sets were conceived so that if the $C_{i}^{hi}$ value for each SRT task was realized then the task set is not schedulable and an overload condition would result. In order to evaluate the effectiveness of HSP the execution times of the overall task sets were increased from [0.00, 1.00], where 0.0 indicates all tasks are executed at their respective $C_{i}^{lo}$ levels and 1.0 indicates all tasks are executed at their respective $C_{i}^{hi}$ levels. The task response times were measured by the high resolution counter/timer used as part of the timestamp mechanism by WindRiver’s System Viewer application. Table 5-3 was used to represent their respective average case and worst case execution times for each task in the set.
Figure 5-10: HRT Task Set Response Time Average

Figure 5-11: SRT Task Set Response Time Average

Figure 5-11 represents the measured response times of the HRT task set. To represent each individual task would create an overly crowded graph so the individual task response times were normalized and then averaged over the whole task set. Specifically each task response time was recorded then compared to the respective task’s estimated response time. Let the actual task
response time be defined as \( R_i^{\text{Act}} \), the estimated lower bound response time is \( R_i^{\text{Lo}} \), the upper bound response times is \( R_i^{\text{Hi}} \) so that the averaged response time difference is defined as:

\[
\Delta t = \frac{(R_i^{\text{Hi}} - R_i^{\text{Lo}}) - R_i^{\text{Act}}}{(R_i^{\text{Hi}} - R_i^{\text{Lo}})}
\]

then the total task set response time average is defined as the average of all \( \Delta t \) for the HRT task set. What this means is a value of 0.0 indicates the measured task response times were at or near their respective \( R_i^{\text{Lo}} \) values and a value of 1.0 indicates \( R_i^{\text{Hi}} \) values. A value greater than 1.0 signifies that one or more tasks exceeded their deadline. Notice that for HSP the response time difference hover around 1.0 this is because the local scheduler does not allow other HRT tasks to execute before a higher priority task \( C_i^{\text{Hi}} \) execution time. Therefore, before the system starts to become overloaded around 0.6 the response times for both the partitioned method (RM-P) and the non-partitioned method (RM-G) outperform those of HSP. Recall, this is an acceptable situation because with HRT tasks we are less concerned about response times as we are with HRT timing constraints. Note, that at times 0.6 to 0.7 both RM-P and RM-G methods start to exceed 1.0 which indicates that tasks in the HRT set are beginning to experience deadline misses while with HSP no HRT tasks experience deadline misses.

The SRT task set performance is illustrated in Figure 5-12. Notice that early on before the system becomes overloaded from 0.0 to 0.4 HSP clearly outperforms both the RM-P and RM-G methods. This is because the HSP is able to take advantage of the slack generated by the HRT task set. Once the system starts to become overloaded at 0.5 HSP starts to converge to RM-G because there is no longer any available slack time. Both the RM-G and the HSP methods outperform RM-P because they are allowed to migrate across the cores in the subsystem.
In this chapter, we present a new adaptive scheduling approach for hierarchical scheduled systems. Currently the parameters for a server in a HSF are only determined statically, it would be beneficial if those parameters could be determined dynamically to more effectively react to environmental changes. In order to close this gap we need to extend hierarchical scheduling to provide a resource based allocation scheme that can adapt to allocation changes at run-time.

Our method utilizes a fuzzy logic based feedback scheduler to react to environmental changes within the application. The primary goal is to provide a scheduling mechanism that can adapt to overload conditions but still present a level of service while enforcing the temporal isolation between independent applications. The scheduler then considers this level of service to make scheduling decisions based upon a task’s service requirements, such as criticality or timeliness. Implemented in VxWorks on a uniprocessor-based platform results show that our adaptive approach provides significant advantages, during overload conditions, over traditional fixed-priority scheduling schemes. How adaptive hierarchical scheduling can be applied to multi-core based platforms is also examined with the actual simulations and implementation reserved for future work.

6.1 Introduction to Fuzzy Based Adaptive Hierarchical Scheduling for Real-Time Tasks

Current embedded systems are becoming considerably more complicated and expected to handle increasingly diverse applications. No longer are they considered special-purpose computing environments but are evolving into more general-purpose type platforms in terms of
their processing and workload requirements. These increasingly diverse applications present new challenges for traditional real-time scheduling mechanisms in that applications can have conflicting objectives. For example, one application may be more concerned with screen update response as opposed to whether a single update is missed. While a mission critical application, such as a navigational task, cannot afford to miss even a single update.

The problem is that traditional real-time scheduling mechanisms do not map well to these diverse types of applications specifically during a processing fault or during periods of computational overload. Faults could occur from longer than unexpected task execution time or an error in the program which could lead to the starvation for all lower-priority tasks. An overload could occur as the result of too many tasks being admitted into the system resulting into what is known as the “domino effect” where all tasks except the newly admitted one miss their deadlines.

The challenge is that many embedded systems are expected to perform continuous operations in potentially harsh environments and execute at least a subset of critical operations during a fault or overload condition. In order to enforce these strict timing constraints required by critical functions during a fault condition a form of temporal isolation is needed so that corresponding timing requirements are respected. During an overload event the system needs to be able to dynamically adapt to the current load so that system performance can degrade gracefully.

As a solution to these challenges our work utilizes hierarchical scheduling to provide the temporal isolation for real-time tasks to guarantee their timing constraints. The hierarchical scheduling framework (HSF) originally proposed by researchers [6] is a component based technique for scheduling complex real-time systems. The initial idea in applying this approach is
that relatively simple components can be used to create larger and more complex systems. In this way, the timing constraints of the individual components can be verified, a type of divide and conquer approach. Therefore, by extending this framework to our work we can then schedule each application (i.e. component) such that their timing constraints are satisfied. However, the current limitation with a traditional HSF-based approach is that the scheduling parameters for each component are assigned statically. Unfortunately, in a dynamic system the resource demand for each component can vary significantly especially during periods of overload. In this work we provide an adaptive mechanism where the component parameters can adapt to environmental changes in the system. In this way, the system can degrade gracefully in the presence of computational overload while still maintaining a level of serviceability for critical applications.

For this work we apply a novel approach where the component parameters adapt based upon a value-based heuristic instead of a deadline based policy. This value-based approach is applied because authors in [41] have presented the limitations of a deadline based model for real-time scheduling and have concluded that a value-based approach can more accurately represent the cost or benefit of meeting or missing a deadline. The challenge is in assigning this value metric because in the event of an overload we want to degrade the performance gracefully by ensuring that tasks are provided at least some minimum level of service. Therefore, during an overload when the current schedule is unfeasible we want the scheduler to schedule tasks according to some intelligent heuristic. Some possible heuristics would include scheduling the most important tasks first while still maintaining some level of timeliness for the less important tasks. Our approach is to utilize a heuristic function for guiding the scheduling decisions in a complicated situation where multiple factors may need to be considered such as deadlines, task criticality or task response times.
In this chapter we present a new adaptive hierarchical scheduler for real-time systems (AHS-RT) that provides timing guarantees for critical tasks and a minimum level of service for non-critical tasks during overload conditions. Our approach is to utilize fuzzy logic for the guidance mechanisms because they prove to be easier to express, comprehend and modify than other heuristic functions.

### 6.1.1 Hierarchical Scheduling Framework

Hierarchical scheduling provides a framework for scheduling multiple real-time applications on a single processor which is modeled as a system S. Each system may consist of multiple applications (subsystem $S_i$) such that $S_i \in S$. Each subsystem consists of a number of real-time tasks. Each subsystem is associated with a periodic server which provides the temporal isolation between subsystems. The execution of tasks is performed using a two-level hierarchical scheduling policy: global and local. The global scheduling policy determines which subsystem has access to the processor while the local scheduling policy determines which task should actually execute (Figure 6-1).

![Figure 6-1: AHS-RT Architecture](image-url)
6.1.2 Task Model

We consider a task set $\Gamma_s = \{\tau_1, \tau_2, ..., \tau_n\}$, such that each task $\tau_i$ is defined as $(T_i, C_i, D_i, L_i)$ where $T_i$ is defined as the task period, $C_i$ denotes the task worst case execution time (WCET), $D_i$ is the relative deadline and $L_i$ represents the task criticality value. It is assumed that each task $\tau_i$ is a constrained task such that $C_i \leq D_i \leq T_i$. The criticality value $L_i$ represents the importance or weight of the task as it relates to other tasks in the set. The criticality value along with the deadline and period are used by the fuzzy inference engine to make scheduling decisions by the local scheduler.

6.1.3 Subsystem Model

Each subsystem consists of a task set $\Gamma_{si}$ such that $S_i \leftarrow \Gamma_{si}$. The subsystem is modeled as a periodic task so a subsystem can be scheduled in a similar way as a simple real-time periodic task. The subsystem is defined as $S_i = (P_i, Q_i, L_i)$ where $P_i$ represents the subsystem period, $Q_i$ represents the subsystem budget and $L_i$ represents the subsystem criticality level. Similar to the task model the service value $L_i$ is used to make scheduling decisions at the subsystem level. Note that during overload conditions the subsystem with the highest criticality level is granted its full budget at the possible expense of lower criticality subsystems.

6.1.3.1 Periodic Server

The virtual server is invoked with the corresponding subsystem period $P_i$. If there are any ready tasks within the subsystem then they execute until they complete or the server’s budget $Q_i$ is exhausted. If there are no ready tasks to execute or no higher priority subsystem needs to utilize some of the server’s budget during an overload condition then the capacity is idled away as if a background task were running. After a server’s budget is exhausted the server suspends
the execution of the subsystem until the capacity is replenished at the start of the next period. For this work we choose a periodic server as the fixed priority server algorithm, in part because the simpler design has less overhead but also because authors in [42] have shown it to dominate other fixed-priority server algorithms.

### 6.2 Related Work in Adaptive Hierarchical Scheduled Systems

Hierarchical scheduling framework (HSF) was initially proposed by researchers [4][43][44] as a means to reduce the scheduling complexity for open source embedded systems. Resource partitioning [7] was introduced as a general technique for limiting the effects of overruns in tasks with variable execution times. This resource reservation technique can then be applied by hierarchical schedulers to provide the temporal isolation between subsystems for more predictable behavior, improved reusability and composability. However, the current limitation with HSF is that in order to determine the resource reservations all tasks parameters must be known a priori and fixed during run-time. The problem is that accurate task information may not be known or hard to derive at run-time. Additionally, in order to account for overload conditions the system may need to be over-engineered which could lead to significant under utilization during nominal load periods.

In [38] [45] [46] authors proposed a feedback mechanism to account for the dynamic behavior when the task parameters may not be fully known. The approach was for the scheduler to maximize the CPU utilization, avoid system overload and distribute the computing resource evenly among tasks. By incorporating feedback the scheduler reacts to changes in the workload then tries to keep the overall utilization as close as possible to a desired set point typically using a type of control mechanism, such as a proportional integral derivative (PID) controller. Related work [47] [48] adjusts the resource allocation on-line based upon a quality-of-service (QoS)
scheme where a certain level of service is provided in cases overload. However, the primary objective of this approach is control performance and not necessarily minimizing the number of missed deadlines. It is for this reason that control tasks typically fall into the category of firm instead of soft or hard real-time tasks.

Authors in [49] took a slightly different approach in that they based their scheduler on a benefit based model. Their approach was to schedule the tasks using a traditional deadline based scheduling policy until a potential fault was detected and before an overload condition could occur. After a fault is detected the scheduler switches to a benefit based scheduler that considers task importance, system state and timeliness to schedule tasks. Authors in [50] also took a similar approach in adaptive scheduling except they manipulated the task period of other tasks to achieve the desired level of performance.

Other research [51] [52] [53] treated the uncertainty of varying execution times as a multi-criteria optimization problem then applied fuzzy logic to derive a feasible schedule. Their approach was to treat various task parameters, such as deadline, start time or execution time, as inputs to the fuzzy scheduler then perform fuzzy analysis to assign a task priority value. Additional work [54] utilized fuzzy logic as a means for tuning a feedback controller to provide optimal resource utilization through task period re-adjustment.

Recent work [54] extended hierarchical scheduling to provide an adaptive hierarchical framework for managing overruns in tasks with varying execution times. Their approach was to utilize a feedback control mechanism for adapting the resource allocation by adjusting the amount of budget assigned to a subsystem. By adjusting the budgets at run-time the framework can better adapt to changes in the workload.
Our approach in AHS-RT is similar to the work in [54] in that we also utilize hierarchical scheduling for determinism and temporal isolation. However, AHS-RT differs in how the local scheduling and global scheduling is performed. Local scheduling is based upon a fuzzy scheduler which is more adept at making scheduling decisions when the task parameters are vague. Research by authors in [52] demonstrated that fuzzy logic based approaches outperform traditional deadline based policies such as earliest deadline first (EDF). In AHS-RT global scheduling also uses a feedback controller but the controller is based upon a fuzzy logic heuristic instead of a PID controller. Because fuzzy logic can better tolerate imprecision thereby providing improved run-time flexibility.

6.3 AHS-RT Architecture

This section describes the overall architecture (see Figure 6-1) of the AHS-RT scheduling framework which consists of a two-level hierarchical scheduling framework. The root-level contains the global scheduler which manages how subsystems (i.e. applications) are allocated on the processor. While the node-level contains the local scheduler which manages how tasks are scheduled on the processor.

6.3.1 Global Scheduling

At run-time the global scheduler chooses the highest priority subsystem that has tasks ready to run. The priority is based upon the subsystem period \( P_i \) so the shorter the period the higher the subsystem priority. Therefore if the priority of \( S_j > S_i \) then \( S_j \) would be scheduled first with its full budget then \( S_i \) would be scheduled next with its full budget unless an overload condition is detected. In the event of an overload a higher criticality subsystem may request a budget change at the possible expense of a lower criticality subsystem which may or may not be a lower priority subsystem.
The logical approach may be to re-assign budgets based upon subsystem priority. However, during an overload event studies have shown [55] that a value-based approach offers considerable advantages over traditional deadline-based approaches. For this reason, during an overload event the global scheduler of AHS-RT temporarily switches from a deadline-based scheduling policy to a value-based scheduling policy. Instead of the highest priority subsystem receiving their full budget the subsystem with the highest criticality level \( L_i \) will receive their entire budget. Therefore, the global scheduler redistributes budgets based upon the criticality level which means lower criticality subsystems yield their budgets to higher criticality subsystems. This greedy approach can lead to starvation, even for some high priority subsystems, but this is acceptable in that during overload conditions the highest criticality subsystems are considered superior to lower criticality subsystems.

6.3.1.1 Detecting Overloads

An overload condition is based upon the overall subsystem utilization which is defined as:

\[
U_T = \sum_{S \in S} \frac{Q_S}{T_S}
\]

and because we are using RM then an overload condition is determined by \( U_T \leq m(2^{1/m} - 1) \), where \( m \) is the number of subsystems. An overload can occur because a subsystem requests a budget change in order to adapt to a fault or missed deadline within a task of an individual application. A budget change does not necessarily mean that the system is overloaded just that there is the potential for an overload condition to exist. Consider some unallocated system utilization denoted as \( U'_T \) such that \( U_T + U'_T \leq m\left(2^{1/m} - 1\right) \), and then this extra utilization could be temporarily reallocated to the subsystem requesting the additional
budget. However, if there are not sufficient resources to satisfy all the budget requirements then the system is considered overloaded which implies that a budget reallocation needs to be performed.

6.3.1.2 Budget Reallocation

After the full budget has been allocated to the highest criticality subsystem the lower criticality budgets needs to be re-dimensioned. The next lower criticality subsystems are then assigned budgets based upon the remaining utilization. The algorithm and description for budget dimensioning is provided below.

Algorithm 1 AHS-RT Budget Dimensioning

**Input:** The subsystem subset $S_{ij}$ with criticality levels less than the task set $T_s$

**Output:** A new budget parameter that maintains a schedulable system.

1: FOR each $S_i \in S_{ij}$
2: \[ Q_i = \text{NewBudget}(); \]
3: WHILE not Schedulable($S_p, S_i$) DO
4: \[ Q_i = \text{FindNewBudget}(S_p, Q_i) \]
5: END WHILE
6: END FOR

The budget dimensioning algorithm (Algorithm 1) works by iterating through all the subsystems $S_i$ in the subset $S_{s_i}$ of lower criticality subsystems. In line 2 the new budget is calculated based upon the remaining system utilization. A schedulability test (line 3) is then performed on the modified budget. If the modified budget renders the system unschedulable then
a new budget value is attempted based upon the previous failed value. The algorithm continues to reduce the budgets of lower criticality subsystems until a schedulable system is found.

6.3.2 Local Scheduling

The local scheduling of AHS-RT consists of two primary components; a fuzzy logic based scheduler and a fuzzy logic based feedback controller. The scheduler selects the task to execute on the processor derived from the fuzzy rules based approach to real-time scheduling. The feedback controller gathers system state information for subsystem budget management to maximize utilization and minimize missed deadlines.

6.3.2.1 Fuzzy Scheduler

At run-time the fuzzy scheduler selects the highest priority task that is ready for execution on the processor. The priority of the task is determined by several parameters: task deadline, task criticality and task starting time. The task deadline is the time before the task should be completed. The task criticality relates to the consequences of missing a deadline. The task starting time is the earliest time the task can be made ready for execution. These parameters are then fuzzified and represented as linguistic variables. Fuzzy rules are then applied to the linguistic variables to compute the service value. The linguistic values for the three parameters are defined as: task deadline (early, on-time, late), task criticality (hard, firm, soft) and task starting time (early, on-time, late). Fuzzy rules are then applied to create a fuzzy conclusion for computing the priority level. For example,

*IF* task has an early deadline, a hard criticality and an on-time starting time, *THEN* assign level ~7.
These fuzzy conclusions are then combined to produce a fuzzy variable that represents the criticality level of the task. The variable is then defuzzified to create a value that is compared to other tasks to determine which task should be scheduled next.

---

**Algorithm 2** AHS-RT Fuzzy Scheduler

<table>
<thead>
<tr>
<th>Input:</th>
<th>The task set $T_{ij}$ of subsystem $S_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>The “crisp” criticality level (priority) of each task based upon the system state.</td>
</tr>
</tbody>
</table>

1: WHILE Loop  
2: FOR each $\tau_i \in T_{ij}$  
3: $\text{Priority}_i = \text{FuzzyInferenceEngine}(D_i, T_i, L_i)$;  
4: END FOR  
5: $\tau_i = \text{FindHighestPriorityTask}(T_{ij})$;  
6: $\text{ScheduleTask}(\tau_i)$;  
7: $\text{UpdateSystemStates}(T_{ij})$;  
8: END LOOP

The fuzzy scheduler algorithm (Algorithm 2) iterates through all the tasks $\tau_i$ in the task set for a particular subsystem and for each task passes the deadline ($D_i$), criticality value ($L_i$) and starting time ($T_i$) into the fuzzy inference engine. The output from the inference function is a crisp value used to assign a priority to each task and stored in a priority array ($\text{Priority}_i$). The task with the highest priority is then executed until some scheduling event occurs (task completion, new task instance arrives or server budget exhaustion). The system status is then updated and if a task misses its deadline such as server budget exhaustion then the deadline miss is reported to the feedback controller which could trigger a budget reallocation across the system.

6.3.2.2 Fuzzy Feedback Controller

The feedback controller in AHS-RT is similar to the FC-UM algorithm [46] in that both the miss-ratio and utilization are monitored. The reference inputs for miss-ratio $M_s$ and unused budget $U_s$ are both set to zero. At each sampling instant the miss-ratio $M(k)$ and the unused
subsystem budget $U(k)$ are fed back into the controller. These values are then compared to their respective set points to determine the difference where $du(k)$ represents the utilization error and $dm(k)$ represents the miss-ratio error. The output from the fuzzy controller is the budget dimensioning factor $D_B$.

As part of a typical fuzzy controller (Figure 6-2) we need to specify meaningful linguistic values and membership functions for each input and output variable. For this work the input linguistic values are NB (negative big), NS (negative small), ZE (zero), PS (positive small) and PB (positive big). The output linguistic values are TN (tiny), VS (very small), SM (small), MD (medium), BG (big), VB (very big) and HG (huge).

![Figure 6-2: Internal structure of feedback controller](image)

The membership functions for the input and output linguistic variables $du(k)$, $dm(k)$ and $D_B$ are shown in Figure 6-3.
The rule base is also set up and defined in Table 6-1 which describes the possible rules in the fuzzy controller.

### Table 6-1: Rules base for fuzzy controller

<table>
<thead>
<tr>
<th>$D_B$</th>
<th>NB</th>
<th>NS</th>
<th>ZE</th>
<th>PS</th>
<th>PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NB$</td>
<td>HG</td>
<td>HG</td>
<td>HG</td>
<td>VB</td>
<td>BG</td>
</tr>
<tr>
<td>$NS$</td>
<td>HG</td>
<td>VB</td>
<td>BG</td>
<td>BG</td>
<td>MD</td>
</tr>
<tr>
<td>$Dm(k)$</td>
<td>ZE</td>
<td>VB</td>
<td>BG</td>
<td>MD</td>
<td>MD</td>
</tr>
<tr>
<td>$PS$</td>
<td>BG</td>
<td>MD</td>
<td>SM</td>
<td>SM</td>
<td>VS</td>
</tr>
<tr>
<td>$PB$</td>
<td>MD</td>
<td>SM</td>
<td>SM</td>
<td>VS</td>
<td>TN</td>
</tr>
</tbody>
</table>

6.4 Task Scheduling Example

To demonstrate AHS-RT we have provided an example scheduling scenario. Note for illustration purposes we are only considering one subsystem. So, the primary purpose of this example is present how the fuzzy scheduler manages tasks within the context of one subsystem.

### Table 6-2: Subsystem parameters

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>$P_s$</th>
<th>$Q_s$</th>
<th>$L_i$</th>
</tr>
</thead>
</table>

104
Consider the task set and subsystem listed in Tables 6-2 and 6-3. Table 6-4 describes the scheduling of tasks at the first scheduling event where tasks $\tau_1$, $\tau_2$ and $\tau_3$ all have the same initial
starting time but since $\tau_1$ has the nearest deadline it is assigned the highest priority by the fuzzy scheduler. Therefore, $\tau_1$ is allowed to execute until completion then at time $t3$ task $\tau_2$ executes until time $t5$ when the subsystem’s budget expires. At time $t10$ (see Table 6-5) the subsystem’s budget is replenished where the tasks can continue execution. At this time the fuzzy scheduler performs a re-ordering of task priorities to reflect the system state. Task $\tau_2$ is assigned the highest priority because the start time is the earliest and the deadline is the closest. Note that at time $t12$ task $\tau_2$ will complete execution but task $\tau_3$ will be scheduled over $\tau_1$ even though both tasks have the same relative deadline and start time. This is because $\tau_3$ was assigned a higher priority by the fuzzy controller because $\tau_3$ was defined to be a higher criticality task than $\tau_1$. Also note, due to subsystem budget exhaustion at time $t15$ task $\tau_1$ will miss its deadline which would trigger a budget reallocation request to the fuzzy controller for an increase in the subsystem budget. Finally, at time $t20$ (see Table 6-6) the scheduler re-orders the task priorities where once again $\tau_1$ will be assigned the highest priority.

### 6.5 Subsystem Reallocation Example

Consider the following subsystems with parameters presented in Table 6-7 which is used to illustrate how a subsystem is scheduled by AHS-RT.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>$P_s$</th>
<th>$Q_s$</th>
<th>$L_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>12</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$S_2$</td>
<td>15</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>$S_3$</td>
<td>20</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6-8: Subsystem Reallocation snapshot
<table>
<thead>
<tr>
<th>Subsystem</th>
<th>$P_s$</th>
<th>$Q_s$</th>
<th>$L_i$</th>
<th>$DU(k)$</th>
<th>$DM(k)$</th>
<th>$D_B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>12</td>
<td>3</td>
<td>10</td>
<td>-0.2</td>
<td>0.1</td>
<td>~4.0</td>
</tr>
<tr>
<td>$S_2$</td>
<td>15</td>
<td>3</td>
<td>8</td>
<td>0.0</td>
<td>0.0</td>
<td>~3.0</td>
</tr>
<tr>
<td>$S_3$</td>
<td>20</td>
<td>5</td>
<td>5</td>
<td>0.0</td>
<td>0.0</td>
<td>~5.0</td>
</tr>
</tbody>
</table>

Suppose that at some scheduling instant subsystem $S_1$ has a current budget $Q_1 = 3$ but due to a deadline miss the fuzzy controller recommends a budget increase to 4. Also suppose that $S_2$ and $S_3$ report no deadline misses or under utilization so the fuzzy controller recommends no budget changes. However, the increased budget of $S_1$ causes the schedulability test to fail because $U_T > m \left(2^{\frac{1}{m}} - 1\right)$ so now the criticality level $L_i$ is considered and since $S_1$ has the highest criticality level it is granted the full budget. After $Q_1 = 4$ the budget dimensioning algorithm is performed to redistribute the remaining utilization. Initially, the budgets for $S_2$ and $S_3$ will be $Q_2 = 3$ and $Q_3 = 0$ then a successful schedulability test will be performed. Next the budget for $S_3$ will be $Q_3 = 5$ but the schedulability test will fail. Since the system is no longer schedulable the budget for $S_3$ will now be $Q_3 = 4$. This time the system is schedulable so the adjusted budgets are reallocated to their respective subsystems.

6.6 Simulation

AHS-RT was implemented as part of the VxWorks 6.9 real-time operating system (RTOS). The simulations were executed using the SIMNT vxsim simulator. For evaluation purposes we ported the SNU Real-Time Benchmark Suite [36] to compare deadline misses. The SNU real-time benchmark suite contains small C programs used for worst-case execution time analysis. The programs are mostly numeric and DSP algorithms. In order to represent the
periodic task model of an embedded system a subset of the programs in the benchmark suite were chosen and assigned arbitrary task rates and criticality levels.

Illustrated in Figure 6-4 both AHS-RT and the VxWorks native fixed-priority preemptive scheduler (FPPS) are comparable as long as the load factor is below ~0.70 which corresponds with the lower bound for deadline based algorithms. Notice that AHS-RT experiences significantly fewer deadline misses than FPPS when the system starts to become overloaded (> ~0.70). Also note that AHS-RT manages overload more effectively in that it does not start to report deadline misses until closer to a ~0.80 load factor. Another important observation depicted in Figure 6-5 is that AHS-RT manages deadline misses much more effectively than FPPS for higher criticality tasks. Notice that AHS-RT does not even start to report deadline misses until close to a ~1.25 load factor while FPPS starts to report deadlines as early as ~0.85. Clearly, AHS-RT is the superior scheduling mechanism as compared to FPPS specifically during periods of overload.

![Figure 6-4: Number of Deadline Misses (All Tasks)](image-url)
Figure 6-5: Number of Deadline Misses (Highest Criticality Tasks)
This chapter describes the implementation of the ARAP and the RACPwP protocols to demonstrate the practicality of the approach as part of a ground-based command and control test set used for satellite telemetry processing. A hardware-in-the-loop (HWIL) simulator was used to provide the workload for our system. Two particular use cases were chosen; a PCM-based telemetry processing application (Figure 7-1) and a avionics telemetry processing application (Figure 7-2). These use cases were chosen because in both applications processing times can vary considerably depending upon the data rate how densely the telemetry frame is populated. In this way, we can use the HWIL simulator to generate/simulate transient overload conditions.

Figure 7-1: Telemetry Uplink System for Hardware-in-the-Loop Simulations
7.1 Hardware/Software Components

The hierarchical scheduler was implemented as part of WindRiver’s VxWorks 6.5 real-time operation system. The hardware for the embedded system consisted of a PowerPC MPC7455/MPC7457 single board computer. A separate special-purpose telemetry processor which is used for bit synchronization and decommutation of the pulse code modulated telemetry stream. The serial communication is provided by a FPGA-based RS422 PMC module. All devices were connected via a VME32 backplane.

The three main software components of the system include a hard real-time periodic task that performs the bit synchronization, frame decommutation and frame processing of a telemetry stream. The second component is a soft real-time task that provides health and status monitoring for the vehicle. The third component is an aperiodic task that transmits serial uplink commands.
to the vehicle. The hard real-time telemetry processing task and the soft real-time monitoring task share a global resource which is the decomutated telemetry buffer.

7.2 RACPwP Application Results

The data illustrated in Figures 7-2 and 7-3 represents the recorded task response times for the telemetry processing and health/status monitoring tasks. The response times were recorded at the completion of each periodic task.

As the figures confirm RACPwP provides improved task response times for hard real-time tasks and comparable response times for soft real-time tasks. Notice that for the periods of increased processing in Figure 7-2 with the hard real-time task using RACPwP outperformed PIP in terms of response times. (Note: the PIP protocol was chosen as a comparison because priority inheritance is the traditional resource allocation protocol used in real-time systems). This reduced response time leads to better determinism for hard real-time tasks because some of the issues like unbounded blocking that plague PIP are eliminated with RACPwP. Also notice in Figure 7-3 that on occasion RACPwP recorded comparable or even slightly better response times than PIP. This indicates that acceptable soft real-time response times do not have to be sacrificed at the expense of hard real-time determinism.
Figure 7-3: Hard Real-Time Response Times

Figure 7-4: Soft Real-Time Response Times

7.3 ARAP Application Results

The hard real-time telemetry processing task and the soft real-time monitoring task share a global resource which is the decommutated telemetry buffer. Similar to the simulation environment we used the HWIL simulator to model transient overloads between 0%-25%. Two
traditional resource synchronization protocols (PIP and FCFS) were used in the evaluation for comparison.

The results for the overall deadline miss rates were separated out based upon soft and real-time tasks. Soft real-time tasks were scheduled by the local subsystem scheduler which is scheduled by the global fixed priority scheduler of VxWorks. Hard real-time tasks were scheduled directly by the global VxWorks scheduler. Soft real-time tasks were modeled to allow their execution time to exceed their budget while the hard real-time tasks were designed to not exceed their pre-defined utilization budget.

![Figure 7-5: Hard Real-Time task miss rate](image)

Figure 7-5: Hard Real-Time task miss rate
Figure 7-5 and Figure 7-6 show the miss rates of the hard real-time tasks and soft real-time tasks respectively. The figures show the feedback mechanism represented as FPPS-ARAP in the graph was compared against the priority inheritance protocol (FPPS-PIP) and the first-come-first-server (FPPS-FCFS) protocol. Indicated by the data in Figure 7-5 when hard and soft real-time tasks share a resource and the soft real-time task has that resource locked it clearly affects the deadlines of the hard priority tasks. Notice how even when using priority inheritance a lower priority task can still cause a higher priority task to miss their deadline. The reason is that while PIP does solve the priority inversion problem it does not solve the problem of unbounded blocking.

As illustrated in Figure 7-6 ARAP (along with hierarchical scheduling) does provide the mechanism for eliminating the extended blocking by a lower priority task however, the soft real-time task could be affected causing increased missed deadlines for the soft real-time task. Notice that in Figure 7-6 ARAP reports the highest number of deadline misses for soft real-time tasks. The reason for this behavior is that during a transient overload the task may be denied access to
the resource and have to wait until the next budget replenishment period. As shown above using a feedback mechanism can directly benefit the determinism of a hard real-time at the expense of other soft real-time tasks that share the global resource.
8 CONCLUSIONS

For this work we considered making the next generation embedded systems more predictable and adaptive to dynamic computational changes. We focused our research on the adaptive scheduling of real-time embedded applications in uniprocessor and multi-core environments. We utilized the hierarchical scheduling framework to improve predictability and reliability and feedback based control mechanisms for efficient adaptability.

We first considered the problem of sharing global resources in hierarchical scheduled systems. Traditionally, HSF was designed for soft real-time applications, in part due to the problem of unbounded resource holding times between global resources. Our initial approach combined software transactional memory to provide better response times, than other state-of-the-art synchronization protocols, for higher priority tasks without drastically sacrificing soft real-time performance. The idea is that if a soft-real time task is holding an exclusive access resource long enough that is affects the timing of a hard real-time task then that task could be safely preempted. The motivation for this work stems for the aerospace industry where systems are routinely over engineered in the interest of real-time determinism. It is a common perception that an embedded system is considered “safe” at only 50% total utilization and considered unsafe above 75%.

However, the limitation with an approach that preempts the shared resource is that is can only be considered for resources that can be safely preempted such as shared or mapped memory devices. We also considered other traditional non-preemptable resources for sharing resources across subsystems. Therefore instead of using preemption we utilized feedback from the actual system to estimate future usage. This approach provided greater flexibility and allowed for the
system to adapt to changes better than other state-of-the-art synchronization protocols. By implementing this adaptive approach as part of an actual embedded system application we were able to effectively eliminate deadline misses for a critical high priority task.

The trend for next generation embedded computing is progressing towards multi-core platforms so we also evaluated the problem of how to assign and schedule hard real-time and soft real-time tasks in a symmetric multiprocessor environment to more effectively adapt to environmental changes. Those changes such as unexpected computational workload deviation were managed by hierarchical scheduling to provide the temporal isolation between tasks. The efficient assigning and scheduling of processors was accomplished by combining mixed-criticality and semi-partitioned scheduling. The result was a demonstrated improvement of response times for soft real-time tasks and schedulability guarantees for hard real-time tasks where no deadlines were missed during periods of overload. As further confirmation for the validity of this approach we also implemented this approach as part of the VxWorks RTOS.

In performing this research we recognized that the hierarchical scheduling framework requires fixed bandwidth allocation for the schedulers. While this framework provides for temporal isolation guarantees it can lead to resource under utilization due to conservative bandwidth estimates. For this reason, we presented an approach that can adapt the scheduler bandwidth to the changing computational workload on a uniprocessor system. This adaptability was accomplished by using a fuzzy based heuristic which has been proven to be more effective than traditional deadline based approaches especially during periods of overload. The results are a demonstrated reduction in deadline misses for all tasks during periods of overload as compared to traditional fixed priority based scheduling mechanisms. As further confirmation for the practicality for this approach we implemented it as part of the VxWorks RTOS.
We propose that we can build more efficient embedded systems by more effectively managing the tasks within that system. The traditional deadline based task model does not map as well to the next generation embedded platforms with increasingly diverse workloads that are expected to work continuously in solitary and harsh environments. For this reason, we present a new scheduling framework to adapt to these new challenges.
9 REFERENCES


10 APPENDIX

This chapter provides an overview of the enhancements that were made to VxWorks and RTSIM. The changes made to VxWorks were implemented in kernel mode through wrapping various kernel or system calls and utilizing any system-level hooks that are provided. RTSIM was modified in order to implement a periodic server and the adaptive resource allocation protocol (ARAP). The modifications were made to provide a test bed to evaluate the performance of ARAP. These modifications were provided to evaluate the performance of the RACwP protocol, ARAP protocol, hierarchical scheduling in multi-core processors and fuzzy logic based hierarchical scheduling.

10.1 VxWorks Implementation

This section illustrates the modifications and wrappers created in VxWorks to support the hierarchical scheduling mechanisms. The work is based upon the architecture presented in [27] and extended to also work in a SMP-based platform.

10.1.1 Local Scheduler Implementation

The native VxWorks scheduler can schedule tasks using either a preemptive priority based or a round-robin scheduling policy. In VxWorks 6.x and greater WindRiver introduced the concept of real-time processes (RTP) which more closely resemble processes in general purpose operating systems like Linux. Tasks in kernel mode or processes in RTP mode are scheduled in the same way. Processes are created with memory protection so kernel memory space, ISRs and direct hardware access are prohibited. Tasks that operate in kernel mode have full access to kernel resources and are not subject to the same limitations as processes in RTP mode.
We choose to implement HSF in kernel mode because the overhead in RTPs are prohibitive and HSF needs access to the kernel resources for task management. HSF was implemented on top of the native VxWorks scheduler as a type of extension or middleware that sits between the hierarchical scheduler and the VxWorks native scheduler. The VxWorks RTOS provides functions to extend the capability so various kernel mechanism can be customized to support HSP. For example, the scheduler can be extended with either a customized ready queue structure or to attach an interrupt handler that is executed at every clock tick.

The native VxWorks scheduler dispatches the highest priority task in the ready queue. Our approach utilizes the system call `tickAnnounceHookAdd()` that is invoked at every tick interrupt and called before the native scheduler accesses the ready queue to dispatch the highest priority task. The ready queue is then manipulated by resuming a task `taskResume()`, suspending a task `taskSuspend()` or setting/changing priorities `taskPrioritySet()`. The kernel’s tick counter is also utilized to read `tickGet()` and set `tickSet()` as a means to manage the notion of time when the tick interrupt ISR is invoked.

The primary function of the local scheduler is to arrange tasks in the ready queue at every period start, in effect extend the VxWorks scheduler to support periodic tasks. The local scheduler is implemented as part of a custom ISR that is attached with the `tickAnnounceAdd()` system call. The system call routines mentioned previously are then called to change the status of the task or to change task priorities. The native VxWorks scheduler is then invoked to perform the necessary functions (i.e. context switching) to dispatch the task on the appropriate processor. The pseudo code listed in Algorithm 1 below provides an overview of the local scheduler.
Algorithm 1 HSF Local scheduler algorithm

1: FOR each \( de_i \in DE_i \)
2: \( \text{IF } DEQ[i].\text{task} = \text{ready} \text{ THEN} \)
3: \( \text{logMsg(deadline_miss, } DEQ[i].\text{task}) \)
4: \( \text{END IF} \)
5: updateEventQueue(\( DEQ[i].\text{task} \))
6: \( \text{ENDFOR} \)
7: FOR each \( pe_i \in PE_i \)
8: insertReadyTask(\( PEQ[i].\text{task} \))
9: updateEventQueue(\( PEQ[i].\text{task} \))
10: \( \text{END FOR} \)
11: event = getNextEvent(\( DEQ, PEQ \))
12: expire = event – systemTime
13: setInterrupt(expire, localSchedIsr)
14: systemTime = event

The first step of the algorithm is to check if the task is still in the ready queue (lines 2-4) the then the deadline event queue (DEQ) is updated (line 5) to track the task deadlines. At each period start tasks are inserted into the ready queue (7-8). Tasks deadlines and periods are updated in the periodic event queue (PEQ). The next event is then updated by extracting the closet deadline/period from event queue (lines 11-12). The interrupt is set at the next event and the local system counter is updated (lines 12-14).

10.1.2 Global Scheduler Implementation

Global scheduling is used to implement the notion of servers in a hierarchical scheduled system. The global scheduler is responsible for managing all the events in the system which can include subsystem events, server events and server budget events. The global scheduler itself is a task in VxWorks with its own task control block (TCB) and task event queue. Figure 10-1 below
illustrates the implementation of the required data structures to support global scheduling in HSP for VxWorks.

As for HSF implementation in multi-core SMP based scheduler there are not a significant amount of changes required in the VxWorks defined structures the main difference is now a periodic server is assigned to each core.

Figure 10-1: HSF implementation in VxWorks (single processor)
Figure 10-2: HSF implementation in VxWorks (multi-core)
10.2 RTSIM Modifications

This section provides an architectural as well as a class overview of the changes that were made to RTSIM in order to support hierarchical scheduling and resource synchronization.

![RTSIM AbsTask Class Architectural Diagram](image)

Abstract Task Interface. This interface is common to all objects that will implement something similar to a task, i.e. anything that can be scheduled by a kernel. For example, both Task and Server implement this interface. Therefore, this interface defines all methods that are essential for a kernel to schedule a task.

Public Member Functions

\[
\begin{align*}
\text{virtual} & \quad \sim \text{AbsTask} () \\
\text{virtual void} & \quad \text{schedule} () = 0 \\
\text{virtual void} & \quad \text{deschedule} () = 0 \\
\text{virtual void} & \quad \text{activate} () = 0 \\
\text{virtual bool} & \quad \text{isActive} (\text{void}) = 0 \\
& \quad \text{returns true if the task is active} \\
\text{virtual bool} & \quad \text{isExecuting} (\text{void}) = 0 \\
& \quad \text{returns true if the task is executing}
\end{align*}
\]
virtual Tick getArrival (void)=0
get current arrival time of the job
virtual Tick getLastArrival (void)=0
get arrival time of the last executed job
virtual void setKernel (AbsKernel *)=0
set the kernel for this task
virtual void refreshExec (double oldSpeed, double newSpeed)=0
virtual Tick getMaxExecutionTime ()
returns the wcet for this task, if available! otherwise returns 0.

Constructor & Destructor Documentation
virtual RTSim::AbsTask::~AbsTask ( )
Virtual destructor. It avoids a warning for the presence of virtual functions.

Member Function Documentation
virtual void RTSim::AbsTask::activate ( )
Activates the task. Different from calling onArrival, because this should post an event, rather
than directly calling the function.

virtual void RTSim::AbsTask::deschedule ( )
Called when the task is preempted.

virtual void RTSim::AbsTask::refreshExec ( double oldSpeed, double newSpeed)
It refreshes the state of the executing task when a change of the CPU speed occurs.

virtual void RTSim::AbsTask::schedule ( )
Called when the task is scheduled to execute.

AbsRTTask Class Reference
Interface for a real-time task. In addition to the AbsTask interface, here we have also the
deadline.

Public Member Functions
virtual Tick getDeadline ()=0
virtual Tick getRelDline ()=0
virtual int getTaskNumber ()=0

**Member Function Documentation**

virtual Tick RTSim::AbsRTTask::getDeadline ()
returns the task's deadline

virtual Tick RTSim::AbsRTTask::getRelDline ()
returns the task's relative deadline

virtual int RTSim::AbsRTTask::getTaskNumber ()
returns the Task ID.

**Server Class Reference**

This class implements a generic aperiodic server.

**Public Member Functions**

Server (AbsTask *t, char *name)
virtual void schedule ()
virtual void deschedule ()
virtual void activate () throw (ServerExc)
virtual bool isActive ()
virtual Tick getArrival ()
virtual Tick getLastArrival ()
virtual void setKernel (AbsKernel *k)
virtual AbsKernel * getKernel ()
virtual Tick getDeadline ()
virtual Tick getRelDline ()
void setDeadline (Tick d)
virtual bool isExecuting ()
virtual void activate (AbsTask *)
virtual void suspend (AbsTask *)
virtual void dispatch ()
virtual CPU * getProcessor (AbsTask *)
virtual int getCPUIndex ()
virtual void onArrival (AbsTask *t)
virtual void onEnd (AbsTask *t)
ServerStatus getStatus ()
virtual void computeNewParms ()=0
virtual void updateParms ()=0
virtual void setServerEvt ()=0
virtual void print ()=0
void newRun ()
void endRun ()
virtual double getSpeed ()
virtual double setSpeed (double)
void refreshExec (double, double)
int getTaskNumber ()

Constructor & Destructor Documentation

RTSim::Server::Server ( AbsTask * t, char *name )

Parameters:
    t     served task
    name  server name

Member Function Documentation

virtual void RTSim::Server::activate ( AbsTask * )
Inherited from AbsKernel. Cannot be called here (throws an exception).

virtual void RTSim::Server::activate () throw (ServerExc)
Inherited from AbsTask. Cannot be called here (throws an exception).

virtual void RTSim::Server::computeNewParms ()
This function is called when a new instance of the served task arrives, so there is the need to
compute new parameters for the server.

virtual void RTSim::Server::deschedule ()
Inherited from AbsTask. It is implemented here

virtual void RTSim::Server::dispatch ()
Inherited from AbsKernel. Cannot be called here (throws an exception).

virtual Tick RTSim::Server::getArrival ()
Inherited from AbsTask. Returns the arrival time of the current instance
virtual Tick RTSim::Server::getDeadline ()
Inherited from AbsTask. Returns the current absolute deadline

virtual AbsKernel* RTSim::Server::getKernel ()
Inherited from AbsKernel. Returns the kernel for this server

virtual Tick RTSim::Server::getLastArrival ()
Inherited from AbsTask. Returns the arrival time of the previous instance

virtual CPU* RTSim::Server::getProcessor (AbsTask * )
Inherited from AbsKernel. Calls the corresponding function of RTKernel

virtual Tick RTSim::Server::getRelDline ()
Inherited from AbsTask. Returns the current relative deadline (if any)

virtual double RTSim::Server::getSpeed ()
Function inherited from AbsKernel. It returns the current speed of the CPU.

int RTSim::Server::getTaskNumber ()
returns the Task ID.

virtual bool RTSim::Server::isActive ()
Tell if the server is active (whatever it means). It can be redefined in the derived classes.

virtual bool RTSim::Server::isExecuting (void )
returns true if the server is executing

virtual void RTSim::Server::onArrival (AbsTask * t )
Inherited from AbsKernel.

virtual void RTSim::Server::onEnd (AbsTask * t )
Inherited from AbsKernel.

void RTSim::Server::refreshExec (double, double )
Function inherited from AbsTask. It refreshes the state of the executing task when a change of the CPU speed occurs.

virtual void RTSim::Server::schedule ()
Inherited from AbsTask. It is implemented here
void RTSim::Server::setDeadline ( Tick d )
Sets the current relative deadline (if any)

virtual void RTSim::Server::setKernel ( AbsKernel * k )
Inherited from AbsTask. Set the kernel for this server

virtual void RTSim::Server::setServerEvt ()
This is called every time the server is scheduled to execute.

virtual double RTSim::Server::setSpeed ( double )
Function inherited from AbsKernel. It sets the speed of the CPU accordingly to the new system load, and returns the new speed.

virtual void RTSim::Server::suspend ( AbsTask * )
Inherited from AbsKernel. Cannot be called here (throws an exception).

virtual void RTSim::Server::updateParms ()
This function is called when the task yields the processor, because it is finished, or it has been de-scheduled.

CapServer Class Reference
This is the base class for capacity-based servers. We define events for this class and a new handler function.

Public Member Functions
CapServer (Task *t, char *name="")
~CapServer ()
virtual void onBudgetExhausted ()=0
void setTrace (Trace *t)
void newRun ()
void endRun ()

Protected Attributes
DlineMissEvt _missEvt
DlinePostEvt _dlineEvt
BandExEvt _bandExEvt

Member Function Documentation
virtual void RTSim::CapServer::onBudgetExhausted ()
This function is invoked every time the _bandExEvt is triggered
CBSServer Class Reference
Inherited from AbsTask. It is implemented here

Public Member Functions
CBSFTServer (Tick Q, Tick p, Tick exe, RTScheduler *s, const char *name="")
virtual void onJobArrival (AbsTask *t) throw (CBSFTExc)
virtual void onJobEnd (AbsTask *t)
virtual void onJobBandEx (AbsTask *t) throw (CBSFTExc)
virtual void schedule ()
void setCash (Cash *c)
void setFault (Fault *f)
virtual void newRun () throw (CBSFTExc)
virtual void endRun () throw (CBSFTExc)

BandRpEvt Class Reference
Inherited from CapServer. This class is used by the periodic server to report a budget expired event to the kernel.

Public Member Functions
BandRpEvt(PServer *s)
virtual void doit();
PServer* getServer()

PServer Class Reference
Inherited from Server. This class is used to implement a periodic server.

Public Member Functions
PServer(Tick budget, Tick period, Task *t, char *name = "");
~PServer();
virtual void computeNewParms()
virtual void updateParms()
virtual void setServerEvt()    
virtual void onBudgetExhausted() 
virtual void onBudgetReplenish() 
virtual void newRun()    
virtual void endRun()    
Tick getRemBudget()

Member Function Documentation
virtual void computeNewParms()
This function is called when a new instance of the served task arrives, so there is the need to compute new parameters for the server. It is called only by Server::onArrival()

virtual void updateParms()
This function is called when the task yields the processor, because it is finished, or it has been descheduled.

virtual void setServerEvt()
This is called every time the server is scheduled to execute.

virtual void onBudgetExhausted()
This function is called when the server budget has been exhausted

virtual void onBudgetReplenish()
This function is called when the server budget has been replenished

virtual void newRun()
This function is called for each instance of a new server

virtual void endRun()
This function end is called for each de-instantiation of a server.

Tick getRemBudget()
This function is called to retrieve the remaining server budget
ResManager Class Reference

The generic resource manager. A specific resource manager should be derived from this class.

**Public Member Functions**

- `ResManager (const char *n="")`
- `virtual void addResource (char *name="", int n=1)`
- `virtual void addResource (const string &name, int n=1)`
- `void setKernel (AbsKernel *k)`
- `bool request (AbsTask *t, const string &name, int n=1)`
- `void release (AbsTask *t, const string &name, int n=1)`
- `virtual void addUser (AbsTask *t, const string &name, int n=1)=0`

**Protected Member Functions**

- `virtual bool request (AbsTask *t, Resource *r, int n=1)=0`
- `virtual void release (AbsTask *t, Resource *r, int n=1)=0`

**Protected Attributes**

- `AbsKernel * _kernel`
- `set< Resource * > _res`

**Member Function Documentation**
virtual void RTSim::ResManager::addResource (const string & name, int n = 1)  
Adds the resource to the set of resources managed by the Resource Manager.

virtual void RTSim::ResManager::addUser (AbsTask * t, const string &name, int n = 1)  
Function called to specify that task t uses the resource called name. This function is not necessary in simple resource managers, like FCFSResManager. It is useful for SRPManager, for computing the ceilings!

void RTSim::ResManager::release (AbsTask *t, const string &name, int n = 1)  
Function called by a task instr to perform the release of a specific resource. The consequence of this call could be the reactivation of one or more suspended tasks.

bool RTSim::ResManager::request (AbsTask * t, const string &name, int n = 1)  
Function called by a task instr (the WaitInstr) to perform an access request to a specific resource. That access could be granted or the task could be suspended. It returns true if the resource has been locked successfully; it returns false if the task has been blocked.

void RTSim::ResManager::setKernel (AbsKernel *k)  
Set the corresponding kernel.

SRPManager Class Reference
This class implements the stack resource policy (SRP) resource manager

Public Member Functions

SRPManager (const char *n="")
void setScheduler (RTScheduler *s)
void newRun ()
void endRun ()

Protected Member Functions

virtual bool request (AbsTask *t, Resource *r, int n=1)
virtual void release (AbsTask *t, Resource *r, int n=1)
void addUser (AbsTask *, const string &, int=1)

Member Function Documentation

void RTSim::SRPManager::addUser (AbsTask *, const string &, int = 1)  
Function called to specify that task t uses the resource called name.

void RTSim::SRPManager::newRun ()
void RTSim::SRPManager::setScheduler (RTScheduler * s)  
Sets the scheduler for this resource manager
HSRPManger Class Reference
This class inherits the SRPResManager class and implements the hierarchical stack resource policy (HSRP) resource manager

Public Member Functions
HSRPManger (const char *n="")
void setLocalScheduler (RTScheduler *s)
void setGlobalScheduler (RTScheduler *s)
void newRun ()
void endRun ()

Protected Member Functions
virtual bool request (AbsTask *t, Resource *r, int n=1)
virtual void release (AbsTask *t, Resource *r, int n=1)
void addUser (AbsTask *, const string &, int=1)

Member Function Documentation
void RTSim::HSRPManger::addUser (AbsTask *,const string &, int = 1)
Function called to specify that task t uses the resource called name.

void RTSim::HSRPManger::newRun ()
void RTSim::HSRPManger::setLocalScheduler (RTScheduler * s )
Sets the scheduler for the local resource manager

void RTSim::HSRPManger::setGobalScheduler (RTScheduler * s )
Sets the scheduler for the global resource manager

SIRAPManger Class Reference
This class inherits the SRPResManager class and implements the SIRAP resource manager

Public Member Functions
SIRAPManger (const char *n="")
void setLocalScheduler (RTScheduler *s)
void setGlobalScheduler (RTScheduler *s)
void newRun ()
void endRun ()

Protected Member Functions
virtual bool request (AbsTask *t, Resource *r, int n=1)
virtual void release (AbsTask *t, Resource *r, int n=1)
void addUser (AbsTask *, const string &, int=1)
Tick getRemBudget(Server *s)

**Member Function Documentation**
void RTSim::SIRAPManager::addUser (AbsTask *t, Server *s, const string &, int = 1)
Function called to specify that task t uses the resource called name.

void RTSim::SIRAPManager::newRun ()
void RTSim::SIRAPManager::setLocalScheduler (RTScheduler * s )
Sets the scheduler for the local resource manager

void RTSim::SIRAPManager::setGlobalScheduler (RTScheduler * s )
Sets the scheduler for the global resource manager

Tick getRemBudget(Server *s)
The function gets the servers remaining budget to determine if the resource can be locked.

**BOManager Class Reference**
This class inherits the SRPResManager class and implements the budget overrun (BO) resource manager

**Public Member Functions**
BOManager (const char *n="")
void newRun ()
void endRun ()

**Protected Member Functions**
virtual bool request (AbsTask *t, Resource *r, int n=1)
virtual void release (AbsTask *t, Resource *r, int n=1)
void addUser (AbsTask *, const string &, int=1)

**Member Function Documentation**
void RTSim::BOManager::addUser (AbsTask *t, Server *s, const string &, int = 1)
Function called to specify that task t uses the resource called name.

void RTSim::BOManager::newRun ()

**ARAPManager Class Reference**
This class inherits the SRPResManager class and implements the ARAP resource manager
**Public Member Functions**

ARAPManager (const char *n="")
void setLocalScheduler (RTScheduler *s)
void setGlobalScheduler (RTScheduler *s)
void newRun ()
void endRun ()

**Protected Member Functions**

virtual bool request (AbsTask *t, Resource *r, int n=1)
virtual void release (AbsTask *t, Resource *r, int n=1)
void addUser (AbsTask *, const string &, int=1)
Tick getRemBudget(Server *s)
Tick estimateBudget(AbsTask *t)

**Member Function Documentation**

void RTSim::ARAPManager::addUser (AbsTask *t, Server *s, const string &, int = 1)
Function called to specify that task t uses the resource called name.

void RTSim::ARAPManager::newRun ()
void RTSim::ARAPManager::setLocalScheduler (RTScheduler *s )
Sets the scheduler for the local resource manager
void RTSim::SIRAPManager::setGlobalScheduler (RTScheduler *s )
Sets the scheduler for the global resource manager

Tick getRemBudget(Server *s)
The function gets the servers remaining budget to determine if the resource can be locked.

Tick estimateBudget(Server *s)
This function estimates the budget requirements to determine if the resource can be locked.