The Dissertation Committee for Xiang Feng
certifies that this is the approved version of the following dissertation:

Design of Real-Time Virtual Resource Architecture for Large-Scale Embedded Systems

Committee:

Aloysius K. Mok, Supervisor

Don Batory

James C. Browne

Mohamed G. Gouda

Tei-Wei Kuo

Yongguang Zhang

Design of Real-Time Virtual Resource Architecture for Large-Scale Embedded Systems

by

Xiang Feng, BS, ME, MS

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2004

Dedicated to Yunda,

with a real time virtual resource

of 100% rate and zero partition delay.

Acknowledgments

Getting a doctorate is a long-cherished dream of mine. Numerous people have encouraged me and helped me to turn this dream into reality. Six years in graduate school has greatly improved my research skills and profoundly impacted my personality. I have been blessed to have an extensive network of family, friends, and professional colleagues to support me throughout this journey.

My gratitude to my advisor, Doctor Aloysius Mok, is beyond words. He has infused me with enthusiasm for research, guided me through different research stages, left me enough freedom to do things the way I thought they should be done, trusted me when I doubted myself, and has always availed himself to me no matter what kind of problems I have. I just cannot imagine a better advisor.

I also thank Doctor Don Batory, Doctor James C. Browne, Doctor Mohamed G. Gouda, Doctor Tei-Wei Kuo and Doctor Yongguang Zhang who served on my dissertation committee and provided me with invaluable suggestions and comments.

Thanks also go to past and present members of the Real-Time Systems Research Group at the University of Texas at Austin. In particular, I would like to thank Doctor Deji Chen for helping me start my research. I would like to thank Zhengting He for assisting me with the implementation of RTVR into Linux. Weirong Wang, Yanbin Liu, Ruiqi Hu, Weijiang Yu, Chan-Gun Lee, Honguk Woo and Wing-Chi Poon, thank you for your helpful discussions, for showing me there is more to life than graduate school, and for being great friends of mine. Because of you — the real time folks — I had a really good time.

I cannot imagine being in this position without the unwavering support and love from my parents. They taught me the value of education, ensured that I had every opportunity, encouraged me to make the most of each, and showed their pride in all my accomplishments.

Most of all, and no surprise, I thank my wife, Yunda. She always encourages me to do what I love, even when it means sacrifice on her side. This dissertation, hence, is dedicated to her. I also thank my baby daughter Anita (Neenee). Her toothless smile always brightens up my lonely days in front of the computer.

XIANG FENG

The University of Texas at Austin
May 2004

Design of Real-Time Virtual Resource Architecture for Large-Scale Embedded Systems

Publication No. ______________________

Xiang Feng, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Aloysius K. Mok

Embedded computers have become pervasive and complex. Every microwave oven has one. The Volvo S80 has more than fifty. A Boeing 777-300 has hundreds. Meanwhile, more and more embedded systems are inter-connected to perform sophisticated functions on their hosts such as the air information management systems on modern aircrafts. However, due to real-time and fault-tolerance concerns, embedded systems are traditionally implemented on dedicated hardware. This approach entails at least three serious consequences: firstly, profligate and rigid usage of resources necessitated by the binding of subsystems to hardware platforms; secondly, significantly more difficult system integration because individually developed and tested systems are not guaranteed to work in combination; thirdly, the lack of higher-level system control because conceptually indivisible functions are isolated on the hardware level. These problems have caused not only financial loss such as unusable systems due to the high cost of unjustified resource redundancy and integration failures, but also the loss of human lives as exemplified by a number of fatal accidents induced by the third problem. It is critical that system engineers have a solid basis for addressing these fundamental design problems in large-scale real-time embedded systems. An ideal solution should achieve a complete separation of concerns so that: (1) each task group may be executed as if it had access to its own dedicated resource, (2) there is minimal interaction between the resource level scheduler and the task level scheduler, and (3) in case of hardware failure, task groups could easily migrate to other resources.

Towards this end, in this dissertation we introduce the notion of a Real-Time Virtual Resource (RTVR) which operates at a fraction of the rate of the shared physical resource and whose rate of operation varies with time but is bounded. Tasks within the same task group are scheduled by a task level scheduler that is specialized to the real-time requirements of the tasks in the group. The scheduling problems on both task level and resource level are analyzed.

We specifically investigate RTVRs on the integer domain. For the case of regular resource partitioning, we show that the utilization bounds of both fixed-priority scheduling and dynamic-priority scheduling remain unchanged from those for dedicated resources. We determine the utilization bounds for the more general case of irregular partitioning. In particular, both types of partitions can be efficiently constructed by exploiting compositionality properties vis-a-vis the regularity measure.

We further extend the applicability of the RTVR in several directions. First, we propose a hierarchical real-time virtual resource model that permits resource partitioning to be extended to multiple levels. Through this model, partitions on each level are scheduled as if they had access to a dedicated resource. Interference between neighboring partition levels is also minimized. Second, we apply RTVR to gang scheduling which is a popular scheduling technique used in parallel systems. We show that the clean isolation between resource-level and task-level scheduling makes RTVR an ideal candidate for implementing the gang scheduling solution in the real-time systems environment. Third, RTVRs in distributed environments are also discussed and end-to-end delay of a series of RTVRs is calculated. Finally, we investigate the resource locking issues in RTVR and present a resource server solution which has a highly efficient admission test. We also present an optimization scheme called Partition Coalition which is based on the server solution and which can substantially reduce the blocking time due to resource locking. These results provide a foundation for implementing RTVR on small-scale multiprocessor or processor cluster systems that are increasingly available.

Based on the previous theoretical framework, we implement RTVRs on the Linux 2.4 kernel. The first RTVR implementation uses a static resource level scheduler which can be applied to systems with predefined application task sets. The second implementation has a novel dynamic resource level scheduler under which task groups can join and leave dynamically. We further virtualize network devices. Several experiments are conducted to measure system performance in various aspects such as the effect of the scheduling quantum size, interrupt request response time and scheduling overhead. The experiments demonstrate that RTVRs can be efficiently implemented while satisfying their theoretical properties.

Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiv**

List of Figures **xv**

Chapter 1 Introduction **1**

1.1 Real-Time Embedded Systems 1

1.2 Related Work 6

1.2.1 Real Time Virtual Resource 6

1.2.2 Linux Implementation 9

1.3 Synopsis 11

Chapter 2 Real Time Virtual Resource Model **13**

2.1 Introduction 13

2.2 The Static Resource Partition Model 13

2.2.1 Definition 14

2.2.2 Fixed Priority Scheduling 15

2.2.3 Dynamic Priority Scheduling 18

2.2.4 Critical Partition 19

2.2.5 Least Supply Function 20
2.2.6 Critical Partition 22
2.2.7 Fixed Priority Scheduling 23
2.2.8 Dynamic Priority Scheduling 24
2.3 Bounded-Delay Resource Partition Model 25
2.4 Regularity-Based Resource Partition Model 32
2.4.1 Regularity 33
2.5 Task Level Scheduling 35
2.5.1 Regular Partition 35
2.5.2 Fixed Priority 35
2.5.3 Dynamic Priority 42
2.5.4 Irregular Partitions 45

Chapter 3 Resource Level Scheduling **47**

3.1 Introduction 47
3.2 Problem Description 47
3.3 Static Resource Level Scheduling 48
3.4 Dynamic Resource Level Scheduling 50
3.4.1 Quantum Based Scheduling Consideration 53
3.4.2 Two-Level Resource Scheduling 54
3.5 Resource Level Scheduling for Regularity-Based Partitions 57
3.5.1 Regular Partition 57
3.5.2 Irregular Partitions 58
3.5.3 Mixed Partitions 62

Chapter 4 Hierarchical Resource Level Scheduling **64**

4.1 Multi-level Partition Scheduling 64
4.2 Partition Scheduling Algorithm 66

4.3 Partition Scheduling with Quantum Size Requirements 69

Chapter 5 Distributed and Parallel RTVR **78**

5.1 Introduction 78

5.2 Parallel RTVRs 78

5.3 Parallel RTVRs 80

5.3.1 Solution Generalization 83

5.3.2 Admission Control 84

5.4 Distributed RTVRs 84

5.4.1 Process Migration 85

Chapter 6 Resource Locking in RTVR **86**

6.1 Introduction 86

6.1.1 Intra-Partition Locking 86

6.1.2 Inter-Partition Locking 87

6.1.3 Admission Control 89

6.1.4 Partition Coalition for Resource Locking 90

6.2 Discussion 92

6.2.1 Relation of Rate and Partition Delay 92

6.2.2 Scheduling Quantum Consideration 92

Chapter 7 Implementation on Linux Kernel **95**

7.1 Introduction 95

7.2 Original Linux Kernel Structure 97

7.3 RTVP Kernel Structure 98

7.4 Scheduling Implementation 100

7.5 Network Virtualization 102

7.6 Memory Consumption Concern 103

7.7 System Calls and Utilities 104

7.8 SVP & ISR 105
7.9 Experiments 105
7.9.1 Keyboard Response Time 106
7.9.2 Scheduling Overhead 107
7.9.3 Competing for a Single Resource 108
7.9.4 Competing for Multiple Resources 109
7.9.5 System Performance Tuning 110

Chapter 8 Conclusion 113
8.1 Summary 113
8.2 Future Research 115
8.2.1 Model Refinement 115
8.2.2 Scheduling 115
8.2.3 Implementation 116

Appendix A Notational Conventions and Acronyms 117

Appendix B Algorithm to Calculate Least Supply Function 120

Bibliography 123

Vita 136

List of Tables

3.1 CPU Partition Table Generation Example 50

B.1 Computing Table 121

List of Figures

1.1 Overview of Real Time Virtual Resource Structure 11

2.1 Timing Diagram of Example Partition Π_1 14
2.2 Example in Traditional Critical Instance Test 16
2.3 Computing Supply Function 22
2.4 Critical Instance Test 24
2.5 Instant Regularity at Time 4 of Partition Π_1 in Example 2.1 34

3.1 Worst Case Execution Scenario Case 1 51
3.2 Worst Case Execution Scenario Case 2 52
3.3 Two Level Scheduling Example 56

4.1 Quantum Size Limit 71

5.1 Parallel Task Structure 81
5.2 General Parallel Task Structure 84

6.1 Inter Partition Resource Locking Solution 87
6.2 Rate-Delay Relation 93

7.1 Real Time Virtual Resource Structure Overview 96
7.2 Linux Kernel State Transition Diagram 97

7.3 New Kernel Process Scheduling and State-Transition Diagram 99
7.4 New Linux Kernel State Transition Diagram 99
7.5 Scheduler State-Transition Diagram 101
7.6 Experiment (1)-(3) Results 107
7.7 Experiment (4) Results 110
7.8 Partition Rate versus Partition Delay 111

Chapter 1

Introduction

1.1 Real-Time Embedded Systems

The alarm clock blares at seven o'clock in the morning. Suddenly you jump out of the bed realizing that today begins your long-awaited vacation to Hawaii. You congratulate yourself, "Finally, I can get away from computers." Seconds later, you put your breakfast in the microwave oven. While packing, you kiss good-bye to your laptop, but you slip the cell phone into your pocket since you imagine it will be cool to call your co-workers from the beach. Minutes later, you are speeding to the airport. Hours later, flying above the pacific ocean, you smile smugly and assure yourself, "I am computer free".

But are you sure? The alarm clock that woke you up has one microprocessor. The microwave oven has another one. Your cell phone has two. A car like the Volvo S80 has more than fifty [5]. A Boeing 777-300 has hundreds [3]. In fact, more than 95% of microprocessors produced in the world are used in embedded systems today: embedded systems are simply everywhere.

Embedded computer systems differ from traditional computer systems in several aspects. First, as the name "embedded" implies, embedded systems need to be embedded into host devices to perform their functionalities. For example, an ABS system has to reside onboard a vehicle to fulfill its purpose by design. On the contrary, traditional systems do not require any host. Desktop computers can run stand-alone. Second, embedded systems are transparent to users. Most of their user interfaces are customized towards the requirements of their hosts. In the case of an ATM machine, clients would notice only its essential functions such as withdraw or deposit and the embedded system behind is easily overlooked. This also explains why people that use embedded systems every day do not even know of their existence. On the contrary, traditional computer systems like laptop PCs are immediately recognizable.

Among all features of embedded systems, the most prominent one is real-time. According to a survey conducted by Evans Data Corp [4], how well real-time is supported is the leading concern in choosing an embedded system platform. By definition, real-time systems have to meet timing constraints in addition to the logical correctness of the results. These systems are playing a more and more important role together with other technologies in people's daily lives. As computers become ubiquitous, people do not desire simply service alone, but also quality of service (QoS), which is usually expressed in the form of timing requirements. For instance, people are frequently annoyed with frame loss when they watch videos on the Internet. This situation poses a serious obstacle for online entertainment industries such as online gaming and video-on-demand.

Being real-time does not simply mean being fast. Instead, it is more concerned with predictability. Let us look at the Internet video case again. If the network transmits video data too fast so that the amount of data exceeds the buffer size in the end computer, these data have to be dropped and need to be retransmitted later. Therefore, being fast here might not help at all, conversely, it harms. What users really need is the consistency of the video quality. Then we extend the concept of real-time to a broader context, it can be concluded that virtually every computer system would need real-time since predictability is always desirable.

Recently real-time has become more difficult to achieve because more and more embedded systems are inter-connected to perform sophisticated functions on their hosts. Embedded systems are traditionally implemented on dedicated hardware. System integration is significantly more difficult for embedded systems across different hardware platforms because individually developed and tested systems are not guaranteed to work in combination.

Another serious obstacle to the implementation of real time in embedded systems is fault tolerance. Embedded systems are generally less tolerant of faults. For example, a fault of onboard air bag systems can be disastrous. A single fault of one application would affect not only itself but also others by causing them to be delayed and possibly to miss their deadlines. Therefore, a fault containment mechanism must be provided among those applications so that any fault would not be propagated to other applications, but rather be contained within the application that generated it.

Take for example the design of avionics systems. Traditionally, the control system of an aircraft is decomposed by functionality into distinct subsystems such as autopilot and auto throttle. A "federated" approach has been used to provide fault containment simply by dedicating a separate fault-tolerant computer to implement each of the subsystems. The only interface between subsystems is through sensor or control data exchange. On its face, this federated approach perfectly solves the fault containment problem. The narrow interface between subsystems in the federated approach makes a timing fault unlikely to propagate. However, the federated approach also introduces a serious problem. Those subsystems in the federated approach are in fact tightly coupled because some control variables are affected directly by one another due to the flight dynamics. Hence, minimizing the amount of data that is exchanged among those subsystems prohibits a higher level control of those subsystems. For example, whereas engine thrust is controlled by the auto-throttle and pitch angle by the autopilot, a change in either of them may cause the other to change significantly. The coupling of control variables renders simple functionalities such as "cruise speed control" far from autonomous.

A more recent development is a resource sharing scheme via partitioning known as Integrated Modular Avionics (IMA) [7, 134]. The purpose of IMA is to accomplish the same isolation requirement as the federated approach and to support higher level autonomous control and to maximize resource utilization. In IMA, a single computer system with internal replication is used as a common platform to run different subsystems. Inasmuch as they share the same platform, a single timing fault of a subsystem may cause other subsystems to miss their deadlines. Therefore, any implementation of IMA ensures fault isolation by *partitioning* in both time and space. The real-time virtual resource concept provides a way to resolve the temporal partitioning problem.

Notice that IMA is substantially different from the centralized time-shared mainframe system in several aspects. First, they have different goals. IMA aims to achieve fault containment and a higher integration level but mainframe systems mainly aim to reduce cost. Second, IMA is internally a distributed system whereas replicated resources are not bound to a specific function, but can be allocated dynamically as required: normal operations can continue as long as the total number of no-fault resources is sufficient to provide the required level of replication to each function.

In academia, until recent years, the study of real-time scheduling problems has been primarily concerned with allocating only dedicated resources to service a set of real-time applications. Since the first real-time system model was introduced by Liu and Layland in 1973 [111], there have been many variations proposed to model real-time systems, e.g., the sporadic model [119], and the pinwheel model [72]. The schedulability analysis of these models always assumes that the resource to be allocated is made available at a uniform rate and accessible exclusively by the tasks under consideration.

This assumption, obviously, no longer holds when the resources are shared. This environment is also called an *open system* [45] by real-time system researchers. In an open system, a physical resource is shared by different classes of applications, some hard-real-time, others soft-real-time or even non-real-time. Sharing is enforced by some kind of partitioning scheme that time-multiplexes the physical resource between the different application task groups with the goal that each application task group may be programmed as if it had dedicated access to a physical resource, i.e., without interference from other task groups due to resource sharing.

The general idea of resource sharing is to view each task group as accessing a virtual resource that operates at a fraction of a physical resource shared by the group, but the rate varies with time during execution. Ideally, a virtual resource should achieve a complete separation of concerns so that: (1) each application task group running on a virtual resource may be executed as if it had exclusive access to its own dedicated resource, and (2) there is minimal interaction between the resource-level scheduler and the application-task-level scheduler.

In this dissertation, we characterize the rate variation of each virtual resource by means of a delay bound Δ that specifies the maximum extra time the task group may have to wait in order to receive its fraction of the physical resource over any time interval starting from *any* point in time. This way, if we know that an event e will occur within x time units from another event e', assuming that the virtual resource operates at a uniform rate and event occurrence depends only on resource consumption (i.e., virtual time progresses uniformly), then e and e' will be apart by at most $x + \Delta$ time units in real time. If infinite time-slicing is possible, the delay bound is zero. In general, the delay bound of a virtual resource will be task-group-specific. The characterization of virtual resource rate variation by means of the delay bound will allow us to better deal with more general types of timing constraints such as jitter. We call those virtual resources, whose rate of operation variation is bounded, real-time virtual resources.

1.2 Related Work

1.2.1 Real Time Virtual Resource

In [116], the concept of a real-time virtual resource was introduced which operates at a fraction of the rate of the shared physical resource and whose rate of operation varies with time but is bounded. The focus of [116] is mainly on task-level schedulability with respect to given partitions. [115] is a companion to [116] in that we discussed in detail resource-level schedulability by making use of a specific approach to partitioning: regular and k-irregular partitions. The concept of regularity was first introduced by Shigero, Takashi and Kei in [138] where they proved the feasibility of task groups whose utilization is no bigger than the availability factor of the regular partition on which the task group executes. In [115], we made the more general observation that the schedulability bound results in the Liu and Layland paper still hold with regular partitions. We also generalized the regularity concept to include both temporal and supply regularities and gave a number of compositionality results.

The real-time virtual resource concept addresses some of the crucial issues of the open system environment. As such, this dissertation is related to the work pioneered by Deng and Liu [45]. In the open system environment, the admission test on a real-time task needs to be independent of any other task in the system, thus making a global schedulability analysis impossible. This concept was first discussed based on an EDF kernel scheduler and was later extended to a fixed priority scheduler[89]. It was further applied to parallel and distributed systems [87]. Because in an open real-time environment the parameters of real-time tasks are no longer required to be known *a priori*, efficient online scheduling algorithms are needed [13, 152]. Also needed are practical mechanisms to provide isolation among tasks. One interesting approach is to assign each task a server with certain parameters [15, 104]. However, the interaction between tasks and the higher-level scheduler may increase the unpredictability in task execution and hence make other requirements such as output jitter bound difficult to realize.

There are several differences between our approach and other approaches. A major one is that we minimize the interaction between the resource-level scheduler and the application-task-level scheduler to a simple interface. Unlike previous approaches, our resource-level scheduler does not require knowledge of the task-level deadlines or their derivatives in partition scheduling. Conversely, the task-level scheduler may need to know at most the regularity bound of the partition on which it executes. More importantly, the related delay bound of the partition allows the application task scheduler to determine not only compliance with deadline requirements but also event-separation types of constraints. If the application task groups are not all specified in one common task model such as Liu and Layland's periodic tasks, our partition model can still be used. We only need to figure out the schedulability conditions of the new system model on partitions. The effect of the partition scheduling on task group scheduling is characterized by the partition regularity in this dissertation.

Specifically, our definition of bounded delay differs from that of latency in other literature in two distinctive aspects. First, ours applies to all time intervals regardless of starts or ends, while others apply either to a specific interval or to intervals with a fixed starting point. Weighted Fair Queue (WFQ) [44] and Latency-

Rate (LR) Server [149] are two well-known examples. This flexibility with starts and ends is essential not only for the separation of concerns among different levels but also for meeting open system requirements. Second, our definition measures the service fluctuation from resources. In contrast, rate and latency in other literature are used to measure tasks in many instances.

Let us illustrate the differences with an interval from t to $t+l+\Delta$. According to the definition of RTVR, $l \times \alpha$ amount of service is guaranteed during that period. Suppose Δ amount of time is fully serviced from t to $t+\Delta$ and $l \times \alpha - \Delta$ is serviced from $t+\Delta$ to $t+l+\Delta$. In addition, a busy session (also called a busy interval) starts at time $t+\Delta$ and ends at $t+l+\Delta$. In this case, the amount of service from $t+\Delta$ to $t+l+\Delta$ does not meet the requirement of LR server. Therefore, a scheduler that satisfies RTVR does not necessarily satisfy the LR server. Similarly, during the interval from t to $t+l+\Delta$ if there is no busy interval and no time is serviced, the LR server, not the RTVR is satisfied.

Our designing concerns are also significantly different. The partition delay is introduced not because of sporadic tasks but because of scheduling quantum and other scheduling overhead. If infinite time-slicing were practical, the partition delay could be eliminated. Hence, we put more stress on resources rather than on tasks.

Having stated those general comments above, we now review the related approaches respectively.

First of all, Ferrari and Gupta proposed a similar idea of partitioning resource in [57, 63]. However, their discussion focuses only on network and they consider only EDF and other soft-real-time types of scheduling algorithms such as FIFO and priority queue scheduling.

Secondly, our work is also differentiated from Start-time Fair Queuing(SFQ) [62] and Fluctuation Constrained Servers (FCS) [94, 160]. Both SFQ and FCS have the similar notion of supply deviation for any time interval as in our work. However, while SFQ aims to minimize the delay to achieve near-optimal fairness, our work measures the delay as long as the delay can guarantee the schedulability of real-time tasks. Furthermore, SFQ depends on the number of threads (which are equivalent to partitions in our work) being scheduled; while the resource level delay in our work is specified by the partition request, thus providing a stronger guarantee. Another difference is that FCS is intended to be on a per stream basis while our work is on a per task group basis. FCS does not provide any real-time schedulability analysis if a group of tasks instead of one single task (stream) are running on FCS, while task level scheduling is a major problem that our work addresses.

Next, in Hierarchical Loadable Schedulers (HLS) [130] schedulers may be converted to one another by means of service guarantee. The goal of HLS is also quite different from that of RTVRs since HLS aims at constructing a hierarchy of schedulers while the goal of RTVRs is to construct a hierarchy of virtual resources.

Last but not least, Shin and Lee recently presented their work of a periodic resource model [140], whereas a resource allocation of X time units for every Y time units is guaranteed. Our approach differs from theirs mainly in that we do not require periodicity, thus providing a more general modelling for real-time.

1.2.2 Linux Implementation

There have been many initiatives to make Linux real-time. Two general categories of solutions have been proposed in academia and commercially implemented. The first one modifies the current Linux kernel or simply replaces it with a new one. The new kernel retains and implements the original kernel API with a full set of system calls. Examples using this approach include TimeSys (Linux/RK) [128], Red-Linux [157] and Qlinux[62]. The other approach creatively imposes another level of kernel (called sub-kernel) on the top of the existing Linux kernel. In this way, Linux is treated as the lowest priority task of the sub-kernel OS. RTLinux [2] and RTAI [1] are examples of this category.

While our approach falls into the first category, it differs from others in several aspects. First of all, we minimize the interaction between the resource level scheduler and the task level scheduler to a simple interface. Unlike other approaches, our resource level scheduler does not require knowledge of the task level deadlines or their derivatives in partition scheduling. On the subordinate level, our task level scheduler requires the knowledge of only the delay bound of the partition on which it executes. More importantly, the related delay bound of the partition allows the application task scheduler to determine not only compliance with deadline requirements, but also event-separation types of constraints. If the application task groups are not all specified in one common task model such as Liu and Layland periodic tasks, our partition model can still be used. In the following paragraphs, we discuss three exemplary approaches to make Linux real-time.

First, Linux/Resource Kernel (RK) [128] allows applications to specify only their resource demands and leaves the kernel to satisfy those demands to hidden resource management schemes. The resource demands are usually expressed in a form of computation time, period and deadline, which is distinctly different from ours.

Second, Red-Linux [157] aims to provide a general scheduling framework to integrate three paradigms, namely, priority-driven, time-driven and share-driven paradigms. In order to do that, Red-Linux identifies four scheduling attributes, i.e., priority, start-time, finish time and budget.

Finally, Constant Bandwidth Server (CBS) [105] uses a deadline postponing scheme to provide bandwidth isolation. A CBS Server is described by two parameters: budget and period. The bandwidth is calculated as budget over period and the period also serves as deadline.

1.3 Synopsis

A structure overview of virtual resources is shown in Figure 1.1. At the top of the figure a physical resource is partitioned into several virtual resources; then, each virtual resource can be partitioned recursively into several lower level virtual resources. Eventually, each virtual resource will be associated with one task group, which consists of one or more tasks. The mapping relation between resource and partitions is 1 to n, between the partition and task group is 1 to 1, between the task group to tasks is 1 to n. Two scheduling problems could be identified in this structure: one is to schedule tasks within a task group; the other one is to schedule virtual resources on a physical resource.

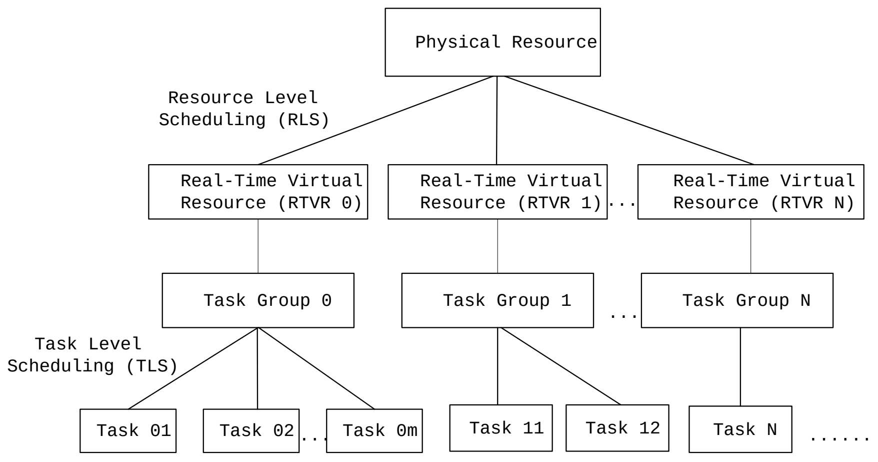

Physical Resource
Resource Level Scheduling (RLS)
Real-Time Virtual Resource (RTVR 0)
Real-Time Virtual Resource (RTVR 1)
Real-Time Virtual Resource (RTVR N)
Task Group 0
Task Group 1
Task Group N
Task Level Scheduling (TLS)
Task 01
Task 02
Task 0m
Task 11
Task 12
Task N

Figure 1.1: Overview of Real Time Virtual Resource Structure

Throughout the dissertation it is assumed that time values have the domain of the non-negative real numbers unless otherwise specified. Preemptive scheduling is assumed, i.e., a task executing on the shared resource can be interrupted at any instant in time, and its execution can be resumed later. Although a resource can be a processor, a communication bus, etc., we talk about a single processor as the resource to be shared.

Definition 1.1 *A task T is defined as (c, p), where c is the (worst case) execution time requirement, p is the period of the task.*

Even though we do not specify a per-period deadline explicitly, we define deadlines when they are relevant to the results in this dissertation.

Definition 1.2 *A task group τ is a collection of n tasks that are to be scheduled on a real-time virtual processor (a partition), $\tau = \{T_i = (c_i, p_i)\}_{i=1}^{n}$.*

We use the term task group to emphasize its difference from the term task set in that a task set is to be scheduled on a dedicated resource while a task group is scheduled on a partition of the shared physical resource.

The notational conventions that are used throughout this dissertation can be found in Appendix A.

The rest of this dissertation is organized as follows. Chapter 2 introduces three resource partition models: static partition model, bounded-delay partition model, and regularity-based partition model. Static Resource Partitions serves as the starting point of our discussion of resource partitioning. Bounded Delay Resource Partitions further generalize and characterize resource partitions. Regularity Based Resource Partitions fall in the domain of integer values and have distinctively interesting properties. The architecture of RTVR, which is based upon the bounded-delay partition model and the regularity-based partition model, consists of two levels of scheduling: task level scheduling and resource level scheduling. Task level scheduling is discussed when we introduce each partition model. Chapter 3 discusses resource level scheduling which is further expanded to a multi-level structure in Chapter 4. From Chapter 5 to Chapter 7, we expand the concept of RTVR into three closely related directions. RTVRs in parallel and distributed environments are studied in Chapter 5. Chapter 6 investigates resource locking problems in RTVRs. Implementation and experiments of RTVR on Linux are then discussed in Chapter 7. We finally draw our conclusion in Chapter 8.

Chapter 2

Real Time Virtual Resource Model

2.1 Introduction

In this chapter we formalize the concept of Real Time Virtual Resource. We propose three models of real time virtual resource that partition physical resources, i.e. Static Resource Partition, Bounded Delay Resource Partition and Regularity Based Resource Partition. Static Resource Partitions form the starting point of our discussion of resource partitioning. Bounded Delay Resource Partitions further generalize and characterize resource partitions. Regularity Based Resource Partitions fall in the domain of integer values and have distinctively interesting properties.

2.2 The Static Resource Partition Model

Intuitively, a static resource partition is simply a collection of time intervals during which the physical resource is made available to the task group being scheduled on the partition. In this section, we investigate the problem of scheduling task groups for a given partition whose time intervals are explicitly specified by a list. By making use of the technique of supply functions, we analyze both fixed and dynamic priority schedulers with respect to this partition model. The schedulability results are obtained based on the key idea of the critical partition. Finally we discuss the (second-level) scheduling problem of the partitions themselves.

2.2.1 Definition

Definition 2.1 *A resource partition* Π *is a tuple* (Γ, P)*, where* Γ *is an array of* N *time pairs* $\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}$ *that satisfies* $(0 \leq S_1 < E_1 < S_2 < E_2 < ... < S_N < E_N \leq P)$ *for some* $N \geq 1$*, and* P *is the partition period. The physical resource is available to a task group executing on this partition only during time intervals* $(S_i + j \times P, E_i + j \times P), 1 \leq i \leq N, j \geq 0$.

The above definition enumerates every time interval that is assigned to a partition and is a general representation of periodic partitioning schemes, including those that are generated dynamically by an on-line partition scheduler. It provides a starting point upon which other approaches of defining partitions may be considered.

We refer to the intervals where the processor is unavailable to a partition *blocking time* of the partition. In traditional models where resources are dedicated to a task group, there is no blocking time and we may consider this as a special case corresponding to the partition $\Pi = (\{(0, P)\}, P)$.

Example 2.1 $\Pi_1 = \{(1, 2), (4, 6)\}, 6)$ *is a resource partition whose period is 6 with available resource time from time 1 to time 2 and from time 4 to time 6 every period.See Figure 2.1.*

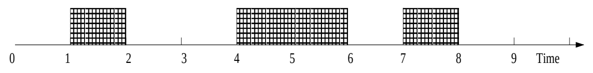


Figure 2.1: Timing Diagram of Example Partition Π_1

Definition 2.2 *We call a resource partition where $N = 1$ a Single Time Slot Periodic Partition (*STSPP*). A partition is otherwise a Multi Time Slot Periodic Partition (*MTSPP*).*

Definition 2.3 *The availability factor of a resource partition* Π *is* $\alpha(\Pi) = (\sum_{i=1}^{n}(E_i - S_i))/P$.

The availability factor of Π_1 in Example 2.1 is $\alpha(\Pi_1) = ((2-1)+(6-4))/6 = 0.5$.

Definition 2.4 *The Supply Function $S(t)$ of a partition Π is the total amount of time that is available in Π from time 0 to time t.*

From the definition we can easily prove some properties of $S(t)$.

- $S(0) = 0$
- $S(t)$ is a monotonically non-decreasing function for t ($t \geq 0$).
- $S(u) - S(v) \leq u - v$ for $u > v \geq 0$.
- $S(t+P) - S(t) = S(P)$ $(t \geq 0)$ (Because the partition is periodic.)
- $S(t) = \lfloor \frac{t}{P} \rfloor \times S(P) + S(t - P\lfloor \frac{t}{P} \rfloor)$ for $t > 0$.

2.2.2 Fixed Priority Scheduling

Given a partition, we now analyze the schedulability problem of a task group that may execute only during the available time of the partition. For both STSPP and MTSPP, the classical utilization bound no longer holds, as can be seen by a contradiction. Suppose there exists a utilization bound U. Consider a task whose period P is equal to the longest blocking time of the partition, and whose execution time is smaller than $U \times P$. Obviously the task is not schedulable on the partition if it requests at the beginning of the longest blocking time, although its utilization factor is smaller than U. Therefore, there is no non-zero utilization bound.

The lack of a utilization bound leads us to reconsider the concept of critical instances. Recall that Liu and Layland [111] defined critical instances to be the time when the task is requested simultaneously with requests of all higher priority tasks. The essential idea of the definition is to construct a worst-case scenario. If the task system is schedulable in the worst case scenario it is definitely schedulable at any time. Therefore, we need to know what the worst case scenario is. An intuitive answer might be the longest blocking time slot. Although this is true for STSPP, it is not the case for MTSPP. For example, consider the schedule of tasks $T_1 = (1, 3)$ and $T_2 = (1, 4)$ in Π_1 in Example 2.1. The task relative deadlines are equal to the corresponding periods. The longest interval for which the resource is unavailable starts from time 2 and ends at time 4. Suppose the "critical instance" starts at time 2. The first requests of both tasks will finish before the deadlines, but the second request of T_2 misses the deadline, as shown in Figure 2.2. Hence, the longest blocking time slot is not necessarily the worst case. This is because the supply after the so defined "critical instance" is not necessarily larger than the supply inside the "critical instance". In general, we need to test for schedulability at more than one "critical instance".

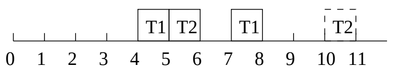


Figure 2.2: Example in Traditional Critical Instance Test

Definition 2.5 *We call $E_1, E_2, ..., E_N$ of a partition*

$$\Pi(\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}, P)$$

interval-based critical points (IBCPs). If a task is requested simultaneously with all higher priority tasks at an IBCP, it is called an interval-based critical instance (IBCI).

Theorem 2.1 *For fixed priority assignment and a task group whose relative deadlines are no more than the periods, a task is schedulable in a partition* Π *if and only if its first request is schedulable in all* IBCI*s.*

Proof. We only need to prove that if a task $T = \{c, p\}$ with relative deadline $d \leq p$ is unschedulable, it will fail on one of the IBCIs.

Let t be the time T misses the deadline but all its previous requests are successfully scheduled. Searching from t backward, let t_0 be the first time when no outstanding execution time of all higher priority tasks and T exists. t_0 always exists because 0 or the finish time of T's previous request are such time points. Only the last request of T exists in (t_0, t) and only higher priority tasks and T are scheduled in (t_0, t). For any higher priority task, we move it backward so that the first request after t_0 now requests at t_0. This makes T's situation worse, so T still misses its deadline at t. Then we move T backward so that T requests at t_0. In the new schedule T will miss deadline at time $t_0 + d$.

- If t_0 is one of the IBCPs, T fails the corresponding IBCI.
- If $S_i + j \times P \leq t_0 < E_i + j \times P$, $1 \leq i \leq N, j \geq 0$, we shift the task requests together so that all those requesting at t_0 now request at E_i. The resource availability between T's request and its deadline decreases by $(E_i + j \times P) - t_0$ in the front and increases by no more than $(E_i + j \times P) - t_0$ in the end. This is an IBCI at E_i and T is not schedulable.
- If t_0 is not in any of the intervals where the resource is available, we shift the task requests together so that all those requesting at t_0 now request at the closest point $E_j + j \times P$ satisfying $1 \leq j \leq N, j \geq 0$ before t_0. The resource availability between T's request and its deadline does not increase in the front and may decrease in the end. This is an IBCI at E_j and T is not schedulable.

So there is always an IBCI that T will fail if T is unschedulable on Π. ■

Following the response-time analysis method of the traditional models [9, 85], we can give the following algorithm as the schedulability test.

Algorithm 2.1

Test $T = (c, p)$ with deadline d. G is the set of higher priority tasks:

```
for $(i = 0; i \leq N; i++)$ /* each loop tests an IBCI */ {
  $r = c$;
  for (;;) {
     $r' = c + \Sigma_{T_i \in G}(\lceil \frac{r}{p_i} \rceil c_i)$;
     $r'' = x$ where $r' == S(E_i + x) - S(E_i)$; }
     If $(r'' > d)$ {deadline missed; return fail; }
     If $(r'' == r)$ {deadline met for this IBCI; break; }
     $r = r''$;
  }
}
return success;
```

Also using similar analysis techniques for traditional task models, we can prove the following corollary from Theorem 2.1.

Corollary 2.1 *For the preemptive fixed priority scheduling discipline, a task group whose relative deadlines are no bigger than the periods is schedulable on a partition with the rate/deadline-monotonic priority assignment (*RMA/DMA*) if it is schedulable on the partition by some priority assignment.*

2.2.3 Dynamic Priority Scheduling

We now turn to the schedulability of task groups on partitions by using dynamic priority schedulers. It turns out that the earliest deadline first (EDF) scheduling algorithm is still optimal in this case. In other words, if there exists a schedule for a task group on a resource partition, the task group is also schedulable using EDF. Same applies to the least slack first scheduling (LSF) algorithm. We could apply the proof techniques in [111, 119] to prove the the next result.

Theorem 2.2 *If a task group τ is schedulable in partition Π by a scheduling policy, it is also schedulable by* EDF *or* LSF*.*

For a periodic task group whose relative deadlines are equal to the corresponding periods, the utilization bound of EDF is 1.0 if the resource is always available. If the group is scheduled on a resource partition, however, the simple bound no longer applies. In the following, we call the recurring instances of a periodic task the *jobs* of the task.

Definition 2.6 ([20]) *Let T be a task, and t a positive real number. The demand bound function $\mathsf{dbf}(T, t)$ denotes the maximum cumulative execution requirement of the jobs of T that have both arrival times and deadlines within any time interval of duration t.*

Theorem 2.3 *A task group G is infeasible on a partition Π if and only if*

$$\Sigma_{T \in G} \mathsf{dbf}(T, t) > S(t_0 + t) - S(t_0)$$

for some positive real numbers t_0 and t.

The proof of Theorem 2.3 is not shown here as it is almost the same as that in [20] except that the resource is not always available. Theorem 2.3 is computationally difficult to convert into a feasibility test algorithm. We derive a better result next.

2.2.4 Critical Partition

The lack of critical instance and the complexity of EDF testing motivate this subsection, where we define the least supply function, the critical partition, critical instance, and a better EDF feasibility testing algorithm.

We allow a task to start issuing requests at any time. As such, we may assume, without loss of generality, that time 0 is the start time for each partition. For the same partition, we may get different representations of the partition by choosing different time instants to start making the resource available, but all of these representations are equivalent for the purpose of the following analysis.

2.2.5 Least Supply Function

Definition 2.7 *The Least Supply Function (*LSF*)* $S^*(t)$ *of a resource partition* Π *is the minimum of* $(S(t+d) - S(d))$ *where* $t, d \geq 0$*.*

Definition 2.8 *A Least Supply Time Interval* (u, v) *is an interval that satisfies* $(S(v) - S(u)) = S^*(u - v)$*.*

Intuitively, $S^*(t)$ is the smallest amount of resource time available to a partition in any interval of length of t.

Some relevant properties of $S^*(t)$ are:

- $S^*(P) = S(P)$

  Proof: By definition $S^*(P) = \min(S(t+P) - S(t)) = S(P)$ ■

- $S^*(t+a) - S^*(t) \geq S^*(a)$, $t \geq 0, a \geq 0$

  Proof. Let $S^*(t+a) = S((t+a)+b) - S(b)$. We have $S^*(t+a) = (S(t+(a+b)) - S(a+b)) + (S(a+b) - S(b)) \geq S^*(t) + S^*(a)$. ■

- $S^*(t+P) - S^*(t) = S^*(P)$, $t \geq 0$

  Proof. From previous property, we have $S^*(t+P) - S^*(t) \geq S^*(P)$. Suppose $S^*(t+P) - S^*(t) > S^*(P)$. Let $S^*(t) = S(t+b) - S(b)$. $S(b+t+P) - S(b) = S(b+t) + S(P) - S(b) = S^*(t) + S^*(P) < S^*(t+P)$. This contradicts the definition of $S^*(t+P)$. Therefore $S^*(t+P) - S^*(t) = S^*(P)$. ■

- $S^*(t)$ is a monotonically non-decreasing function for $(t \geq 0)$.

  Proof. Suppose there exist t_1, t_2, $t_1 < t_2$ and $S^*(t_1) > S^*(t_2)$. Let $S^*(t_2) = S(a+t_2) - S(a)$, then $S^*(t_2) = S(a+t_2) - S(a) > S(a+t_1) - S(a) \geq S^*(t_1)$. Contradiction. ■

- $S^*(t) = \lfloor \frac{t}{P} \rfloor \times S^*(P) + S(t - P\lfloor \frac{t}{P} \rfloor)$ for $t > 0$.

Starting from time 0, $S(t)$ is a step function with N steps every P time units and repeats every P time units; $S^*(t)$ is also a step function with at most $N \times N$ steps every P time units and repeats every P time units.

Note that $S^*(t)$ may be regarded as a special case of supply functions; all the properties of $S(t)$ also hold for $S^*(t)$. Next, we show how to compute the LSF for a partition.

Lemma 2.1 ([116]) *Any time interval where a partition* Π *receives the least resource time has an equal amount of available time in an interval that starts with an* IBCP *point of* Π.

Proof. We can prove this by using a time-slice swapping argument. Given a least supply time interval (u, v), if u is an IBCP point obviously the lemma holds, otherwise let E_i denote the latest IBCP point before u, E_{i+1} the earliest point after u and S_{i+1} the starting point of the available time interval between E_i and E_{i+1}. There are two cases:

1. If $u < S_{i+1}$, then $S^*(v-u) = S(v) - S(u) = S(v) - S(E_i) \geq S(v - (u - E_i)) - S(E_i)$. Because (u, v) is the least supply time interval, only the equal sign can be true. Therefore, (u, v) and $(E_i, (v - u + E_i))$ have the same amount of available time.

2. If $u \geq S_{i+1}$, then $S^*(v-u) = S(v)-S(u) = S(v)-(S(E_{i+1})-(E_{i+1}-u)) = (S(v) + E_{i+1} - u) - S(E_{i+1}) \geq S(v + E_{i+1} - u) - S(E_{i+1})$. For the same reason as in 1, (u, v) and $(E_{i+1}, (v - u + E_{i+1}))$ have the same amount of available time. ■

To compute LSF, first locate all the IBCP points; then for each IBCP point compute S(t) from time 0 to time P taking the IBCP point as the starting point, i.e. time 0. For every t, the minimum of the supply functions S(t) so computed yields the $S^*(t)$ we need.

Figure 2.3 shows how to compute the $S^*(t)$ for Partition $\Pi_1 = (\{(1,2),(4,6)\},6)$ in Example 2.1. There are two IBCP points, time 0 and time 2. Therefore two supply functions are generated and are shown in the figure. The LSF is the bottom envelope of the two $S(t)$s. The algorithm to compute the LSF for a partition is attached in the appendix.

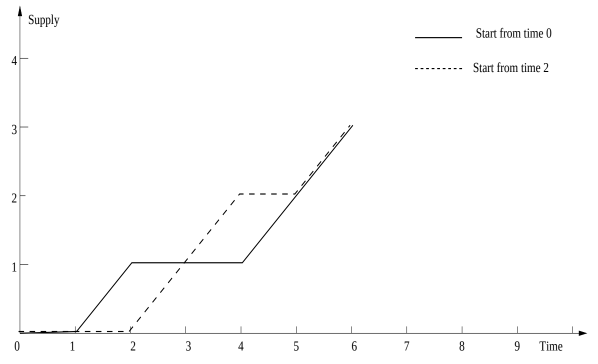


Figure 2.3: Computing Supply Function

2.2.6 Critical Partition

Definition 2.9 *A critical partition of a resource partition* $\Pi = (\Gamma, P)$ *is* $\Pi^* = (\Gamma^*, P)$ *where* Γ^* *has time pairs corresponding to the steps in* $S^*(t)$ *such that* Π^**'s supply function equals* $S^*(t)$ *in* $(0, P)$.

Corollary 2.2 Π^**'s supply function equals* $S^*(t)$ *for* $t \geq 0$.

The critical partition of Π_1 in Example 2.1 is $\Pi_1^* = (\{(2,3),(4,6)\},6)$.

Theorem 2.4 *A task group* τ *is feasible in a partition* Π *if and only if it is feasible in its critical partition* Π^*.

Proof: (i) If τ is infeasible in Π, then according to Theorem 2.3, there exists u and v such that $\mathsf{dbf}(u-v) > S(u) - S(v)$. Suppose π is the request pattern of τ satisfying the inequality. Let us discard all task requests before v and shift π backward so that v is aligned up at time 0, then try to schedule this request pattern on Π^*. The total request with deadlines before $u-v$ is $\mathsf{dbf}(u-v) > S(u) - S(v) > S^*(u-v)$, the available resource before $u-v$. So τ is infeasible in Π^*.

(ii) If τ is infeasible in Π^*, according to Theorem 2.3, there is u and v such that $\mathsf{dbf}(u-v) > S^*(u) - S^*(v)$. Suppose π is the request pattern of τ satisfying the inequality. Let's discard all task requests before v and shift π backward so that v is aligned up at time 0. The total request with deadlines before $u-v$ is $\mathsf{dbf}(u-v) > S^*(u) - S^*(v) > S^*(u-v)$, the available resource before $u-v$. Now let $S^*(u-v) = S((u-v)+a) - S(a)$. We shift π again from 0 to a and try to schedule π on Π. The request of π after a that must be finished before $(t-t_0)+a$ is $\mathsf{dbf}(u-v) > S^*(u-v) = S((u-v)+a) - S(a)$. So τ is infeasible in Π. ■

2.2.7 Fixed Priority Scheduling

Definition 2.10 *The critical instance of a task on partition Π is when it is requested simultaneously with all higher priority tasks at time 0 on the critical partition Π^*.*

Theorem 2.5 *Suppose the preemptive fixed priority scheduling policy is used to schedule a task group on a partition by some priority assignment where all deadlines are no bigger than the corresponding periods. If a task's first request is schedulable in a partition's critical instance, then the task is schedulable in the partition.*

Proof: We show that if a task is unschedulable, it will fail in the critical instance. If a task $T = (c, p)$ with deadline $d \leq p$ is unschedulable, it will fail in one of the IBCI. Let the IBCP be E. We compare the resource supply between IBCI and the critical instance. By definition $S(E+x) - S(E) \geq S^*(x)$, $0 \leq x \leq d$. The resource supply in the critical instance always lags behind that in the IBCI. So the schedule sequence of all tasks does not change in the critical instance. In other words, T could not be finished before d due to early resource supply before some higher priority task requests come in. So T will fail in the critical instance. ■

We may modify Algorithm 2.1 to test only the critical instance. Note that Theorem 2.5 is only a sufficiency test. There may be task groups that are schedulable but fail to pass the critical instance test of Algorithm 2.1.

Example 2.2 *Partition* $\Pi_2 = (\{(1,2),(4,6),(7,8)\},8)$. *Its critical partition is* $\Pi_2^* = (\{(2,3),(4,5),(6,8)\},8)$. *Task group* $\{T_1 = (1,4), T_2 = (1,6)\}$ *is schedulable on* Π_2 *with* RMA. *This can be checked with Algorithm 2.1. However, in the critical instance,* T_2 *misses deadline at* 6. *Figure 2.4 is the supply functions of* IBCI*s and the critical instance.*

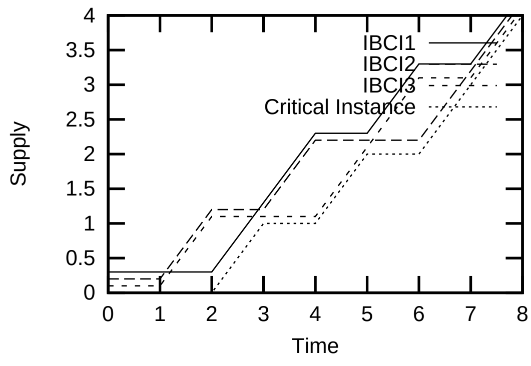


Figure 2.4: Critical Instance Test

2.2.8 Dynamic Priority Scheduling

From Theorems 2.3 we have the following corollary.

Corollary 2.3 *A task group* G *is infeasible on a partition* Π *if and only if*

$$\Sigma_{T \in G} \mathsf{dbf}(T,t) > S^*(t)$$

for some positive real number t.

Since $S^*(t)$ is the supply function of the critical partition, we can devise a pseudo-polynomial feasibility test algorithm for task group scheduling on a partition similar to the ones developed for traditional feasibility test, e.g., in [132].

2.3 Bounded-Delay Resource Partition Model

Although the definition of static resource partitions can model all kinds of resource partitions, it has several serious drawbacks. First, it requires explicit statements of all time intervals. Some resource partitions might not be able to provide these data. Second, not all time interval information is necessary to schedule task groups that run on partitions. Therefore, we need to define a model that not only simplifies the presentation but also characterizes resource partitions on a higher level. Towards this end, in this section, we introduce the bounded-delay resource partition model. We start with a few preliminary definitions.

Definition 2.11 *The Partition Delay Δ of Partition Π is the smallest d so that for any $t_0$and t_1, $(t_1 \geq t_0)$, $(t_1 - t_0 - d)\alpha(\Pi) \leq (S(t_1) - S(t_0)) \leq (t_1 - t_0 + d)\alpha(\Pi)$.*

Definition 2.12 *Let h denote the execution rate of the resource where partition Π is implemented. The Normalized Execution of partition Π is an allocation of resource time to Π at a uniform, uninterrupted rate of $(\alpha(\Pi) \times h)$.*

Intuitively, Δ is the maximum delay of the available time interval of any length in the partition Π relative to its normalized execution regardless of the starting point.

Definition 2.13 *A bounded delay resource partition Π is a tuple (α, Δ) where α is the availability factor of the partition and Δ is the partition delay.*

Note that the definition actually defines a set of partitions because there are many different partitions in the static partition model that may satisfy this requirement.

Definition 2.14 *Virtual Time* $V(t) = \frac{S(t)}{\alpha}$.

Intuitively, virtual time $V(t)$ is the time point that the actual time t simulates. Shown on the supply function graph, virtual time corresponds to the time point on the normalized function with the same amount of supply as the actual time on the supply function.

Definition 2.15 *Actual Time* $P(t)$ *of a virtual time* t *is defined as the smallest time value such that* $V(P(t)) = t$.

Actual Time may be considered as a quasi-inverse function of Virtual Time. Although virtual time function is not a one-to-one mapping, we may add a minimum operation on the mapping, thus eliminating all many-to-one mappings except for the earliest time points. Notice that though $V(P(t)) = t$ is true by definition, $P(V(t)) = t$ might not be true.

Definition 2.16 *Virtual Time Scheduling on a partition* Π *is the scheduling constraint such that a job is eligible to run only when it is released and its release time is no less than the current virtual time of* Π.

In short, Virtual Time Scheduling is a scheduling scheme whereby schedules are performed according to virtual time instead of physical time. Obviously this scheme could be applied to all known scheduling algorithms designed for dedicated resources. Therefore, we have Virtual Time Rate Monotonic Scheduling, Virtual Time Earliest-Deadline-First Scheduling, and other virtual time scheduling algorithms.

Theorem 2.6 *Let S_n denote a scheduler of scheduling $T_i(C_i, P_i)$ $(1 < i \leq n)$ on a dedicated resource with capacity of the same as the normalized execution of Partition $\Pi(\alpha, \Delta)$. Also let S_p denote the virtual time S_n scheduler of scheduling $T_i(C_i, P_i)$ $(1 < i \leq n)$ on Π. If using S_n, every job of T_i $(1 < i \leq n)$ is finished at least Δ earlier than its deadline, then S_p is also a valid scheduler.*

Theorem 2.6 justifies the observation that we may use essentially the same algorithms of scheduling tasks on dedicated resources by applying the virtual time scheduling scheme. We describe this method in two steps:

- Construct Scheduler S_n.
- For any interval (t_1, t_2) assigned to Π'_i in S_n, assign $(P(t_1), P(t_2))$ to Π_i in S_p.

Informally, time on different levels may be expressed in the following way:

Actual time on dedicated resource:	t
Virtual time on partition:	t
Actual time on partition:	$(t - \Delta, t + \Delta)$
Actual time of job finishing is earlier than:	$(t + \Delta - \Delta)$

We present a formal proof below.

Proof: Let (t_1, t_2) denote an interval on S_n, (t'_1, t'_2) an interval on S_p whereas $t'_1 = P(t_1)$ and $t'_2 = P(t_2)$. Also let $S_\Pi(t)$ denote the supply functions for Π.

We have

$$\begin{aligned} & (t_2 - t_1)\alpha \\ = \; & S_\Pi(P(t_2)) - S_\Pi(P(t_1)) \\ = \; & S_\Pi(t'_2) - S_\Pi(t'_1) \end{aligned}$$

Therefore

$$\begin{aligned} & (t_2 - t_1) \\ \leq \ & (t_2 - t_1) \\ \leq \ & (t_2 - t_1) \end{aligned}$$

$$\begin{aligned} & (V(P(t_2)) - V(P(t_1))) \\ \leq \ & (t_2 - t_1) \\ \leq \ & (V(P(t_2)) - V(P(t_1))) \end{aligned}$$

$$\begin{aligned} & (V(t'_2) - V(t'_1)) \\ \leq \ & (t_2 - t_1) \\ \leq \ & (V(t'_2) - V(t'_1)) \end{aligned}$$

$$\begin{aligned} & (t'_2 - t'_1 - \Delta) \\ \leq \ & (t_2 - t_1) \\ \leq \ & (t'_2 - t'_1 + \Delta) \end{aligned}$$

$$\begin{aligned} & (t'_2 - t'_1 - \Delta)\alpha \\ \leq \ & (t_2 - t_1)\alpha \\ \leq \ & (t'_2 - t'_1 + \Delta)\alpha \end{aligned}$$

$$(t'_2 - t'_1 - \Delta)\alpha$$

$$\begin{aligned} &\leq (S_\Pi(t'_2) - S_\Pi(t'_1)) \\ &\leq (t'_2 - t'_1 + \Delta)\alpha \end{aligned}$$

Therefore, if a job is finished at time t'_2 which is Δ earlier than the deadline on S_n, the corresponding one on S_p is guaranteed to finish before $t'_2 + \Delta$ which is no later than its deadline.

■

Virtual time scheduling increases the scheduling complexity by introducing the conversion between virtual time and physical time. However, without virtual time scheduling, original scheduling algorithms will not execute the task group as the same order and amount as that with virtual time scheduling. Therefore, the following theorem is proven to circumvent virtual time conversion and directly use original scheduling algorithms such as RMA and EDF.

We first introduce the concept of robustness which is first defined in [[117]].

Definition 2.17 ([117]) *A scheduling algorithm is robust if schedulability is preserved under reduction in system load.*

System load reduction includes decreasing task computation time, or increasing task period, or using a faster processor. Intuitively, when system load is reduced, a scheduler that used to work would still preserve its schedulability. It can be easily shown that both preemptive RMA and preemptive EDF hold that property. However, as shown in [[117]], some scheduling algorithms like non-preemptive RMA do not.

Definition 2.18 *A scheduling algorithm A is resource-burst-robust if during any of its busy interval when the supply is advanced earlier the schedulability stills preserves.*

Theorem 2.7 *A task group τ is schedulable on Partition $\Pi(\alpha, \Delta)$ using a non-virtual-time scheduling algorithm S if i), τ is schedulable using S on the normalized* *execution of* Π *with every task deadline reduced by* Δ*. ii), S is a resource-burst-robust scheduler.*

Proof: We prove this theorem by contradiction.

Suppose both i) and ii) hold but τ is not schedulable on Π. Let t_2 denote the time when a deadline miss occurs and t_1 is the latest time point before t_2 when a job arrives to the empty job queue.

Though interval (t_1, t_2) can be viewed as a busy interval, notice that t_1 is defined as the last time point when a job arrives to the empty job queue rather than the time point after an idle interval. When a task group runs on a partition, the physical resource is not always available. Therefore, the time when a job arrives at the empty job queue may not be the same as the time when the job starts to run.

Furthermore, we construct another partition Π' such that Π' starts at $t_1 + \Delta$ and then runs the same as the normalized execution of Π. According to i), τ is schedulable on Π' and there is no deadline miss during the interval of (t_1, t_2).

According to the definition of partition delay,

$$(t_2 - t_1 - \Delta)\alpha \le S(t_2) - S(t_1)$$

That is

$$S'(t_2) - S'(t_1) \le S(t_2) - S(t_1)$$

Because of ii), schedulability preserves.

It contradicts with the deadline miss.

■

It is easy to show that both preemptive RMA and preemptive EDF are resource-burst-robust.

Theorem 2.8 *Rate Monotonic Algorithm is resource-burst-robust.*

Proof: To test the schedulability of RMA is to check:

$$\sum_{j=1}^{i-1} \lceil \frac{t}{P_j} \rceil C_j + C_i \leq t$$

for $t \leq T_n$.

Here let $S(t)$ denote the supply function before the resource enhancement and let $S'(t)$ denote the supply function after the resource enhancement. By definition of resource enhancement, we have $S'(t) \geq S(t)$ for any t.

Also for the dedicated resource, we have $S(t) = t$.

Therefore,

$$\sum_{j=1}^{i-1} \lceil \frac{t}{P_j} \rceil C_j + C_i \leq S(t) \leq S'(t)$$

The schedulability is preserved. ∎

Similarly, we have

Theorem 2.9 *Earliest Deadline First Algorithm is resource-burst-robust.*

Proof: To test the schedulability of EDF is to check:

$$\sum_{j=1}^{n} \lfloor \frac{t}{P_j} \rfloor C_j \leq t$$

for $t \leq LCM(T_j)(1 \leq j \leq n)$.

Again let $S(t)$ denote the supply function before the resource enhancement and let $S'(t)$ denote the supply function after the resource enhancement. By definition of resource enhancement, we have $S'(t) \geq S(t)$ for any t.

Also for dedicated resource, we have $S(t) = t$.

Therefore,

$$\sum_{j=1}^{n} \lfloor \frac{t}{P_j} \rfloor C_j \leq S(t) \leq S'(t)$$

The schedulability is preserved. ∎

Corollary 2.4 *A task group is schedulable on a partition* $\Pi = (\alpha, \Delta)$ *if its* RMA *schedule on the normalized execution has the property that all requests finish at least* Δ *before the deadlines.*

Corollary 2.5 *A task group is schedulable on a partition* $\Pi = (\alpha, \Delta)$ *if its* EDF *schedule on the normalized execution has the property that all requests finish at least* Δ *before the deadlines.*

Definition 2.19 *The jitter-tolerance of a periodic task system* τ *is defined to be the largest* Δ *such that even if every job is released* Δ *time units late, all the tasks in* τ *should still be able to meet their deadlines.*

Let S_{EDF} denote the EDF schedule of a task group τ in a normalized execution. Let λ_{EDF} denote the largest λ such that all the tasks in the task group complete execution at least λ before their deadlines in S_{EDF}. Then the jitter-tolerance of τ is exactly equal to λ_{EDF}. A task system τ is schedulable in a partition Π (α, Δ) if $U(\tau) \leq \alpha$ and the jitter-tolerance of τ is no bigger than Δ.

We also have the admission control tests for RMA and EDF:

Theorem 2.10 *A task group* τ $T_i = (c_i, p_i), (1 \leq i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta)$ *by RMA scheduling if* $\sum_{i=1}^{n} \frac{c_i}{p_i - \Delta} \leq \alpha(\Pi) n (2^{\frac{1}{n}} - 1)$.

Similarly, we have

Theorem 2.11 *A task group* τ $T_i = (c_i, p_i), (1 \leq i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta)$ *by EDF scheduling if* $\sum_{i=1}^{n} \frac{c_i}{p_i - \Delta} \leq \alpha(\Pi)$.

2.4 Regularity-Based Resource Partition Model

In previous sections, we introduced two resource partition models: static resource partition model and the bounded delay partition model. Both models study resource partitioning in the real-number domain. In this section, we switch to integer domain. We show that resource partitions in the integer domain have distinctively interesting properties. We first define three types of regularities before defining regularity-based resource partition model. Then we show the scheduling transparency property of a specific type of resource partition –regular partition. Next, we generalize regularity-based resource partitions and discuss the task level scheduling issues. Finally, we look into resource level scheduling issues.

2.4.1 Regularity

In this section we introduce our measurements of partitions. We first define instant regularity and later temporal regularity and supply regularity.

Definition 2.20 *The Instant Regularity $I(t)$ at time t on partition Π is given by $S(t) - t\alpha(\Pi)$.*

The notion of instant regularity is called dynamic regularity by Shigero et al. in [138]. We prefer the term instant regularity because it better captures the idea that it pertains to a particular time point and that there are different types of regularity. At a particular time point the instant regularity measures the difference between the amount of supply that the partition has already gotten from the resource scheduler since time 0 and the amount that the partition would get if it were following its normalized execution. As shown in Figure 2.5 the instant regularity at time t is equal to the distance from the supply function to the normalized function. Notice that it is a real number and could be negative as well.

Definition 2.21 *[138] Let a,b,e,k be non-negative integers, the Temporal Regularity $R_T(\Pi)$ of Partition Π is equal to the minimum value of k such that $\forall a, \forall b.\ a < b$, $0 \leq \exists e \leq k$, $|I(b-e) - I(a)| < 1$.*

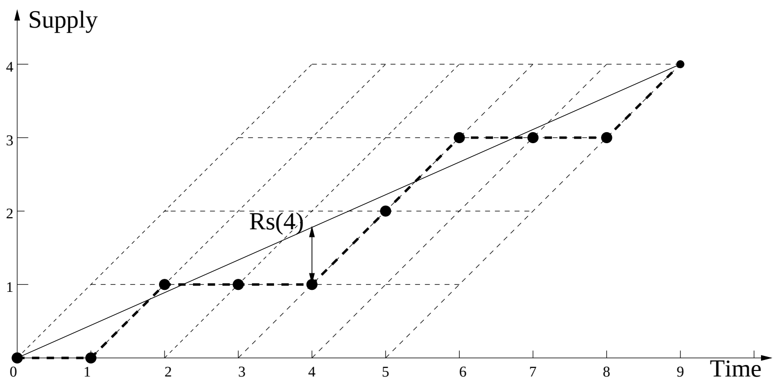


Figure 2.5: Instant Regularity at Time 4 of Partition Π_1 in Example 2.1

The temporal regularity measures the overall difference between the partition supply function and the normalized supply function from the time dimension. This measurement considers all possible time intervals no matter which time point is the start point.

Definition 2.22 *Let a,b,k be non-negative integers, the Supply Regularity* $R_S(\Pi)$ *of Partition* Π *is equal to the minimum value of* k *such that* $\forall a, \forall b.\ a < b,\ 0 \leq k,$ $|I(b) - I(a)| < k$.

The supply regularity is the upper bound on the amount by which the actual supply during any time interval is more than or less than the amount of supply that the partition is supposed to obtain.

These two regularities are actually interchangeable.

Corollary 2.6 *A partition with temporal regularity of* k *has supply regularity of* $1 + k\frac{P}{N}$.

Corollary 2.7 *A partition with supply regularity of* k *has temporal regularity of* $\left\lceil (k-1)\frac{P}{N} \right\rceil$.

While the temporal regularity is an integer, the supply regularity can be a real number. Temporal regularity is more suitable for task-level scheduling due to its real-time nature while supply regularity is more suitable for resource-level scheduling as we shall see in later section. The interchangeability of these two types of regularities bridges the two levels of scheduling and keeps each level focused on its own concerns.

Definition 2.23 *[138] A Regular Partition is a partition with temporal regularity of 0.*

By Corollary 2.6 a partition with supply regularity $R_S(\Pi) \leq 1$ is also a regular partition.

Definition 2.24 *A k-Temporal-Irregular Partition is a partition with temporal regularity of k where $k > 0$. A k Supply-Irregular Partition is a partition with supply regularity of k where $k > 1$.*

2.5 Task Level Scheduling

Given a partition, we now analyze the schedulability problem of a task group that may execute only during the available time slots of the partition. We first investigate the problem of regular partitions and later of irregular partitions.

2.5.1 Regular Partition

In this section, we show that regular partitions preserve the utilization bounds of both fixed priority scheduling (rate monotonic) and dynamic priority scheduling (earliest deadline first).

2.5.2 Fixed Priority

The computation for a utilization bound leads us to consider the concept of critical instances in the partition environment. Results from the previous section apply to the integer time domain as well. In this section, we first recall some results from [116], and then we compute the critical partitions and finally the utilization bounds on regular partitions.

Theorem 2.12 *The critical partition* Π^* *of a regular partition* Π *is a partition with supply function* $S^*(t) = \lfloor \alpha(\Pi)t \rfloor$.

Proof: Let $S(t)$ denote the supply function of Π. To prove this lemma we only need to prove that the least supply function of Π is $\lfloor \alpha(\Pi)t \rfloor$, i.e., $min(S(b) - S(a)) = \lfloor \alpha(\Pi)(b-a) \rfloor$ for $\forall a \geq 0$, $\forall b \geq a$. From the definition of regular partitions, we have

$$|I(b) - I(a)| < 1$$

$$\lfloor \alpha(\Pi) \times (b-a) \rfloor \leq S(b) - S(a) \leq \lceil \alpha(\Pi) \times (b-a) \rceil$$

Since $S(b) - S(a)$ is also an integer, it may be equal to either $\lfloor \alpha(\Pi) \times (b-a) \rfloor$ or $\lceil \alpha(\Pi) \times (b-a) \rceil$. We proceed with a proof by contradiction and suppose there exists t_1 so that

$$min(S(b+t_1) - S(b)) \neq \lfloor \alpha(\Pi)t_1 \rfloor$$

$$\text{Hence } S(b+t_1) - S(b) = \lceil \alpha(\Pi) \times (t_1) \rceil \text{ for } \forall b \geq 0$$

$$\text{and } \lceil \alpha(\Pi)(t_1) \rceil \neq \alpha(\Pi)(t_1) \tag{2.1}$$

Then

$$\begin{aligned} & S(b+2t_1) - S(b) \\ = \; & S(b+2t_1) - S(b+t_1) + S(b+t_1) - S(b) \\ = \; & 2\lceil \alpha(\Pi)t_1 \rceil \end{aligned}$$

For the same reason, for any integer n,

$$S(b+nt_1) - S(b) = n\lceil \alpha(\Pi)t_1 \rceil$$

We also have

$$S(b + nt_1) - S(b) \leq \lceil \alpha(\Pi)(nt_1) \rceil$$

Therefore,

$$n \lceil \alpha(\Pi)t_1 \rceil \leq \lceil \alpha(\Pi)(nt_1) \rceil$$

$$\lceil \alpha(\Pi)t_1 \rceil = \alpha(\Pi)(t_1)$$

This contradicts with (2.1). Therefore,

$$S^*(b - a) = min(S(b) - S(a)) = \lfloor \alpha(\Pi)(b - a) \rfloor$$

■

Lemma 2.2 *A task group G with two tasks with the restriction that the ratio between their request periods is less than 2 is schedulable on a regular partition* Π *by rate monotonic scheduling if* $U(G) \leq 2(2^{\frac{1}{2}} - 1)\alpha(\Pi)$.

Proof: We prove the lemma holds on the critical partition of Partition Π. Similar to the proof in the seminal Liu & Layland paper [111], let τ_1 and τ_2 be two tasks with periods equal to P_1 and P_2 and their worst case execution times $C_1$and C_2, respectively. Assume that $P_2 > P_1$ so τ_1 has higher priority than τ_2. The minimum of the utilization factor U occurs when

$$\begin{aligned} C_1 &= S(P_2) - S(P_1 \times \left\lfloor \frac{P_2}{P_1} \right\rfloor) \\ &= \lfloor \alpha(\Pi) \times P_2 \rfloor - \left\lfloor \alpha(\Pi) \times P_1 \times \left\lfloor \frac{P_2}{P_1} \right\rfloor \right\rfloor \end{aligned}$$

and

$$\begin{aligned} C_2 &= S(P_2) - C_1 \times \left\lceil \frac{P_2}{P_1} \right\rceil \\ &= \lfloor \alpha(\Pi) \times P_2 \rfloor - C_1 \times \left\lceil \frac{P_2}{P_1} \right\rceil \end{aligned}$$

Let $\alpha(\Pi) = \frac{n}{p}$, $P_1 = \frac{b_1 p + a_1}{n}$ where $0 \leq a_1 < p$, $P_2 = \frac{b_2 p + a_2}{n}$ where $0 \leq a_2 < p$.

$$\text{Because } \frac{C_1}{P_1} < \frac{n}{p}, P_1 n > C_1 p$$

$$\text{also because } P_1 n = b_1 p + a_1$$

$$b_1 p + a_1 > C_1 p$$

Since $C_1 \geq 1$, $b_1 \geq 1$. For the same reason, $b_2 \geq 1$.

$$\begin{aligned} C_1 &= \left\lfloor \frac{nP_2}{p} \right\rfloor - \left\lfloor \frac{nP_1}{p} \right\rfloor \\ &= b_2 - b_1 \end{aligned}$$

Similarly,

$$\begin{aligned} C_2 &= b_2 - (b_2 - b_1) \times 2 \\ &= 2b_1 - b_2 \end{aligned}$$

Therefore

- $b_2 - b_1 \geq 1$
- $2b_1 - b_2 \geq 1$

Now we compute the utilization factor:

$$\begin{aligned} U &= \frac{C_1}{P_1} + \frac{C_2}{P_2} \\ &= (\frac{\left\lfloor \frac{nP_2}{p} \right\rfloor - \left\lfloor \frac{nP_1}{P} \right\rfloor}{\frac{b_1 p + a_1}{n}} + \frac{2\left\lfloor \frac{nP_1}{p} \right\rfloor - \left\lfloor \frac{nP_2}{p} \right\rfloor}{\frac{b_2 p + a_2}{n}}) \\ &= (\frac{(b_2 - b_1)}{b_1 p + a_1} + \frac{2b_1 - b_2}{b_2 p + a_2}) \times n \\ &\geq (\frac{(b_2 - b_1)}{b_1 + 1} + \frac{2b_1 - b_2}{b_2 + 1}) \times \frac{n}{p} \end{aligned}$$

Thus the problem translates to another problem of solving the minimum value of

$$(\frac{(b_2 - b_1)}{b_1 + 1} + \frac{(2b_1 - b_2)}{b_2 + 1})$$

where b_1,$b_2$$\geq 1$, $b_2 - b_1 \geq 1$ and $2b_1 - b_2 \geq 1$.

Since the domain of task timing parameters is integral, the execution time of a task can be increased only by an integer. Instead of computing the maximum utilization factor for schedulable task groups in the worst scenario, we may compute the minimum utilization factor for unschedulable task groups in the same scenario in order to compute the utilization bound, i.e., instead of computing the maximum of

$$(\frac{(b_2-b_1)}{b_1+1}+\frac{(2b_1-b_2)}{b_2+1})$$

we compute the minimum of

$$((\frac{(b_2-b_1+1)}{b_1+1}+\frac{(2b_1-b_2)}{b_2+1}),(\frac{(b_2-b_1)}{b_1+1}+\frac{(2b_1-b_2+1)}{b_2+1})) \tag{2.2}$$

by increasing the execution time of either τ_1 or τ_2 by 1.

Let x=b$_1$+1, y=b$_2$+1. Because $b_2 > b_1$

$$\begin{aligned}(2.2) &= (\frac{(b_2-b_1)}{b_1+1}+\frac{(2b_1-b_2+1)}{b_2+1})\\ &= \frac{(y-x)}{x}+\frac{(2x-y)}{y}\\ &= \frac{y}{x}+\frac{2x}{y}-2\\ &\geq 2(2^{\frac{1}{2}}-1)\end{aligned}$$

The equality is reached when $y=\sqrt{2}x$. ∎

Lemma 2.3 *A task group G of m tasks with the restriction that the ratio between any two request periods is less than 2 is schedulable on a regular partition Π by rate monotonic scheduling if $U(G) \leq m(2^{\frac{1}{m}}-1)\alpha(\Pi)$.*

Proof: Let $\tau_1, \tau_2, ..., \tau_m$ denote the m tasks. Let C_1, C_2, …,C_m be the executions times of the tasks that fully utilize the processor and minimize the processor utilization factor. Assume that $P_m > P_{m-1} > \cdots > P_2 > P_1$. Let U denote the processor utilization factor.

U is minimum when

$$\begin{aligned} C_1 &= S(P_2) - S(P_1) \\ C_2 &= S(P_3) - S(P_2) \\ &... \\ C_{m-1} &= S(P_m) - S(P_{m-1}) \\ C_m &= S(P_m) - 2(C_1 + C_2 + ... + C_{m-1}) \end{aligned}$$

Let $|S| = \frac{n}{p}$, $P_1 = \frac{b_1 p + a_1}{n}$ where $0 \le a_1 < p$, $P_2 = \frac{b_2 p + a_2}{n}$ where $0 \le a_2 < p$, ... ,$P_m = \frac{b_m p + a_m}{n}$ where $0 \le a_m < p$.

Because

$$\frac{C_1}{P_1} < \frac{n}{p}$$

$$P_1 \times n > C_1 \times p$$

also because

$$P_1 n = b_1 p + a_1$$

$$b_1 p + a_1 > C_1 p$$

Since $C_1 \ge 1$, $b_1 \ge 1$. For the same reason, $b_2 \ge 1$.

$$\begin{aligned} C_1 &= \left\lfloor \frac{nP_2}{p} \right\rfloor - \left\lfloor \frac{nP_1}{p} \right\rfloor \\ &= b_2 - b_1 \end{aligned}$$

similarly,

$$\begin{aligned} C_2 &= b_2 - (b_2 - b_1) \times 2 \\ &= 2b_1 - b_2 \end{aligned}$$

Therefore

- $b_2 - b_1 > 0$
- $2b_1 - b_2 > 0$

Overall,

$$\begin{aligned} C_1 &= b_2 - b_1 \\ C_2 &= b_3 - b_2 \\ &... \\ C_{m-1} &= b_m - b_{m-1} \\ C_m &= b_m - 2((b_2 - b_1) + (b_3 - b_2) \\ &+... + (b_m - b_{m-1})) \\ &= 2b_1 - b_m \end{aligned}$$

$$\begin{aligned} U &= \frac{C_1}{P_1} + \frac{C_2}{P_2} + ... + \frac{C_m}{P_m} \\ &= (\frac{b_2 - b_1}{\frac{b_1 p + a_1}{n}} + \frac{b_3 - b_2}{\frac{b_2 p + a_2}{n}} + ... + \frac{2b_1 - b_m}{\frac{b_m p + a_m}{n}}) \\ &= (\frac{(b_2 - b_1)}{b_1 p + a_1} + \frac{b_3 - b_2}{b_2 p + a_2} + ... + \frac{2b_1 - b_m}{b_m P + a_m}) \times n \\ &\geq (\frac{(b_2 - b_1)}{b_1 + 1} + \frac{b_3 - b_2}{b_2 + 1} + ... + \frac{2b_1 - b_m}{b_m + 1}) \times \frac{n}{p} \end{aligned}$$

Let $n_1 = b_1 + 1$, $n_2 = b_2 + 1$,...$n_m = b_m + 1$. For the same reason we compute the minimum of

$$(\frac{b_2 - b_1}{b_1 + 1} + \frac{b_3 - b_2}{b_2 + 1} + ... + \frac{2b_1 - b_m + 1}{b_m + 1})$$

$$\begin{aligned} &(\frac{b_2 - b_1}{b_1 + 1} + \frac{b_3 - b_2}{b_2 + 1} + ... + \frac{2b_1 - b_m + 1}{b_m + 1}) \\ = &(\frac{n_2 - n_1}{n_1} + \frac{n_3 - n_2}{n_2} + ... \\ &+ \frac{n_m - n_{m-1}}{n_m} + \frac{2n_1 - n_m}{n_m}) \end{aligned}$$

$$\begin{aligned} &= (\frac{n_2}{n_1} + \frac{n_3}{n_2} + ... + \frac{n_m}{n_{m-1}} + \frac{2n_1}{n_m} - m) \\ &\geq m(2^{\frac{1}{m}} - 1) \blacksquare \end{aligned}$$

Theorem 2.13 *A task group G of m tasks is schedulable on a regular partition Π by Rate Monotonic if $U(G) \leq m(2^{\frac{1}{m}} - 1)\alpha(\Pi)$.*

Proof: Let $\tau_1, \tau_2, ..., \tau_m$ denote the m tasks. Let C_1, C_2, $\ldots$,C_m be the execution times of the tasks that fully utilize Partition Π and minimize the resource utilization factor. Let U denote the resource utilization factor. Suppose that for some i, $\left\lfloor \frac{P_m}{P_i} \right\rfloor > 1$. To be specific, let $P_m = qP_i + r$, $q > 1$ and $r \geq 0$. Let us replace the task τ_i by a task τ_i' such that $P_i' = qP_i$, and $C_i' = C_i$, and increase C_m by the amount needed to again fully utilize the partition. This increase is at most $C_i(q-1)$, the time within the critical time zone of τ_m occupied by τ_i but not by τ_i'. Let U' denote the utilization factor of such a set of tasks. We have

$$U' < U + [(q-1)C_i/P_m] + (C_i/P') - (C_i/P_i)$$

or

$$U' \leq U + C_i(q-1)[1/(qP_i + r) - (1/qP_i)]$$

Since $q - 1 > 0$ and $[1/(qP_i + r)] - (1/qP_i) \leq 0$, $U' \leq U$. Therefore we conclude that in determining the least upper bound of the processor utilization factor, we need only consider task groups in which the ratio between any two request periods is less than 2. The theorem is thus proven. $\blacksquare$

This result is exactly the same as Liu and Layland's bound for m tasks on a dedicated resource which means the utilization bound is not affected by resource partitioning at all for regular partitions.

2.5.3 Dynamic Priority

After examining the utilization bound of fixed-priority scheduling for regular partitions above, we now give two lemmas and then show that the utilization bound of dynamic-priority scheduling also remains the same for regular partitions.

Lemma 2.4 *For any non-negative real numbers m and n, $\lfloor m+n \rfloor - \lceil n \rceil \le \lfloor m \rfloor$.*

Proof: Let A denote the integer part of m, a the fractional part of m, B the integer part of n, b the fractional part of n, then

$$\begin{aligned}
& \lfloor m+n \rfloor - \lceil n \rceil \\
= \; & \lfloor A+a+B+b \rfloor - \lceil B+b \rceil \\
\text{if } b \neq 0 \text{ then} & \\
= \; & A+B+\lfloor a+b \rfloor - (B+1) \\
\le \; & A+B+1-(B+1) \\
= \; & A \\
= \; & \lfloor m \rfloor \\
\text{if } b = 0 \text{ then} & \\
= \; & A+B-B \\
= \; & \lfloor m \rfloor \blacksquare
\end{aligned}$$

Lemma 2.5 *For real numbers m and n, $\lfloor m \rfloor + \lfloor n \rfloor \le \lfloor m+n \rfloor$.*

Proof: Using the same notations as in Lemma 2.4, we have

$$\begin{aligned}
& \lfloor m \rfloor + \lfloor n \rfloor \\
= \; & \lfloor A+a \rfloor + \lfloor B+b \rfloor \\
= \; & A+B \\
\le \; & A+B+\lfloor a+b \rfloor \\
= \; & \lfloor A+a+B+b \rfloor \\
= \; & \lfloor m+n \rfloor \blacksquare
\end{aligned}$$

Theorem 2.14 *[138] A task group G with n periodic tasks, τ_1,τ_2,...,τ_n is schedulable on a regular partition Π by the earliest-deadline-first policy if and only if $U(G) \leq \alpha(\Pi)$.*

Intuitively, a task group is feasible when its least demand during any time interval t is always no greater than the supply during the same time interval. We visualize this on the supply graph. First, because the least demand is no greater than $U(G)t$ and $U(G) \leq \alpha(\Pi)$ the demand function is no greater than the normalized supply function. Second, because the demand is an integer the demand is at most the largest integer number below the normalized supply function, i.e., the supply function of the critical partition. Therefore, the least demand is no greater than the supply, thus guaranteeing the schedulability of the task group. We give a formal proof below.

Proof: By contradiction. Let

$$E(t) = \sum_{i=1}^{n}(\left\lceil \frac{t}{p_i} \right\rceil \times c_i)$$

$$N(t) = \sum_{i=1}^{n}(\left\lfloor \frac{t}{p_i} \right\rfloor \times c_i)$$

Suppose there is a deadline miss after an idle slot when $U(G) \leq \alpha(\Pi)$. Let the first deadline miss occur at $(v+d)$ and v be the last idle slot before the deadline miss. It follows that all jobs that arrived before v are completed, and that there is not any idle slot in $[v, v+d]$. Consequently, $S(v+d) - S(v) < N(v+d) - E(v)$ should be satisfied because $S(v+d) - S(v)$ is the number of the available time slots required in $[v, v+d]$ to meet all deadlines by EDF.

$$\begin{aligned} & N(d+v) - E(v) \\ = \quad & \sum_{i=1}^{n}(\left\lfloor \frac{d+v}{p_i} \right\rfloor \times c_i) - \sum_{i=1}^{n}(\left\lceil \frac{v}{p_i} \right\rceil \times c_i) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{n}((\left\lfloor\frac{d+v}{p_i}\right\rfloor - \left\lceil\frac{v}{p_i}\right\rceil) \times c_i) \\
&\le \sum_{i=1}^{n}(\left\lfloor\frac{d}{p_i}\right\rfloor \times c_i) \text{ Lemma 2.4} \\
&\le \left\lfloor(\sum_{i=1}^{n}(\frac{d}{p_i}) \times c_i)\right\rfloor \text{ Lemma 2.5} \\
&= \lfloor U(G) \times d\rfloor
\end{aligned}
$$

However, from the property of regular partitions we have

$S(v + d) - S(v) \geq \lfloor\alpha(\Pi)d\rfloor \geq \lfloor U(G)d\rfloor \geq N(v + d) - E(v)$

This contradicts that there is a deadline miss.

When $U(G) > \alpha(\Pi)$, clearly, no scheduling algorithm can schedule G on Π.

■

2.5.4 Irregular Partitions

Definition 2.25 *The Virtual Time V(t) of a partition Π is equal to $\lfloor S(t)/\alpha(\Pi)\rfloor$ where $S(t)$ is the supply function of Π.*

Definition 2.26 *Virtual Time Scheduling on a partition Π is a scheduling constraint such that a job is eligible to run only when it is released and its release time is no less than the current virtual time of Π.*

Virtual Time Scheduling may apply to all of the scheduling algorithms designed for dedicated resources. Therefore, we have Virtual Time Rate Monotonic Scheduling, Virtual Time Earliest-Deadline-First Scheduling, and other virtual time scheduling algorithms.

Theorem 2.15 *A task group G $\{T_i = (c_i, p_i)\}_{i=1}^{n}$ is schedulable on a k-temporal-irregular partition Π by virtual time rate monotonic scheduling if $\sum_{i=1}^{n} \frac{c_i}{p_i - k} \leq \alpha(\Pi)n(2^{\frac{1}{n}} - 1)$.*

Proof Sketch: We construct a task group G' as c_i, p_i but with deadline of $p_i - k$ for each task. G' is schedulable on a regular partition Π' with $\alpha(\Pi') = \alpha(\Pi)$ using Deadline Monotonic scheduling algorithm[101]. Let us then schedule G' on Π using virtual time rate monotonic scheduling algorithm (which has the same priority order as Deadline Monotonic in this case). A job J' of Task T_i released at t_1 and finished before $t_1 + p_i - k$ when scheduled on Π' will be scheduled between t_1 and $t_1 + p_i$ when scheduled on Π. Therefore, G is schedulable. ■

Similarly, we have

Theorem 2.16 *A task group G $\{T_i = (c_i, p_i)\}_{i=1}^n$ is schedulable on a k-temporal-irregular partition Π by virtual-time earliest-deadline-first if $\sum_{i=1}^n \frac{c_i}{p_i - k} \leq \alpha(\Pi)$.*

Since earliest-deadline-first is an optimal scheduling algorithm we also have

Theorem 2.17 *A task group G $\{T_i = (c_i, p_i)\}_{i=1}^n$ is schedulable on a k-temporal-irregular partition Π by earliest-deadline-first if $\sum_{i=1}^n \frac{c_i}{p_i - k} \leq \alpha(\Pi)$.*

Chapter 3

Resource Level Scheduling

3.1 Introduction

In Chapter 2, we have proposed three models for real time virtual resource. For each of them, scheduling issues on the task level are discussed in depth. In this chapter, we switch our focus to the resource level and investigate the problem of how to schedule various real time virtual resources on a dedicated resource.

3.2 Problem Description

Given the resource requirement of (α_k,Δ_k) for each partition S_k, a schedule must be constructed at the resource level. Note that the pair of parameters (α_k,Δ_k) requires only that the partition must receive an α_k amount of processor capacity with the partition delay no greater than Δ. It does not impose any restriction on the execution time and period. This property makes the construction of the schedule extremely flexible. In this section, we first discuss the partition scheduling problem on a single level, i.e. scheduling partitions directly on a dedicated resource; then we extend this problem to multi-levels, i.e. scheduling partitions recursively on partitions.

3.3 Static Resource Level Scheduling

In this section we introduce a static resource level scheduling algorithm.

In this approach, the resource schedules every partition cyclically with a period equal to the minimum of T_k= $(\Delta_k/(1-\alpha_k))$ and each partition is allocated with an amount of processor capacity that is proportional to α_k. If the T_k of the partitions are substantially different, we may adjust them conservatively to form a harmonic chain in which T_j is a multiple of T_i, if $T_i < T_j$ for all i and j. This way, the static resource schedule is repeated during every major cycle which has a length equal to the maximum of T_k. Each major cycle is further divided into several minor cycles with a length equal to the minimum of T_k. This would reduce the number of context switches substantially[96, 90].

Static RLS is applicable to systems where all partitions are fixed and their parameters known. The essential idea of the algorithm is to compose the target partition by combining several partitions which are efficiently schedulable. Two theorems are involved to support constructing the scheduler.

Theorem 3.1 *Regular partitions whose availability factors are all powers of some number and whose total availability factor* ≤ 1.0 *are schedulable.*

Example 3.1 *Regular partitions* Π_i $(1 \leq i \leq 4)$ *with availability factors of 1/2, 1/4, 1/8, 1/8 respectively can be easily scheduled on a dedicated resource with the period of* 8 *and the time slot assignment of* $(1, 2, 1, 3, 1, 2, 1, 4)$ *where* i *indicates* Π_i.

Theorem 3.2 *When k partitions each with supply regularity of* 1 *are combined together they form a partition with supply regularity of k.*

Definition 3.1 *Given Partition* Π *with availability factor of* a *and supply regularity of* R_s*, the Adjusted Availability Factor* $AAF(a, R_s)$ *is the total of the availability factors of partitions that are used to compose* Π.

For a system with N partitions and the i_th partition having rate $= \alpha_i$ and supply regularity $= R_{si}$ $(1 \leq i \leq N)$, the partition table generation algorithm is summarized as following:

- For each partition Π_i, calculate its Adjusted Availability Factor $AAF(\alpha_i, R_{si})$ as following:

  If $R_{si} = 1$, $AAF(\alpha_i, R_{si}) = \frac{1}{2^k}$,where $k = \left\lfloor \log_{\frac{1}{2}} \alpha_i \right\rfloor$

  else $AAF(\alpha_i, R_{si}) = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + ... + \frac{1}{2^{i_{R_{si}}}}$

  where $i_1 = \left\lceil \log_{\frac{1}{2}} \alpha_i \right\rceil$, if $\log_{\frac{1}{2}} \alpha_i$ is not an integer,

  otherwise $i_1 = \left\lceil \log_{\frac{1}{2}} \alpha_i \right\rceil - 1$

  for $k \in [2, R_{si} - 1]$

  if $\log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}})$ is not an integer, $i_k = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}}) \right\rceil$

  otherwise,

  $i_k = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}}) \right\rceil - 1$, $i_{R_{si}} = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{R_{si}-1} \frac{1}{2^{i_z}}) \right\rceil$

  Record $2^{i_{R_{si}}}$ as P_i which denotes the partition period for Π_i.

- If $\sum_{i=1}^{N} AAF(\alpha_i, R_{si}) > 1.0$, program exits.

- Allocate M time slots where $M = max\{P_i \mid 1 \leq i \leq N\}$. M is partition table period.

- For each Π_i, $1 \leq i \leq k$, we have

  $AAF(\alpha_i, R_{si}) = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + ... + \frac{1}{2^{i_{R_{si}}}}$

  For each $k \in [1, R_{si}]$, among the whole M time slots, assign 1 out of every i_k time slots to Π_i.

- Assign the rest time slots to non-real-time tasks or the system partition.

- Combine all neighboring slots that are assigned to the same partition.

Partition_table[i]	Virtual_cpu_id	Start	End
0	AVP 1	0	1
1	AVP 3	1	2
2	AVP 1	2	3
3	AVP 2	3	4
4	AVP 1	4	5
5	AVP 3	5	6
6	AVP 2	6	7
7	SVP	7	8

Table 3.1: CPU Partition Table Generation Example

Example 3.2 *The following example shows how a partition table is generated for a system which has 3 AVPs with real-time requirements as* $\{(\alpha_i, R_{si}), i \in [1,3] \mid (0.375, 2), (0.25, 2), (0.25, 1)\}$.

- *Calculate* $AAF(\alpha_i, R_{si})$

$$AAF(\alpha_1, R_{s1}) = 0.25 + 0.125; \qquad P_1 = 8$$

$$AAF(\alpha_2, R_{s2}) = 0.125 + 0.125; \qquad P_2 = 8$$

$$AAF(\alpha_3, R_{s3}) = 0.25; \qquad P_3 = 4$$

- $\sum_{i=1}^{N} AAF(\alpha_i, R_{si}) = 0.875$.
- $M = max\{P_i \mid 1 \le i \le N\} = 8$.
- *Allocating time slots as shown in Figure 3.1.*
- *Combine the neighboring slots belonging to the same VP.*

3.4 Dynamic Resource Level Scheduling

In [115], a static resource level scheduling algorithm was proposed. This approach can efficiently schedule partitions with different delay requirements. However, it cannot accommodate partitions joining and leaving dynamically. To solve this problem, we introduce the concept of dynamic resource level scheduling (RLS).

Noting that similar dynamic task scheduling problem has been widely investigated by real-time systems researchers, we want to exploit past research by establishing a connection between resource scheduling and task scheduling. In this way, we may apply our results from task scheduling to resource scheduling.

Theorem 3.3 *Suppose the execution of a real-time task with computation time of C and period of P is considered as the execution of a partition. The resultant partition satisfies that $\alpha = C/P$ and $\Delta \leq 2P - 2C$ regardless of which scheduling algorithm is used.*

Proof: According to the definition of the rate as the percentage of usage, since the task uses C amount of time every P amount of time, the rate is C/P.

As for the partition delay, we first divide the interval duration into several ranges, then we derive the partition delay.

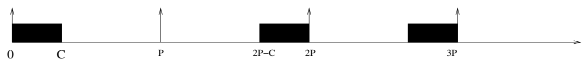


Figure 3.1: Worst Case Execution Scenario Case 1

As shown in Figure 3.1, two consecutive executions can be separated at most by $2P - 2C$, which is the longest unavailable time. Since the longest unavailable time is not always the same as the partition delay, we have to observe the supply with regard to intervals of different ranges.

To form the worst case scenario as shown in Figure 3.1, we assume that all executions for period $(n-1)P$ to nP $(n \geq 2)$ start at time $nP - C$ and finish at time nP. Given an interval (t_1, t_2),$(t_1 \leq t_2)$, let t_1 denote time C in Figure 3.1; we have

- When $0 \le t_2 - t_1 \le 2P - 2C$, i.e., $C \le t_2 \le 2P - C$, $(S(t_2) - S(t_1)) = 0$
- When $2P - 2C < t_2 - t_1 \le 2P - C$, i.e., $2P - C < t_2 \le 2P$, $(S(t_2) - S(t_1)) = t_2 - t_1 - (2P - 2C)$
- When $2P - C < t_2 - t_1 \le 3P - 2C$, i.e., $2P < t_2 \le 3P - C$, $(S(t_2) - S(t_1)) = C$
- ...
- When $nP - 2C < t_2 - t_1 \le nP - C$, i.e., $nP - C < t_2 \le nP$, $(S(t_2) - S(t_1)) = t_2 - t_1 - (nP - 2C) + (n-2)C$ where integer $n \ge 2$
- When $nP - C < t_2 - t_1 \le (n+1)P - 2C$, i.e., $nP < t_2 \le (n+1)P - C$, $(S(t_2) - S(t_1)) = (n-1)C$

Therefore, for any interval (T_1, T_2),$(T_1 \le T_2)$

$$\begin{aligned} & (C/P) \times (T_2 - T_1 - (2P - 2C)) \\ \le\ & (S(T_2) - S(T_1)) \\ \le\ & (C/P) \times (T_2 - T_1 + (2P - 2C)) \end{aligned}$$

$(C/P) \times (T_2 - T_1 - (2P - 2C)) = (S(T_2) - S(T_1))$ when $T_1 = C$, $T_2 = nP - C$ $(n \ge 2)$ as shown in Figure 3.1.

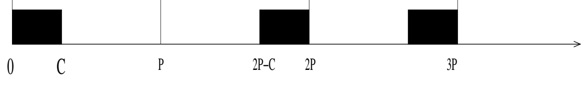


Figure 3.2: Worst Case Execution Scenario Case 2

$(S(T_2) - S(T_1)) = (C/P) \times (T_2 - T_1 + (2P - 2C))$ when $T_1 = P - C$, $T_2 = nP + C$, $n \ge 1$ as shown in Figure 3.2. ■

Corollary 3.1 *To schedule a partition* (α, Δ) *could be converted to schedule a task with* $(\Delta\alpha/(2(1-\alpha)), \Delta/(2(1-\alpha)))$ *as computation time and period, respectively.*

Example 3.3 *To schedule Partition* Π $(0.2, 40ms)$*,* $C = \Delta\alpha/(2(1-\alpha)) = 0.2 \times 40/(2 \times (1-0.2)) = 5ms$ $P = \Delta/(2(1-\alpha)) = 40/(2 \times (1-0.2)) = 25ms$ *Therefore, to schedule* Π *can be converted to schedule a task with computation time of 5ms and period of 25 ms.*

The admission test for new partitions could be expressed as $\sum_{i=1}^{n} \alpha_i \leq Utilization\ Bound$.

The utilization bound depends on which scheduling algorithm is used. It would be 1.0 for earliest deadline first (EDF) and $m(2^{\frac{1}{m}} - 1)$ for rate monotonic scheduling (RM) where m is the number of partitions.

3.4.1 Quantum Based Scheduling Consideration

In the discussion above, we assume that scheduling (especially task switching) can be performed with time being real numbers. However, many systems have a lower limit on the smallest time allocation unit, namely, the scheduling quantum. The scheduling quantum may also be viewed as the unit for specifying the precision of time measurements. It is imperative to consider the effects of the scheduling quantum when we implement the scheduling scheme on real systems. In this subsection we re-examine the issues that we discuss in the previous subsection and we also show the interdependence among rate, partition delay and the scheduling quantum.

Theorem 3.4 *To schedule a partition* (α, Δ) *on a system with scheduling quantum of* Q *is equivalent to schedule a task with* $(\lceil \Delta\alpha/(2Q(1-\alpha)) \rceil \times Q,$ $\lfloor \Delta/(2Q(1-\alpha)) \rfloor \times Q$ *as computation time and period, respectively.*

Because of the interdependence of rate, partition delay and the scheduling quantum that is discussed in [55], the introduction of the scheduling quantum also puts some constraints on rate and partition delay.

- $\alpha \geq 1/(1+\lfloor \Delta/Q \rfloor)$: The partition will receive at least one quantum of available time after the actual partition delay $\lfloor \Delta/Q \rfloor$ happens. This puts a lower bound on the rate which means that the partition rate could not be infinitely small.
- $\Delta \geq Q$: The partition delay should be no less than the scheduling quantum.

Again the admission test with quantum consideration could be expressed as following:

$$\sum_{i=1}^{n} \frac{\lceil \Delta\alpha/(2Q \times (1-\alpha)) \rceil \times Q}{\lfloor \Delta/(2Q \times (1-\alpha)) \rfloor \times Q} \leq Utilization\ Bound$$

3.4.2 Two-Level Resource Scheduling

When implementing RTVR, we observed that the lower bound on the rate could lead to inefficient usage of the resource. For example, the scheduling quantum of Linux was set as 10 ms, while we measured that a typical MP3 application should not be delayed more than 190 ms, which could be considered as its partition delay. Besides, the application needs less than 1 ms every 190 ms of time. However, we know from the relation between scheduling quantum and rate, that for such a partition delay to be achieved, the lowest rate is $(1/(1 + 190/10)) = 5\%$. The actual CPU usage of MP3 is just around 0.5%. Similarly, ISRs also have these characteristics. To solve this problem, we introduce a novel method of two-level partitioning.

The essential idea of this scheduling method is to group all those partitions with small partition delays as well as small rates together and consider them as one partition with a small partition delay but with a larger rate (the sum of the rates of the small partitions). When this partition is scheduled, it divides the scheduling quantum into mini time slots and distributes them among the original small partitions. A more detailed description is given below:

1. Partition Grouping: Small partitions (α_i, Δ_i) are grouped together as one partition Π (α, Δ) where $\alpha \geq \sum \alpha_i$ and $\Delta \leq \Delta_i - Q$.

2. Partition Scheduling: When a time slot assigned to Π, it will be split into N mini time slots. $\lceil (N\alpha_i/\alpha) \rceil$ mini time slots will be assigned to Partition Π_i. The time slots will run in the order of i.

3. Partition Admission: If there are n time slots unused among those N mini time slots. A new partition Π' (α', Δ') could be admitted by Partition Π if

   (a) $\Delta' \geq (\Delta + Q)$

   (b) $\alpha' \leq (n/N) \times \alpha$

4. Partition Leaving: If a partition leaves, instead of leaving a "hole" among the mini time slots, all partitions of the order higher (later) than this partition will run earlier than this partition. It also means all active partitions will run first and if there are mini time slots that are either unused or left by leaving partitions, they will always run latest among all the mini time slots. It is similar to the compression of mini time slots.

Example 3.4 *Schedule* Π_1 *(0.05, 40ms),* Π_2 *(0.02, 43ms),* Π_3 *(0.10, 48ms) using the two-level scheduling described above. Assume* $N = 10$.

1. *Group:* Π *(0.05+0.02+0.10, 40-10ms) = (0.17, 30ms). Because of the quantum size is 10ms and as we showed above that quantum size will put a lower bound to rate, the* Π *will get actually (.25, 30ms).*

2. *Scheduling: Every time slot of* Π *is further divided into 10 mini time slots as shown in Figure 3.3 (a) and (b),* Π_1 *then will get* $\lceil (10 \times 0.05/0.25) \rceil = 2$ *mini time slots as shown in Fig 3.3 (c). Similarly,* Π_2 *will get 1 slots and* Π_3 *4 slots. The remaining 3 slots could be used for non-real-time partition and could be*

assigned to new partition(s) arriving later. When Π *gets to run, the 2 time slots of* Π_1 *will run first, then the 1 time slot of* Π_2*, 4 slots of* Π_3*, finally the remaining 3 time slots.*

3. *Suppose a new partition* Π' *(0.06, 45ms) requests to join, because (1)* $45 > 30 + 10$*, (2)* $0.06 < 3/10 \times 0.25$*, it will be granted admission and* $\lceil(10 \times 0.06/0.25)\rceil = 3$ *will be assigned to* Π' *as shown in Figure 3.3-(d). Those time slots of* Π' *will run latest among all partitions because* Π' *joins the latest.*

4. *Suppose* Π_2 *leaves,* Π_3 *will run right after* Π_1 *instead of letting the resource idling for one mini time slot which was assigned to* Π_2*. Therefore, that one mini time slot will be postponed till the last of the 10 mini time slots as shown in Figure 3.3 (e).*

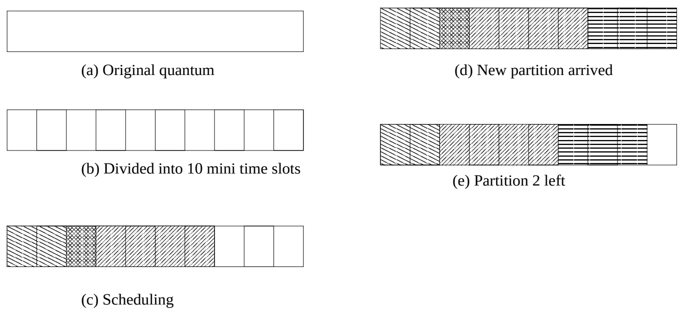

(a) Original quantum
(d) New partition arrived
(b) Divided into 10 mini time slots
(e) Partition 2 left
(c) Scheduling

Figure 3.3: Two Level Scheduling Example

Notice that this two-level resource scheduling method is different from the multi-level resource scheduling introduced in [55]. In [55], resource scheduling is recursively applied to partitions because large resources might need to be partitioned more than once. The two-level resource scheduling introduced here is specifically designed to address the problem of scheduling partitions with rates smaller than the lower bound enforced by scheduling quantum. As for scalability, this method is not as scalable as that in [55].

An interesting question is what if the OS does not support timer interrupts with a rate higher than the current OS clock frequency. This issue, however, does not pose a real problem because in practice the Real-Time clock of the computer hardware runs at a rate significantly higher than 100Hz, which is the typical frequency with the scheduling quantum size of 10ms. This huge frequency gap can be attributed to the scheduling overhead. Our technique exploits this property and pushes the utilization factor to a higher level.

3.5 Resource Level Scheduling for Regularity-Based Partitions

Since regularity-based partitions have unique properties because of their integer domain, we discuss their resource level scheduling issues separately from other partitions

3.5.1 Regular Partition

Theorem 3.5 *[138] A regular partition is uniquely determined by its availability factor except for the offset.*

Theorem 3.6 *Given a set* $\{n_i/p_i, 1 \leq i \leq m\}$ *as the availability factors of regular partitions, the decision problem of whether there exists a schedule containing all the partitions is NP-hard.*

Though the problem is NP-hard, we can always convert the availability factors into a schedulable set as long as the new availability factor is bigger than the old one. This method is similar to one solution of the pinwheel problem [72, 75].

Theorem 3.7 *Regular partitions whose availability factors are all powers of some number and whose total availability factor* ≤ 1.0 *are schedulable.*

Example 3.5 *Regular partitions* Π_i $(1 \leq i \leq 4)$ *with availability factors of 1/2, 1/4, 1/8, 1/8 respectively can be easily scheduled on a dedicated resource with the period of* 8 *and the time slot assignment of* $(1, 2, 1, 3, 1, 2, 1, 4)$ *where* i *indicates* Π_i.

Theorem 3.8 *Given a set* $\{n_i/p_i, 1 \leq i \leq m\}$ *as the availability factors of regular partitions, they are schedulable if* $\sum_{i=1}^{m} \frac{n_i}{p_i} \leq 0.5$.

Proof: For each n_i/p_i let $B_i = \frac{1}{2^j}$ where $\frac{1}{2^j} \geq \frac{n_i}{p_i} > \frac{1}{2^{j+1}}$. We have $B_i < 2 \times \frac{n_i}{p_i}$, therefore $\sum_{i=1}^{m} B_i < 1.0$. Because B_i consists solely of powers of the same base of 2 it is schedulable, hence the set of partitions is also schedulable. ■

A better bound is possible if we use the double-integer reduction technique [33] or if we focus on certain special cases. We do not cover these cases in this dissertation.

3.5.2 Irregular Partitions

Theorem 3.9 *When two regular partitions* Π_1 *and* Π_2 *from the same resource are combined together they form a new partition* Π_3 *with supply regularity of 2.*

Proof: Let $I_1(t)$, $I_2(t)$ and $I_3(t)$ denote the instant regularity functions of Π_1, Π_2 Π_3. $\forall a$,$\forall b$, $a < b$ We have

$$|I_1(b) - I_1(a)| < 1$$

$$|I_2(b) - I_2(a)| < 1$$

$$\begin{aligned} & |I_3(b) - I_3(a)| \\ = \; & |I_1(b) + I_2(b) - I_1(a) - I_2(a)| \\ \leq \; & |I_1(b) - I_1(a)| + |I_2(b) - I_2(a)| \\ < \; & 1 + 1 \\ = \; & 2 \; ■ \end{aligned}$$

The reason why we require that the two regular partitions are from the same resource is to ensure that they cannot possibly have a conflicting time slot with each another. Such conflicts may occur if the partitions are necessarily from different resources such as would be the case in a distributed environment. We do not address this issue any further in this dissertation.

Example 3.6 *When two regular partitions* Π_1 *and* Π_3 *in Example 3.5 are combined together, a new partition with availability factor of* $5/8$ *and supply regularity of* 2 *will be generated.*

Theorem 3.10 *When* k *regular partitions are combined together they form a partition with supply regularity of* k.

Theorem 3.11 *Given a set* $\{a_k, 1 \le k \le n\}$ *as the availability factors of 2-supply-irregular partitions, they are schedulable if* $\sum_{k=1}^{n} a_k \le 0.75$.

Proof: Let us rewrite each a_k as

$$a_k = \frac{1}{2^i} + \frac{x}{2^j}$$

Where $i = \left\lceil \log_{\frac{1}{2}} a_k \right\rceil$, $j = \left\lceil \log_{\frac{1}{2}} (a_k - \frac{1}{2^i}) \right\rceil$ and $x = \frac{(a_k - \frac{1}{2^i})}{\frac{1}{2^j}}$ when $(a_k - \frac{1}{2^i}) \neq 0$; $j = i + 1$ and $x = 0$ when $(a_k - \frac{1}{2^i}) = 0$.

Hence when $x \neq 0$, $1.0 \le x < 2.0$ and $i < j$.

We construct $b_k = \frac{1}{2^i} + \frac{1}{2^{j-1}}$ for each a_i. It is easy to see that $b_k \ge a_k$. Therefore, if b_k is schedulable a_k is also schedulable.

$$\begin{aligned}
\frac{a_k}{b_k} &= \frac{\frac{1}{2^i} + \frac{x}{2^j}}{\frac{1}{2^i} + \frac{1}{2^{j-1}}} \\
&= \frac{2^{j-i} + x}{2^{j-i} + 2} > \frac{2^{j-i} + 1}{2^{j-i} + 2} \\
&\ge \frac{2+1}{2+2} \\
&= 0.75
\end{aligned}$$

Therefore, $\sum_{k=1}^{n} a_k \le 0.75$ then $\sum_{k=1}^{n} b_k < 1$. ■

Example 3.7 *Let us consider the scheduling of three* 2*-supply-irregular partitions* Π_1*,* Π_2 *and* Π_3 *with* 0.36*,* 0.29*,* 0.08 *as their availability factors respectively. First, according to Theorem 3.11,* $0.36 + 0.29 + 0.08 = 0.73 < 0.75$*, therfore, they are schedulable. Second, as for how to construct the schedule, because* $1/4 + 1/16 < 0.36 < 1/4 + 1/8$ *we assign two regular partitions with availability factors of* $1/4$ *and* $1/8$ *to* Π_1*. For the similar reasons we assign* $1/4$ *and* $1/16$ *to* Π_2*,* $1/16$ *and* $1/32$ *to* Π_3*. It is easy to show that the total availability factor does not exceed 1. Hence, a valid schedule is constructed.*

Theorem 3.12 *Given a set* $\{a_i, 1 \leq i \leq n\}$ *as the availability factors of* $k+1$*-supply-irregular partitions, they are schedulable if* $\sum_{i=1}^{n} a_i \leq 1 - \frac{1}{2^{k+1}}$*.*

Proof: Let us rewrite each a_i

$$a_i = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + \cdots + \frac{1}{2^{i_k}} + \frac{x}{2^j}$$

where

$$i_1 = \left\lceil \log_{\frac{1}{2}} a_i \right\rceil$$

$$i_2 = \left\lceil \log_{\frac{1}{2}} (a_i - \frac{1}{2^i_1}) \right\rceil \text{ when } (a_i - \frac{1}{2^i_1}) \neq 0$$

$$\cdots$$

$$i_m = \left\lceil \log_{\frac{1}{2}} (a_i - \sum_{z=1}^{m-1} \frac{1}{2^i_z}) \right\rceil \text{ when } (a_i - \sum_{z=1}^{m-1} \frac{1}{2^i_z}) \neq 0$$

$$\cdots$$

$$i_k = \left\lceil \log_{\frac{1}{2}} (a_i - \sum_{z=1}^{k-1} \frac{1}{2^i_z}) \right\rceil \text{ when } (a_i - \sum_{z=1}^{k-1} \frac{1}{2^i_z}) \neq 0$$

$$j = \left\lceil \log_{\frac{1}{2}} (a_i - \sum_{z=1}^{k} \frac{1}{2^i_z}) \right\rceil \text{ when } (a_i - \sum_{z=1}^{k} \frac{1}{2^i_z}) \neq 0$$

$$x = \frac{(a_i - \sum_{z=1}^{k} \frac{1}{2^{i_z}})}{\frac{1}{2^j}} \text{ when } (a_i - \sum_{z=1}^{k} \frac{1}{2^i_z}) \neq 0$$

We have $1.0 \leq x < 2.0$ and $i_1 < i_2 < ... < i_k < j$ For each a_i we construct

$$b_i = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + \cdots + \frac{1}{2^{i_k}} + \frac{2}{2^j}$$

Then

$$\begin{aligned}
\frac{a_i}{b_i} &= \frac{\frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + \cdots + \frac{1}{2^{i_k}} + \frac{x}{2^j}}{\frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + \cdots + \frac{1}{2^{i_k}} + \frac{2}{2^j}} \\
&= \frac{\sum_{m=1}^{k} 2^{(j-i_m)} + x}{\sum_{m=1}^{k} 2^{(j-i_m)} + 2} \\
&= 1 - \frac{2-x}{\sum_{m=1}^{k} 2^{(j-i_m)} + 2} \\
&\geq 1 - \frac{2-x}{\sum_{m=1}^{k} 2^{m} + 2} \\
&> 1 - \frac{1}{2^{k+1}}
\end{aligned}$$

Therefore, $\sum_{i=1}^{n} a_i \leq 1 - \frac{1}{2^{k+1}}$then $\sum_{i=1}^{n} b_i < 1$. ■

Notice that both Theorem 3.11 and Theorem 3.12 only provide sufficient conditions for schedulability. Actually, as long as the total sum of the availability factors that are assigned to each partition does not exceed 1, these partitions are schedulable. Consider the Π_3 in Example 3.7. If we raise its requested availability factor 0.08 to as high as 0.31 which results in the total availability factor to be 0.96, the partitions are still schedulable. Here we only show the worst-case bound even though it is too pessimistic. We can give a tighter schedulability test as shown below.

Definition 3.2 *Let a denote the original availability factor and k the supply regularity. The Adjusted Availability Factor $AAF(a, k)$ is given by b_i as derived in Theorem 3.12.*

In Example 3.7, $AAF(0.36, 2) = 1/4 + 1/8 = 0.375$.

Theorem 3.13 *Given a set $\{a_i, 1 \leq i \leq n\}$ as the availability factors of k-supply-irregular partitions, they are schedulable if $\sum_{i=1}^{n} AAF(a_i, k) \leq 1$.*

3.5.3 Mixed Partitions

After considering the scheduling problem of partitions with the same supply regularity we give without proof some results on the problem of scheduling partitions with different supply regularities.

Theorem 3.14 *Given a set $\{(a_i, k_i), 1 \leq i \leq n\}$ as the availability factors and supply regularity of partitions, they are schedulable if $\sum_{i=1}^{n}(\frac{a_i \times 2^{k_i}}{2^{k_i}-1}) \leq 1$.*

Theorem 3.15 *Given a set $\{(a_i, k_i), 1 \leq i \leq n\}$ as the availability factors and supply regularity of partitions, they are schedulable if $\sum_{i=1}^{n} AAF(a_i, k_i) \leq 1$.*

Notice the difference of Theorem 3.14 and Theorem 3.15. Theorem 3.14 is derived from the worst case scenario and hence is pessimistic while Theorem 3.15 is derived from the general cases and hence gets more desirable results.

Example 3.8 *Let us consider the scheduling of four partitions Π_i, $1 \leq i \leq 4$ with $(0.43, 3)$, $(0.12, 1)$, $(0.31, 2)$, $(0.11, 2)$ as their availability factors and supply regularities respectively. According to Theorem 3.14,*

$$\begin{aligned} & \sum_{i=1}^{4}(\frac{a_i \times 2^{k_i}}{2^{k_i}-1}) \\ = \;& 0.43 \times 8/7 + 0.12 \times 2/1 \\ & +0.31 \times 4/3 + 0.11 \times 4/3 \\ \approx \;& 1.29 > 1 \end{aligned}$$

It appears that they are not schedulable. However, according to Theorem 3.15, we have

$$\begin{aligned} & \sum_{i=1}^{4} AAF(a_i, k_i) \\ = \;& AAF(0.43, 3) + AAF(0.12, 1) \\ & +AAF(0.31, 2) + AAF(0.11, 2) \end{aligned}$$

$$\begin{aligned} &= (1/4 + 1/8 + 1/16) + (1/8) \\ &\quad +(1/4 + 1/16) + (1/16 + 1/16) \\ &= 1 \end{aligned}$$

Therefore, they are in fact schedulable.

Chapter 4

Hierarchical Resource Level Scheduling

4.1 Multi-level Partition Scheduling

In the previous section, partitions are scheduled directly on top of a dedicated resource. In general, a partition may also reside directly inside another partition instead of a physical resource, and these partitions form a hierarchy. In this section, we first prove a schedulability result on hierarchical partitioning. Then we discuss how to perform the actual scheduling.

Theorem 4.1 *A partition group* $\{\Pi_i(\alpha_i, \Delta_i)\}$ $(1 < i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta)$ *if* $\sum_{i=1}^{n} \alpha_i \leq \alpha$ *and* $\Delta_i > \Delta$ *for all* i, $(1 < i \leq n)$.

Proof: To prove the theorem we first design a schedule, then we show that the resulting partitions satisfy the requirements.

Given any partition $\Pi(\alpha, \Delta)$, we divide every minimum time slice of Π into $n + 1$ parts. Since $\sum_{i=1}^{n} \alpha_i \leq \alpha$, each of the first n parts has α_i/α $(1 < i \leq n)$ of the time slice and is assigned to partition Π_i; the remaining $n + 1$st part has $1 - \sum_{i=1}^{n}(\alpha_i/\alpha)$ of the time slice and is not assigned to any partition. (If $1 - \sum_{i=1}^{n}(\alpha_i/\alpha) = 0$, this part has zero size.) Thus we have,

$$S_i(t) = \frac{\alpha_i}{\alpha} S(t)$$

for each partition Π_i.

Let us show why all the lower level partitions must satisfy the partition delay requirement.

Given any interval (t_1, t_2),$(t_1 < t_2)$ we have

$$\begin{aligned} & (t_2 - t_1 - \Delta)\alpha \\ \leq\ & (S(t_2) - S(t_1)) \\ \leq\ & (t_2 - t_1 + \Delta)\alpha \end{aligned}$$

$$\begin{aligned} & (t_2 - t_1 - \Delta)\alpha\frac{\alpha_i}{\alpha} \\ \leq\ & (S(t_2) - S(t_1))\frac{\alpha_i}{\alpha} \\ \leq\ & (t_2 - t_1 + \Delta)\alpha\frac{\alpha_i}{\alpha} \end{aligned}$$

$$\begin{aligned} & (t_2 - t_1 - \Delta)\alpha_i \\ \leq\ & (S_i(t_2) - S_i(t_1)) \\ \leq\ & (t_2 - t_1 + \Delta)\alpha_i \end{aligned}$$

$$\begin{aligned} & (t_2 - t_1 - \Delta_i)\alpha_i \\ \leq\ & (S_i(t_2) - S_i(t_1)) \\ \leq\ & (t_2 - t_1 + \Delta_i)\alpha_i \end{aligned}$$

Therefore, they satisfy the partition delay requirement. ■

Theorem 4.1 provides a method to determine the schedulability of scheduling partitions (a partition group) on another partition. However, it does not explain how to perform the actual scheduling since the infinite time slicing scheme that is used in the proof is impractical. Therefore, the question remains how to schedule partitions using methods with finite context switch overhead.

4.2 Partition Scheduling Algorithm

In this section, we propose a scheduling scheme that transforms the problem of scheduling partitions on partitions back to the one of scheduling partitions on resources. We first present the notion of virtual time scheduling; then we introduce the scheduling scheme and we also prove the validity of this scheme.

Theorem 4.2 *Given a partition group* $\{\Pi_i(\alpha_i, \Delta_i)\}$ $(1 < i \leq n)$ *to be scheduled on a partition* $\Pi(\alpha, \Delta)$*. Let* S_n *denote a scheduler of scheduling* $\Pi'_i(\alpha_i/\alpha, \Delta_i - \Delta)$ $(1 < i \leq n)$ *on a dedicated resource with capacity of the same as the normalized execution of* Π*. Also let* S_p *denote the virtual time* S_n *scheduler of scheduling* Π_i *on* Π*. Then* S_p *is valid if* S_n *is valid.*

Theorem 4.2 justifies the observation that we may use essentially the same algorithms of scheduling partitions on dedicated resources for hierarchical partitioning by applying the virtual time scheduling scheme. We describe this method in two steps:

- Construct Scheduler S_n.
- For any interval (t_1, t_2) assigned to Π'_i in S_n, assign $(P(t_1), P(t_2))$ to Π_i in S_p.

Informally, time on different levels may be expressed in the following way:

Actual time on higher partition:	t
Virtual time on higher partition:	$(t-\Delta, t+\Delta)$
Virtual time on lower partition:	$((t-\Delta-(\Delta_i-\Delta)),(t+\Delta+(\Delta_i-\Delta)))$
i.e.,	$(t-\Delta_i, t+\Delta_i)$

We present a formal proof below.

Proof: Let (t_1, t_2) denote an interval on Π_i' in S_n. and $t_1' = P(t_1)$ and $t_2' = P(t_2)$. Also let $S_{\Pi_i}(t)$ and $S_{\Pi_i'}(t)$ denote the supply functions for Π_i and Π_i' respectively. We have

$$\begin{aligned} & S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1) \\ = \; & S_{\Pi_i}(P(t_2)) - S_{\Pi_i}(P(t_1)) \\ = \; & S_{\Pi_i}(t_2') - S_{\Pi_i}(t_1') \end{aligned}$$

Therefore

$$\begin{aligned} & (t_2 - t_1 - (\Delta_i - \Delta))\alpha_i/\alpha \\ \leq \; & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1))/\alpha \\ \leq \; & (t_2 - t_1 + (\Delta_i - \Delta))\alpha_i/\alpha \end{aligned}$$

$$\begin{aligned} & (t_2 - t_1 - (\Delta_i - \Delta))\alpha_i \\ \leq \; & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1)) \\ \leq \; & (t_2 - t_1 + (\Delta_i - \Delta))\alpha_i \end{aligned}$$

$$\begin{aligned} & (V(P(t_2)) - V(P(t_1)) - (\Delta_i - \Delta))\alpha_i \\ \leq \; & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1)) \\ \leq \; & (V(P(t_2)) - V(P(t_1)) + (\Delta_i - \Delta))\alpha_i \end{aligned}$$

$$\begin{aligned}
& (S_{\Pi}(t_2')/\alpha - S_{\Pi}(t_1')/\alpha - (\Delta_i - \Delta))\alpha_i \\
\leq\ & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1)) \\
\leq\ & (S_{\Pi}(t_2')/\alpha - S_{\Pi}(t_1')/\alpha + (\Delta_i - \Delta))\alpha_i
\end{aligned}$$

$$\begin{aligned}
& ((t_2' - t_1' - \Delta)\alpha/\alpha - (\Delta_i - \Delta))\alpha_i \\
\leq\ & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1)) \\
\leq\ & ((t_2' - t_1' - \Delta)\alpha/\alpha + (\Delta_i - \Delta))\alpha_i
\end{aligned}$$

$$\begin{aligned}
& (t_2' - t_1' - \Delta - (\Delta_i - \Delta))\alpha_i \\
\leq\ & (S_{\Pi_i'}(t_2) - S_{\Pi_i'}(t_1)) \\
\leq\ & (t_2' - t_1' - \Delta + (\Delta_i - \Delta))\alpha_i
\end{aligned}$$

$$\begin{aligned}
& (t_2' - t_1' - \Delta_i)\alpha_i \\
\leq\ & (S_{\Pi_i}(t_2') - S_{\Pi_i}(t_1')) \\
\leq\ & (t_2' - t_1' - \Delta_i)\alpha_i
\end{aligned}$$

■

Again, we try to remove the virtual time scheduling by investigating its resource robustness.

Theorem 4.3 *Resource level scheduling is resource-burst-robust.*

Proof: By definition, a partition has to meet the requirement of $S(t) \geq (t - \Delta)\alpha$ for any t. After the resource advancement, $S'(t) \geq S(t) \geq (t - \Delta)\alpha$ for any t. The requirement still holds. ■

Therefore, we have

Theorem 4.4 *Given a partition group* $\{\Pi_i(\alpha_i, \Delta_i)\}$ $(1 < i \leq n)$ *to be scheduled on a partition* $\Pi(\alpha, \Delta)$. *If* $\Pi'_i(\alpha_i/\alpha, \Delta_i - \Delta)$ $(1 < i \leq n)$ *are schedulable on on a dedicated resource with capacity of the same as the normalized execution of* Π, *then they are also schedulable on* Π.

Example 4.1 *Let us schedule Partitions* Π_1 $(0.2, 5)$, Π_2 $(0.25, 6)$ *and* Π_3 $(0.05, 8)$ *on Partition* $\Pi(0.5, 4)$. *Using this method, the scheduling problem is transformed into scheduling*

$$(0.2/0.5, 5 - 4), (0.25/0.5, 6 - 4), (0.05/0.5, 8 - 5)$$

i.e.,

$$(0.4, 1), (0.5, 2), (0.1, 4)$$

on a dedicated resource. Suppose we use the simplest half-half algorithm describe in [116]. This problem may be further converted into an equivalent problem of scheduling three tasks with (period, computation time) given by, respectively,

$$(0.5, 0.2), (1, 0.5), (2, 0.2)$$

Obviously, they are schedulable.

4.3 Partition Scheduling with Quantum Size Requirements

So far, we have discussed the partition scheduling problem with regarding to bounded delay resource partition assuming that context switching may occur at any time. In practice, this assumption, however, might not hold because the nature of the resource may necessitate a quantum-size requirement to enforce atomicity in sharing.

In fact, it may be argued that all discrete-time computing devices have quantum-size requirements since on the micro-instruction level, machine instructions are not interruptible, thus prohibiting context switch from occurring at those time points. Therefore, in this section, we investigate the partition scheduling problem with quantum size requirements. We first review some notions concerning regularity partitions. Then we look at the scheduling problem where all partitions have the same quantum size and finally we study partitions having different quantum sizes.

Definition 4.1 *The quantum size Q of a partition Π is the smallest length of time of which any continuous available interval of Π is a multiple.*

Definition 4.2 *An Extended Bounded Delay (EBD) partition with quantum Π is a tuple (α, Δ, Q), $(0 < \alpha \leq 1, \Delta \geq 0, Q \geq 0)$, where α is the availability factor of the partition, Δ is the partition delay and Q is the partition quantum. When $Q = 0$, Π is called a real-number partition that (in this special case) allows infinite time-slicing.*

Example 4.2 *EBD Partition Π (1/6, 5, 1) may be considered as a Bounded Delay Partition (1/6,5) with quantum size of 1.*

The difference of the partition scheduling problem for EBD partitions from Bounded Delay partitions is that for EBD partitions context switching may occur only at certain discrete time points, as opposed to any time point for the case of real-number partitions.

Definition 4.3 *Integer EBD Partitions are EBD Partitions with quantum size of 1.*

There is an upper bound restriction on the quantum size.

Theorem 4.5 *The maximum unavailable time interval of Partition Π (α, Δ) is Δ and the maximum available time interval is $(\frac{\alpha}{(1-\alpha)}\Delta)$.*

Proof: Consider an available time interval and an adjacent unavailable time interval of Partition Π as shown in Figure 4.1. Let t_1 and t_2 denote the start point and the end point of the available time intervals respectively. Let t_3 denote the end point of the unavailable time interval. According to the definition of the delay bound,

$$(S(t_3) - S(t_2)) \geq (t_3 - t_2 - \Delta)\alpha$$

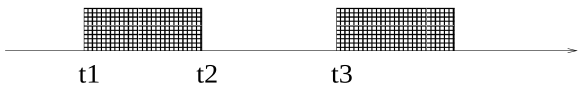


Figure 4.1: Quantum Size Limit

We show that $t_3 - t_2 \leq \Delta$ because otherwise

$$S(t_3) - S(t_2) = 0$$

while

$$(t_3 - t_2 - \Delta)\alpha > 0$$

Contradiction. Similarly we have

$$(S(t_2) - S(t_1)) \leq (t_2 - t_1 + \Delta)\alpha$$

$$(t_2 - t_1) \leq t_2 - t_1 + \Delta)\alpha$$

$$(t_2 - t_1) \leq (\frac{\alpha}{(1-\alpha)}\Delta)$$

■

The maximum available time interval is actually the maximum quantum size for partitions with quantum requirement. Therefore, we have

Corollary 4.1 *The maximum quantum size of Partition* Π (α, Δ) *is* $(\frac{\alpha}{(1-\alpha)}\Delta)$.

For the rest of the dissertation, we assume that the quantum size values are all within the proper bounds. Now we turn to partition scheduling problem with quantum size requirements. We first focus on the scheduling problem of Integer EBD partitions and later extend the results to General EBD partitions.

Theorem 4.6 *EBD Partition group* $\{\Pi_i(\alpha_i, \Delta_i, Q)\}$ $(1 < i \leq n)$ *is schedulable on a dedicated resource if and only if EBD Partition group* $\Pi_i(\alpha_i, \Delta_i/Q, 1)$ $(1 < i \leq n)$ *is also schedulable on a dedicated resource.*

Proof: We first construct a mapping between these two schedules and then show they are essentially equivalent. We label the schedule for Π_i as S_0 and the schedule for Π_i' as S_1. We use $S_0(t)$ and $S_1(t)$ to denote their supply functions respectively. Since they both have specific quantum sizes, we only consider those time points that are multiple of Q for S_0 and integers for S_1. To show the necessity, given S_0 we construct S_1 as follows: Let n denote a non-negative integer. Construct $(n, n+1)$ in S_1 according to $(nQ, (n+1Q))$ in S_0, i.e., if during interval $(nQ, (n+1))Q$ in S_0 the resource is available (or unavailable) to Π_i, it is also available (or unavailable) to Π_i' during interval $(n, n+1)$ in S_1. Since every Q units of supply in S_0 is mapped to every 1 unit of supply in S_1, we have $S_1(nQ) = S_0(n)Q$. Therefore, for any interval (n_0, n_1) in S_1 we have

$$\begin{aligned} & (Qn_1 - Qn_0 - \Delta_i)\alpha_i \\ \leq\ & S_0(Qn_1) - S_0(Qn_0) \\ \leq\ & (Qn_1 - Qn_0 + \Delta_i)\alpha_i \end{aligned}$$

$$\begin{aligned} & (Qn_1 - Qn_0 - \Delta_i)\alpha_i \\ \leq\ & S_1(n_1)Q - S_1(n_0)Q \\ \leq\ & (Qn_1 - Qn_0 + \Delta_i)\alpha_i \end{aligned}$$

$$\begin{aligned}
& (n_1 - n_0 - \Delta_i/Q)\alpha_i \\
\leq\ & S_1(n_1) - S_1(n_0) \\
\leq\ & (n_1 - n_0 + \Delta_i/Q)\alpha_i
\end{aligned}$$

Therefore S_1 is a valid schedule for Π'_i, i.e., partition group $\Pi_i$is also schedulable on a dedicated resource. Every step above is reversible, and hence sufficiency also holds. ■

Theorem 4.7 *A partition group which consists of integer partitions* $\Pi_i(\alpha_i, \Delta_i, 1)$ $(1 < i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta, 1)$ *if partition group* $\Pi'_i(\alpha_i/\alpha, (\Delta_i - \Delta)\alpha, 1)$*are schedulable on a dedicated resource.*

Proof: Since the Π'_i are schedulable on a dedicated resource, according to Theorem 4.6, it follows that $(\alpha_i/\alpha, (\Delta_i - \Delta), 1/\alpha)$ $(1 < i \leq n)$ are schedulable on a dedicated resource. We use S to denote a valid schedule for it.

Using the partition scheduling method described in Section 4.2 to construct a schedule on partition Π, we have a schedule of partitions Π_i on a partition $\Pi(\alpha, \Delta, 1)$. Since S is constructed on a dedicated resource but with a lower speed of the resource where Π resides and the ratio of this lower speed to the speed of the dedicated resource is α, α units of time in S will be mapped to 1 unit of time in Π. Therefore, the quantum size also conforms to the requirements. ■

Theorem 6.1 also transforms the scheduling problem of partitions within a partition back to the problem of partitions on a dedicated resource.

Example 4.3 *Consider scheduling Partitions* $\Pi_1(1/2, 1, 1)$ *and* $\Pi_2(1/6, 5, 1)$ *on Partition* $\Pi_3(2/3, 1, 1)$*. Obviously, they are schedulable while* $((1/2)/(2/3), 1 - 1, 1) = (3/4, 0, 1)$ *is not a schedulable partition on a dedicated resource.*

Similar to context switching, partition delay may possibly occur at any time interval for real-number partitions while for partitions with non-zero quantum size it may possibly occur only at intervals that both start and end at quantum boundary time points. We do not explore much about this property in scheduling methods in this dissertation because we want to put more emphasis on achieving complete isolation between different partition levels and also on designing a clean model to capture the overall characteristic of partitions.

We can easily extend Theorem 6.1 to partitions with different quantum sizes on different levels.

Corollary 4.2 *A partition group* $\{\Pi_i(\alpha_i, \Delta_i, Q')\}$ $(1 < i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta, Q)$ *if* Q *is a multiple of* Q' *and partition group* $\Pi'_i(\alpha_i/\alpha, Q'/Q(\Delta_i - \Delta)\alpha, 1)$ *is schedulable on a dedicated resource.*

Now we consider partitions with different quantum size requirements.

Theorem 4.8 *A partition group* $\{\Pi_i(\alpha_i, \Delta_i, Q_i)\}$ $(1 < i \leq n)$ *is schedulable on a partition* $\Pi(\alpha, \Delta, Q)$ *if* Q *is a common multiple of* $Q_1, Q_2, ...Q_n$ *and partition group* Π'_i $(\alpha_i, \Delta_i, LCM(Q_i))$ *is schedulable on* Π *where* $LCM(Q_i)$ *is the least common multiple of* $Q_1, Q_2, ...Q_n$.

Proof sketch: Since $LCM(Q_i)$ is a multiple of any Q_i, quantum size of $LCM(Q_i)$ also meets the requirement of quantum size of Q_i. ■

In Theorem 4.8 for cases where n is large, $LCM(Q_i)$ may be a huge number, thus rendering Π'_1 unschedulable on Π. Since the tasks may be scheduled not only in lowest level partitions but also in other intermediate ones, they may be scheduled as if the partitions were all on a single flat level. Considering this property and Corollary 4.2, we may use multi-level partition (vertical) scheduling to simulate single-level (horizontal) scheduling.

We suggest the following method:

Suppose we are given a partition group $\Pi_i(\alpha_i, \Delta_i, Q_i)$ $(1 < i \leq n)$ to be scheduled on $\Pi(\alpha, \Delta, Q)$ and Q is a multiple of $LCM(Q_i)$.

- Step 1: Compute the *divides* relation of the set containing $LCM(Q_i)$ and the Q_is. Represent this relation as a graph.
- Step 2: Starting from the top level of the graph which is $LCM(Q_i)$, schedule one level at a time using the quantum size one level up.
- Step 3: Repeat Step 2 down to lower levels till all partitions are scheduled or till they are found to be unschedulable.

We use an example to illustrate the process:

Example 4.4 *Consider scheduling partitions* Π_i $(1 \leq i \leq 5)$ *with quantum size* $(6, 4, 3, 2, 1)$ *respectively on a partition with quantum size of* 12.

First, we compute the $LCM(Q_i) = 12$ *and draw the graph of the divides relation for the set* $(12, 6, 4, 3, 2, 1)$

Second, we start from the first level which is 12*. Since there is no partition corresponding to this size we skip it.*

Third, we go down to the level of size 6 *and* 4 *where* Π_1 *and* Π_2 *are about to be scheduled. We use the quantum size of one level up which is* 12 *to schedule* Π_1 *and* Π_2*.*

Fourth, we use the unscheduled part from above which is also a partition with quantum size of 12 *to schedule the lower level that consists of* Π_3 *and* Π_4*. Again we use quantum size of one level up which is* 6*.*

Finally, to schedule Π_5 *on the lowest level in the unscheduled part from the previous step, we simply use quantum size of* 1 *since there is only one partition yet to be scheduled.*

One advantage of this method is that instead of promoting the quantum size of all partitions to the *LCM* we only promote them to at most one level up. Therefore, the chance of schedulability is greatly enhanced.

For the scheduling of more partitions, we may also consider bottom-up approach.

To compute the partition of the unscheduled part on a dedicated resource, we have the following theorem.

Theorem 4.9 *Given a partition group* $\{\Pi_i(\alpha_i, \Delta_i, Q)\}$ $(1 \le i \le n)$ *that is scheduled on a dedicated resource with quantum size of* Q *and* $\sum_{i=1}^{n}(\alpha_i) < 1$*, the unscheduled part on the resource forms a partition* $\Pi_0((1 - \sum_{i=1}^{n}(\alpha_i)), (\sum_{i=1}^{n}(\alpha_i \Delta_i))/(1 - \sum_{i=1}^{n}(\alpha_i), Q)$ *.*

Proof: It is obvious that the availability factor and the quantum size of Π are $(1 - \sum_{i=1}^{n}(\alpha_i))$ and Q respectively. We focus on the partition delay. Let use $S_i(t)$ $(0 \le i \le n)$ denote the supply functions of Π_i $(0 \le i \le n)$. Since they are scheduled on a dedicated resource which has supply function $S(t) = t$

$$\sum_{i=0}^{n}(S_i(t)) = S(t) = t$$

$$S_0(t) = t - \sum_{i=1}^{n}(S_i(t))$$

Therefore, for any time interval(t_1, t_2), we have

$$\begin{aligned} & S_0(t_2) - S_0(t_1) \\ = \;& (t_2 - \sum_{i=1}^{n} S - i(t_2)) - (t_1 - \sum_{i=1}^{n} S_i(t_1)) \\ = \;& t_2 - t_1 - (\sum_{i=1}^{n}(S_i(t_2) - S_i(t_1))) \end{aligned}$$

Therefore,

$$\begin{aligned}
&(t_2 - t_1 - \sum_{i=1}^{n}(\alpha_i(t_2 - t_1 + \Delta_i)) \\
\leq\ & (S_0(t_2) - S_0(t_1)) \\
\leq\ & (t_2 - t_1 - \sum_{i=1}^{n}(\alpha_i(t_2 - t_1 - \Delta_i))
\end{aligned}$$

$$\begin{aligned}
&((1 - \sum_{i=1}^{n}\alpha_i)(t_2 - t_1) - \sum_{i=1}^{n}(\alpha_i\Delta_i) \\
\leq\ & (S_0(t_2) - S_0(t_1)) \\
\leq\ & ((1 - \sum_{i=1}^{n}\alpha_i)(t_2 - t_1) + \sum_{i=1}^{n}(\alpha_i\Delta_i)
\end{aligned}$$

$$\begin{aligned}
&((1 - \sum_{i=1}^{n}\alpha_i)(t_2 - t_1 - \sum_{i=1}^{n}(\alpha_i\Delta_i)/((1 - \sum_{i=1}^{n}\alpha_i)) \\
\leq\ & (S_0(t_2) - S_0(t_1)) \\
\leq\ & ((1 - \sum_{i=1}^{n}\alpha_i)(t_2 - t_1 + \sum_{i=1}^{n}(\alpha_i\Delta_i)/((1 - \sum_{i=1}^{n}\alpha_i))
\end{aligned}$$

■

Similarly, we can extend the result to the unscheduled part on a partition.

Theorem 4.10 *Given a partition group* $\{\Pi_i(\alpha_i, \Delta_i, Q)\}$ $(1 \leq i \leq n)$ *that is scheduled on a partition* $\Pi(\alpha, \Delta, Q)$*and* $\sum_{i=1}^{n}(\alpha_i) < \alpha$*, the unscheduled part of* Π *forms a partition* $\Pi_0((\alpha - \sum_{i=1}^{n}(\alpha_i)), (\sum_{i=1}^{n}(\alpha_i\Delta_i) + \alpha\Delta)/(\alpha - \sum_{i=1}^{n}(\alpha_i), Q)$.

Chapter 5

Distributed and Parallel RTVR

5.1 Introduction

In previous chapters, we discussed the concept of RTVR only in the context of single physical resource. In the first part of this chapter, we extend RTVRs to parallel environments then in the second part of this chapter to distributed environments.

5.2 Parallel RTVRs

Parallel computing has long been an important research area. Recent years have witnessed the increasing availability of small-scale multiprocessors and processor clusters and the burgeoning applications in signal processing, video games and physical simulations where tasks are run in parallel to meet timing requirements. Examples range from radar systems with parallel DSPs to servers providing wireless services [93].

A well-known issue in parallel computing is that the performance of parallel applications may be substantially degraded unless the scheduling of the processes of a parallel application is coordinated so that the execution of the processes maintains an appropriate phase relation with respect to one another. This is because the processes of a parallel task (group) usually have control and/or data dependencies; if the start/stop of one process is temporally misaligned with other processes, the other processes may have to wait (sometimes wasting cycles by spinning) to get back into alignment in order to proceed, thus lengthening the completion time of the entire parallel task. Maintaining temporal alignment is also important for performance improvement because of more efficient communication among processes and better utilization of shared caches [54, 84].

Currently, the most popular scheme for coordinated scheduling is gang scheduling [54]. In gang scheduling, all the processes of a parallel task (group) are grouped into a "gang" and are scheduled to run in parallel on the different PEs (Processing Elements) of a multiprocessor. In order to maintain temporal alignment, the concurrent preemption and subsequent resumption of all the processes of the gang on the same set of PEs is supported. This technique ensures high parallelism and throughput for parallel tasks while providing high overall system responsiveness and utilization. The gang scheduling technique has been widely adapted in high performance systems such as IBM RS/6000 [52] and is supported by next-generation operating systems such as the K42 operating system under development at IBM. The K42 is a new high performance, open source, general-purpose operating system kernel for multiprocessors [153]. Both real time tasks (hard real time and soft real time) and gang-scheduled tasks are considered in the K42, together with two other task types (general purpose and background tasks). In real-time systems research, Lee et all [93] proposed an indicator called "task utilization workload" to estimate the amount of time that non-real-time tasks could be scheduled before real-time tasks would miss their deadlines. Silva et al [142] have studied the worst-case response time of gang scheduling in the real-time environment.

For these reasons, we first study the problem of scheduling real time parallel tasks together with real time sequential tasks (we refer to the latter as local tasks since each of them is scheduled only on a pre-assigned processor). In our approach, we aim to "seamlessly" integrate parallel tasks with local tasks exploiting the isolation property of RTVR. Specifically, we assign separate partitions to the parallel and local tasks at the first partition level. The clean isolation between different scheduling levels in RTVRs enables parallel task groups on every PE to automatically synchronize with one another as long as (1) the partition parameters and scheduling algorithm on the first partition level (the level closest to the physical resource) are the same on every PE, and (2) the clocks of the PEs are perfectly synchronized with respect to one another. (Imperfect clock synchronization in practice can be taken into account by adding the effect of the clock skew to the jitter parameters of the partitions; we do not elaborate the details for ease of explanation.) Lower level partitions may be further partitioned to accommodate parallel tasks requiring different numbers of PEs and those tasks requiring application isolation. Under this architecture, both parallel tasks and local tasks can be efficiently admitted using the same admission control schemes as in Chapter 2.

5.3 Parallel RTVRs

The essential idea of RTVR solution is to enable all PEs to automatic synchronize the execution of the parallel partition by using identical settings on the first level partitioning on every PE. The detailed description is given below and shown in Figure 5.1:

- On each PE group all parallel tasks as a parallel task group and all local tasks as a local task group.
- On each PE compute a set of partition parameters α_i and Δ_i so that its local task group is schedulable on such a partition.

- Choose the largest partition rate α_{max} among α_i and smallest partition delay Δ_{min} among Δ_i as the local partition parameters for all PEs.
- Compute another set of partition parameters α_p and Δ_p for all PEs so that the parallel task group is schedulable on $\Pi_{parallel}$ (α_p, Δ_p).
- Schedule two partitions Π_{local} $(\alpha_{max}, \Delta_{min})$ and $\Pi_{parallel}$ (α_p, Δ_p) on the first level (the level closest to physical resources) in the RTVR partition hierarchy.
- On each PE assign Π_{local} to schedule its local task group and assign $\Pi_{parallel}$ to schedule its parallel task group.
- Choose a same scheduling algorithm for the first level partitioning on each PE assuming Π_{local} and $\Pi_{parallel}$ are schedulable.
- Choose a same scheduling algorithm to schedule parallel tasks in the parallel task group.
- Start all PEs at the same time.

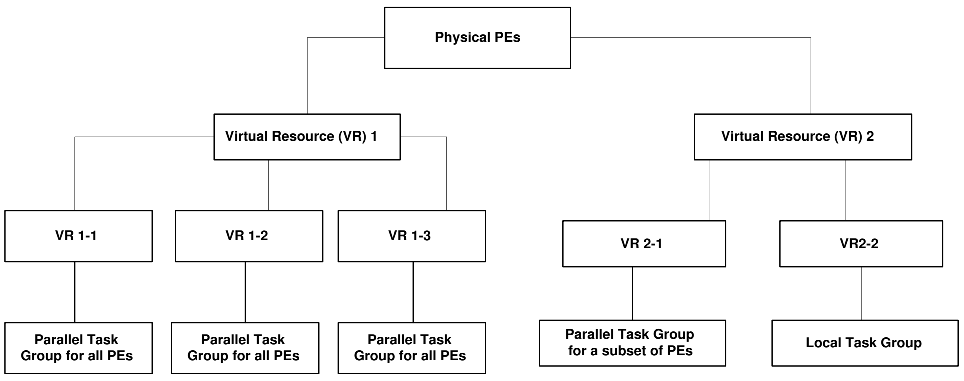

Physical PEs
Virtual Resource (VR) 1
Virtual Resource (VR) 2
VR 1-1
VR 1-2
VR 1-3
VR 2-1
VR2-2
Parallel Task Group for all PEs
Parallel Task Group for all PEs
Parallel Task Group for all PEs
Parallel Task Group for a subset of PEs
Local Task Group

Figure 5.1: Parallel Task Structure

Theorem 5.1 *The scheduling algorithm described above generates scheduling that executions of the parallel task group on all PEs are synchronized and neither parallel task nor local task would miss its deadline.*

Proof Sketch: To show why each processor would run the parallel tasks at the same time:

- Because on RTVRs changing parameters on a lower level does not affect higher level scheduling the difference of local tasks among PEs is contained in levels that are lower than the first one.
- On the first partition level every PE has identical settings that include:
  - Same number of partitions.
  - Same partition parameters.
  - Same resource level scheduler.
  - Same task level scheduler used inside parallel task group.

Parallel tasks and local tasks would not miss their deadlines because they are schedulable on their own partitions that in turn are schedulable on each PE. ■

Let us better illustrate the solution by showing an example.

Example 5.1 *Consider scheduling two parallel tasks T_{p1} $(1,3)$ and T_{p2} $(1,5)$ on two processors τ_1 and τ_2. There is a local task T_1 $(1,7)$ running on τ_1 and another local task T_2 $(2,10)$ running on τ_2. The scheduling approach is applied in the following steps:*

- *Group both parallel tasks (T_{p1} and T_{p2}) together and assign them a partition with parameters that satisfy*

  $$\sum_{i=1}^{2} C_{pi}/(P_{pi} - \Delta) \leq \alpha$$

  i.e.

  $$1/(3 - \Delta) + 1/(5 - \Delta) = \alpha$$

  We choose Δ as 1 and thus α as 0.75.

- *Because the parallel partition utilizes* 75% *of CPU, the utilization factor of the partition for local task could be at most* 25%. *Let* Δ_1 *and* Δ_2 *denote the maximum partition delay for the local partition on* τ_1 *and* τ_2. *On* τ_1, *computing* Δ_1 *from* $1/(7 - \Delta_1) = 0.25$ *results in* Δ *as* 3. *Similarly, on* τ_2, $2/(10 - \Delta_2) = 0.25$ *results in* Δ_2 *as* 2.

- *Choose the smallest value from* Δ_1 *and* Δ_2 *above as the partition delay of local partition so that the local task(s) on each processor could be schedulable. In the example,* 2 *is chosen as the local partition's partition delay.*

- *Finally, for either processor, parallel tasks are scheduled on Partition* $\Pi_{parallel}$ $(0.75, 1)$ *and the local task is scheduled on Partition* Π_{local} $(0.25, 2)$.

5.3.1 Solution Generalization

The result could be extended in the following directions:

- There could be arbitrary number of partitions on the first level as long as the settings with regard to those aspects mentioned above on that level are the same on each PE.

- Parallel partitions could also be further partitioned as long as same partitioning scheme and parameters are the same on each participating processor.

- Local partitions could also be recursively partitioned as described in [55].

- If there are parallel tasks that run on different numbers of PEs. In a system with 10 processors, there are parallel tasks requiring all 10 processors and there are also parallel tasks requiring only 6 processors. In this case, the first level local partition of those 6 processors could be further partitioned into a second level parallel partition and a second level local partition as shown in Figure 5.2.

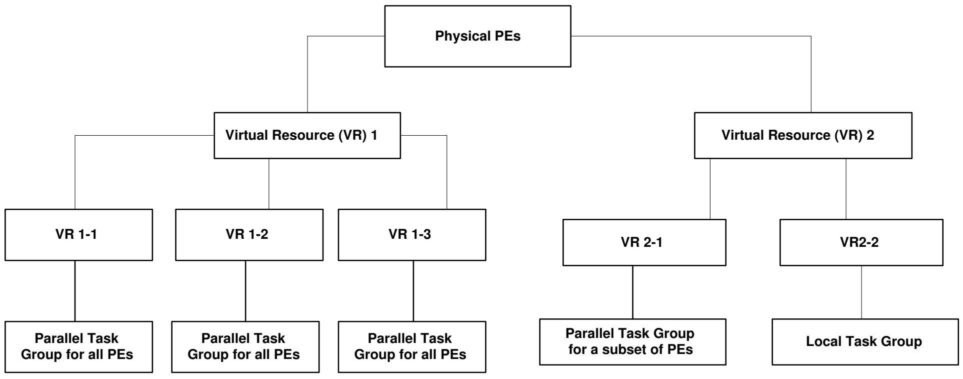

Physical PEs
Virtual Resource (VR) 1
Virtual Resource (VR) 2
VR 1-1
VR 1-2
VR 1-3
VR 2-1
VR2-2
Parallel Task Group for all PEs
Parallel Task Group for all PEs
Parallel Task Group for all PEs
Parallel Task Group for a subset of PEs
Local Task Group

Figure 5.2: General Parallel Task Structure

5.3.2 Admission Control

Since we study the scheduling problem in open systems in which tasks may join and leave dynamically, the admission control is an important issue. In order to admit parallel tasks or local tasks, the admission control schemes proposed in Chapter 2 are directly applicable.

5.4 Distributed RTVRs

Advances in computing applications in recent years have prompted the demand for distributed computing. Distributed computing is the processing of a data set by allocating the data and resources among multiple computers, ranging from loosely coupled computers through networks to tightly coupled computers such as SMP. Distributed computing in many ways is replacing the need for supercomputers for most applications by bringing more speed at a much less expensive price. In this section, we focus on how the concept of RTVR can be utilized by process migration, an important aspect of distributed computing.

5.4.1 Process Migration

Process migration is the act of transferring a process among different machines. It enables dynamic load distribution, fault resilience, eased system administration, and data access locality. With the increasing deployment of distributed systems in general, and distributed operating systems in particular, process migration is again receiving more attention in both research and product development. As high-performance facilities shift from supercomputers to networks of workstations, and with the ever-increasing role of the World Wide Web, we expect process migration to play a more important role and eventually to be widely adopted.

Theorem 5.2 *A RTVR Group consisting multi-hops of RTVRs* $\Pi_i(\alpha_i, \Delta_i)$ $1 \leq i \leq n$ *has a maximum end-to-end delay of* $\sum_i^n \Delta_i$.

Proof Sketch: At each hop Π_i, at most Δ_i delay may occur. Therefore, for all hops, at most $\sum_i^n \Delta_i$ may occur. ■

Theorem 5.2 provides an upper bound for the partition delay in distributed environments. The bound, however, is quite pessimistic and may be improved in future work.

Chapter 6

Resource Locking in RTVR

6.1 Introduction

An important issue of RTVRs that remains unsolved is the resource locking problem. In previous literature [116, 115, 55], all tasks are assumed to be *independent*. However, it is very common for tasks to share some mutually exclusive resources like shared memory. In this section, we first consider resource locking issues by tasks from a same partition (intra-partition locking), then we focus on resource sharing by tasks from different partitions (inter-partition locking).

6.1.1 Intra-Partition Locking

Intra-partition locking is similar to the locking on a dedicated processor. Hence, the results of traditional Priority Inheritance or Priority Ceiling could be easily extended to this scenario. For example, let B_i $(1 \leq i \leq n)$ denote the maximum blocking time experienced by task T_i (C_i, P_i) $(1 \leq i \leq n)$, to decide the schedulability of T_i using Priority Inheritance Protocol or Priority Ceiling Protocol and dynamic task scheduling on partition Π (α, Δ), the admission control takes the following form:

$$\sum_{j=1}^{i} C_j/(P_j - \Delta) + B_i/(P_i - \Delta) \leq \alpha$$

6.1.2 Inter-Partition Locking

Inter-partition locking without any extra enforcement might cause serious problems. For example, Tasks T_1 and T_2 belong to different task groups and in turn different partitions and they both access a mutually exclusive resource R_1. When T_1 locked R_1 and is running inside the critical section, T_2 tries to lock R_1 at the same time and becomes blocked. T_2 might experience unbounded delay in cases like (1) T_1 overruns inside the critical section or (2) another task in the same partition with T_1 preempts T_1 and then fails. The blocking chain would go further. Notice that T_2 belongs to a different partition from T_1 and is supposed to be isolated from failures of other partitions. Hence, the application isolation property of RTVRs is violated due to this type of locking.

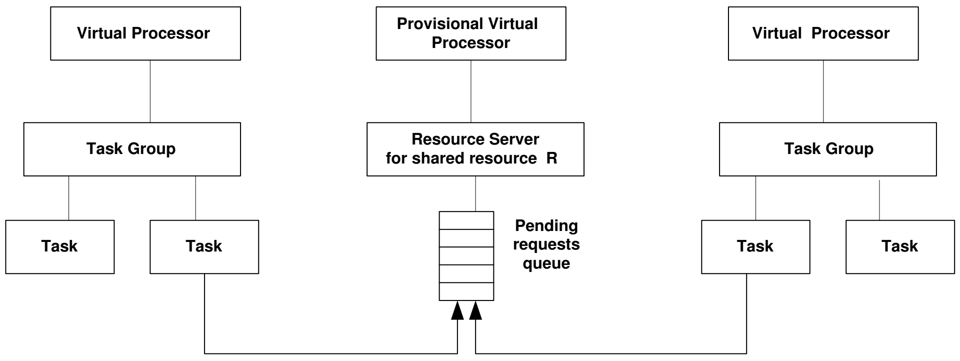


Figure 6.1: Inter Partition Resource Locking Solution

To solve this problem, we were inspired by the work on locking in Proportional-Fair systems [71] and propose a resource sever solution described below. For simplicity, we also assume that there are no nested critical sections and thus no deadlock will occur.

- For each sharing resource R_i $(1 \leq i \leq m)$, a separate partition is created as the resource server of R_i. We use S_i to denote the server partition. All requests for accessing R_i will be directed to S_i and will be processed in FCFS fashion.

- Each S_i is a provisional partition in the way that whenever its request queue becomes empty, it will be deactivated and thus be removed from resource level scheduling, it will become activated whenever there is a pending request in the request queue. Dynamic RLS shown in Section 3.4 makes this approach possible.

- Let α_i, Δ_i $(1 \leq i \leq m)$ be the rate and partition delay of S_i. The maximum computation requirement in the request queue is $\sum_{k=1}^{n} B_k$ where n is the total number of tasks that would access R_i and B_k is the longest duration of accessing R_i for each of those tasks.

  The longest response time W_i for any task for accessing R_i till its request is finished is:

$$W_i = (\sum_{k=1}^{n} B_k)/\alpha_i + \Delta_i$$

- The total interference time I_h for an instance of task T_h would be

$$I_h = \sum_{j=1}^{m} (W_j * M_j)$$

  where M_j is the number of times the job would access shared resource R_j. If the job does not lock a certain resource R_j then M_j equals zero.

- Assuming there are N tasks in the task group that is scheduled on partition Π (α, Δ) the schedulability test for tasks on Pi takes the following form:

$$\sum_{h=1}^{N} (C_h/(P_h - \Delta - I_h)) \leq \alpha$$

The server also needs to implement a mechanism to track the processing time of each critical section. If a task locks the resource longer than it declares, the server will intervene and release the lock. Since a job could not have two requests in the same queue (it would be blocked by the first request before it issues another request), the resource server does not need to track how many times the job has requested to access the resource. If the job does lock a resource more often than it declared, it would only affect its own partition but no other partitions. Hence, the application isolation property among partitions is still preserved.

6.1.3 Admission Control

In the discussion above, if a new task that would lock resource R requests to join the system, not only the longest response time for accessing R needs to be recomputed, but also the schedulability test needs to be performed again for all tasks that access R. This would incur a much higher computational complexity that is $O(N)$ where N is the number of tasks that lock R, rather than that of the conventional admission test that is $O(1)$. To address this issue, we propose a new admission test method.

Let B_{new} denote the longest duration that the new task T_{new} would access R. Instead of recomputing W_i, we could re-adjust the server partition S_i so that W_i would keep constant even after admitting T_{new}. Let C_{new} and P_{new} denote the computation time and period of T_{new} respectively, α_i', Δ_i' be the new rate and the new partition delay of S_i. The formula to keep W_i constant is:

$$W_i = (\sum_{k=1}^{n} B_k)/\alpha_i + \Delta_i = ((\sum_{k=1}^{n} B_k) + B_{new})/\alpha_i' + \Delta_i'$$

Admitting a new task would then depend on two conditions:

- Successful admission of the new task to the task group that is assigned to partition Π (α, Δ):

$$(C_{new}/(P_{new} - \Delta - I_{new})) \leq (\alpha - \sum_{j=1}^{f}(C_j/(P_j - I_j - \Delta)))$$

where the right side of the inequality has been computed during the start of the system and previous admission tests and there are f tasks that exist in the task group when this admission test is performed.

- Successful adjustment of server partition S_i from (α_i, Δ_i) to (α'_i, Δ'_i) with complexity similar to adjust an independent task parameter in EDF scheduling which is $O(1)$.

After the admission, parameters of S_i could be fine-tuned by a background process checking the longest response time of each task that locks R.

6.1.4 Partition Coalition for Resource Locking

The result above could be further improved if each task that would lock a resource R is the only task in its own partition. This could be done by creating a new partition level below the existing partition and assigning one partition to the task that accesses R and assigning other partitions on the same level to other tasks. In this way, partitions with tasks that were blocked when trying to lock R could join server partition S to process requests to R. Therefore, we have an additional rule for resource sharing:

> *The request for locking R will be processed not only in server partition S but also in partitions whose tasks are blocked from accessing R.*

To apply this rule to resource locking we have:

- When task T_i tries to lock resource R, its partition Π_i will join the partition coalition of the server partition S for R and other partitions with task requests in the queue of R to process the pending requests together till T_i locks R and thereafter releases R.

- After T_i releases R, it will withdraw from the partition coalition for R.

However, to compute the exact bound of the longest response time under this scheme is not trivial. The longest response time may not occur when all tasks that access R issue requests at the same time. This is because that a request with a tiny duration of the critical section, a large partition rate and a small partition delay would not increase the response time but reduce it. Moreover, the order of the requests would also affect the response time although the total computation demand to processing the requests remains the same. It is because on the supply side, the partition coalitions that process each request are different. In fact, we argue that to find a request sequence that would incur the longest response time is NP-hard.

Nevertheless, an upper bound that is not that tight would even considerably reduce the longest response time for T_i. Consider only Π_i and S in the partition coalition, the longest response time W_{jh} for T_h for accessing R_j could be computed as:

$$\sum_{k=1}^{n} B_k = (W_{jh} - \Delta_h)\alpha_h + (W_h - \Delta_S)\alpha_S \tag{6.1}$$

Notice that unlike in Section 6.1.2, the computed longest response time is different for tasks that access R. This is because each task's own partition (α_h, Δ_h) is considered during the computation. Since Δ_m is already taken into consideration in admission test on Π_h:

$$(C_h/(P_h - I_h - \Delta_h)) \leq \alpha_h$$

Therefore, from Equation (6.1), we have

$$\sum_{k=1}^{n} B_i = W_{jh} \times \alpha_h + (W_h - \Delta_R)\alpha_R \tag{6.2}$$

In general cases, B_i is much smaller (locking a variable takes only few instruction cycles on the magnitude of micro-seconds) than Δ that is typically lower bounded by the system scheduling quantum that is usually on the magnitude of milli-seconds. Therefore, using this optimization, the longest response time to lock a resource would be substantially reduced.

Further improvement could be obtained when W_h is smaller than any partition delay since each item could not be negative in Equation (6.2). Also notice that by using this approach, the resource server partition could actually be a dummy partition with rate of zero as long as all tasks are schedulable. Further research on resource locking of RTVRs could be pursued utilizing the property of scheduling quantum so that all critical sections are contained inside scheduling quanta [32, 71].

6.2 Discussion

6.2.1 Relation of Rate and Partition Delay

Throughout the dissertation, the problem of how to choose rate and partition delay for a certain partition has arisen in many places. The relation between the rate and the delay is shown on Figure 6.2. The valid value range of partition delay is $(0, P_{min})$ where P_{min} is the shortest period among tasks. In the general case, when the delay is 0, the rate equals $\sum_{i=1}^{n} (C_i/P_i)$; when the delay approaches P_{min}, the rate approaches infinity. Given the rate, the binary search method could be used to efficiently compute the partition delay in the range of $(0, P_{min})$

6.2.2 Scheduling Quantum Consideration

So far we have assumed that scheduling (especially task switching) can be performed with time being real numbers. However, many systems have a lower limit on the smallest time allocation unit, namely, the scheduling quantum. The scheduling quantum may also be viewed as the unit for specifying the precision of time measurements. It is imperative to consider the effects of scheduling quantum when we implement the scheduling scheme on real systems. In this subsection we re-

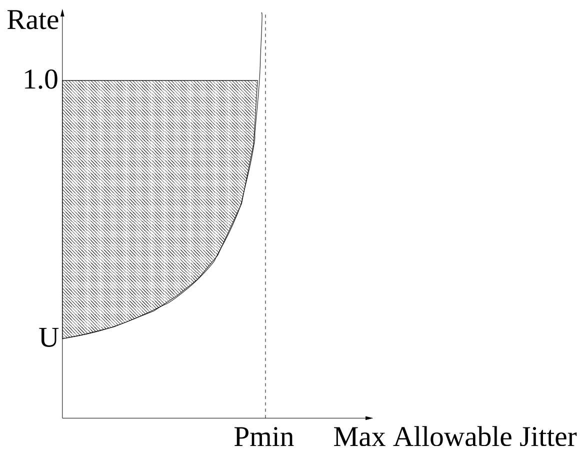


Figure 6.2: Rate-Delay Relation

examine the issues that we discuss in the previous subsection and we also show the interdependence among rate, partition delay and the scheduling quantum.

Theorem 6.1 *To schedule a partition* (α, Δ) *on a system with scheduling quantum of* Q *is equivalent to schedule a task with* $(H \times Q,\ \lfloor H/\alpha \rfloor \times Q$ *as computation time and period, respectively where* $H = \lceil \Delta\alpha/(2Q(1-\alpha)) \rceil$.

Proof: The scheduling quantum requires both computation time and period to be multiples of the quantum. Therefore, we need to normalize the result of Corollary 3.1 with regard to value Q. In order to meet the partition rate requirement, we put a ceiling function to computation time and put a floor function to task period.
■

Because of the interdependence of rate, partition delay and scheduling quantum that is discussed in [55], the introduction of scheduling quantum also puts some constraints on rate and partition delay.

- $\alpha \geq 1/(1+\lfloor \Delta/Q \rfloor)$: The partition will get at least one quantum available time after the actual partition delay $\lfloor \Delta/Q \rfloor$ happens. This puts a lower bound on the rate which means that the partition rate could not be infinitely small.

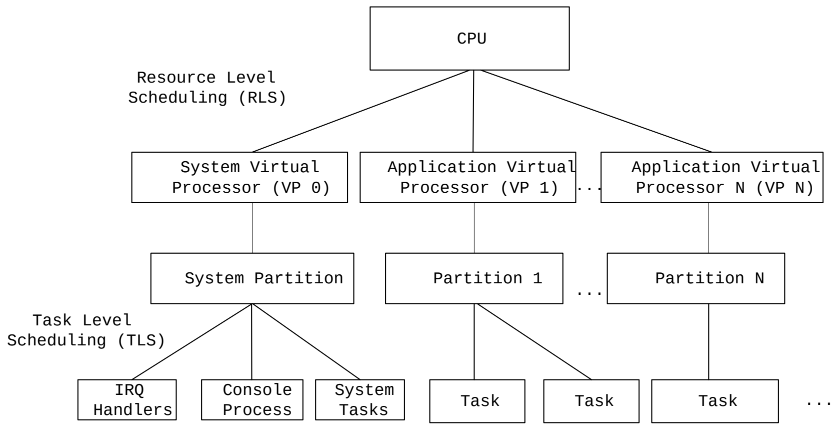
- $\Delta \geq Q$: The partition delay should be no less than the scheduling quantum.

Chapter 7

Implementation on Linux Kernel

7.1 Introduction

Based on the theoretical results obtained from previous chapters, we have implemented two real-time virtual resource (RTVR) prototypes based on Linux 2.4.18.3 kernel. We used the Linux operating system because of its open source code and because of its popularity in the personal computing and embedded system area. Our implementation architecture is shown in Figure 7.1. A CPU is partitioned into $N+1$ virtual processors (VP). VP 0, on which all system tasks run, is called system virtual processor (SVP). All other N virtual processors are called application virtual processors (AVP). The first prototype implements static resource level scheduling (Static RLS) which can be applied to systems with a predefined application task set. The second one implements dynamic RLS (Dynamic RLS) where partitions can be dynamically join and leave the system. As a general purpose OS, Linux kernel is not preemptive in kernel mode, which makes accurate CPU partition and deadline guarantee for real-time applications impossible. In both of our prototypes, (1) the kernel is made to be preemptive by spawning a kernel thread for each interrupt service routine (ISR) except for ISR 0 which is the routine for timer interrupt; (2) under the original Linux scheduler, a RLS is inserted to temporally partition the CPU, and the original Linux scheduler becomes task level scheduler (TLS); (3) system calls and utilities are provided to assist applications to specify real-time parameters and pass them to kernel.


CPU
Resource Level Scheduling (RLS)
System Virtual Processor (VP 0)
Application Virtual Processor (VP 1)
...
Application Virtual Processor N (VP N)
System Partition
Partition 1
...
Partition N
Task Level Scheduling (TLS)
IRQ Handlers
Console Process
System Tasks
Task
Task
Task
...
Figure 7.1: Real Time Virtual Resource Structure Overview

Linux is an open source, Unix style general purpose operating system which has become popular not only in server markets, but also in the personal computing and embedded system area. Our RTVR prototypes are built based on the Linux 2.4.18.3 kernel. However, Linux does not support RTVR directly. Therefore, in this chapter, we first explain the original Linux kernel structure, then we show what modifications were made to Linux to support RTVR. The detailed description of RTVR kernel scheduling implementation will be given and some other issues will finally be discussed.

7.2 Original Linux Kernel Structure

Although Linux is still evolving to be stronger and more reliable, the concept behind kernel process scheduling and state-transition remains unchanged. Figure 7.2 shows the scheduler framework of Linux kernel 2.4, which is similar to that of kernel 2.2 and of the later kernel 2.6.

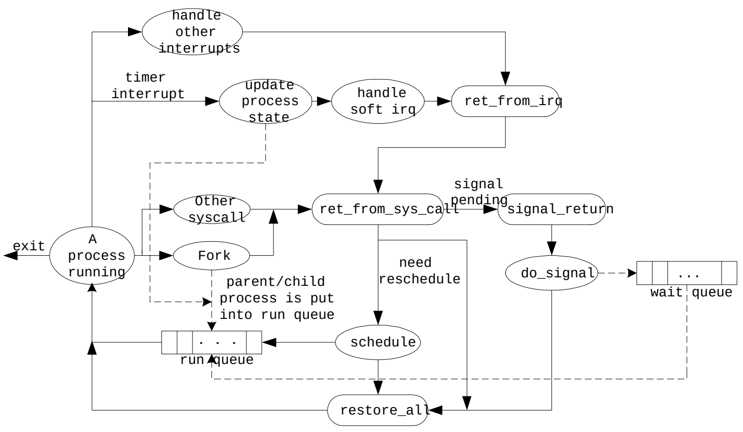


Figure 7.2: Linux Kernel State Transition Diagram

After the machine is booted and the kernel is loaded into memory, function *start_kernel()* does nearly all system initialization work. After some of the components of the kernel are initialized, *start_kernel()* forks a process called *init*, sets *need_resched* flag and then goes into an idle loop as process 0. As shown in Figure 7.2, *ret_from_sys_call* will notice that flag *need_resched* is raised and let the scheduler pick a process to run, which will be *init* process at that time.

Process *init* is unique in a number of ways [113]. First of all, it is the first user process run by the kernel, and it is responsible for firing off all the other processes that are needed to put the system as a whole into a usable state. Normally it will set up *getty* processes to let users login, establish network services such as FTP and HTTP daemons. Moreover, *init* is the ancestor of every subsequent process. Since it is the responsibility of the parent process to clean up after a process that has exited. If the parent has exited already, *init*, which never exits, becomes responsible for reaping it.

Inside the timer interrupt handler, the priority of the current process will be dynamically adjusted and the process will be re-inserted into the proper place in the run queue accordingly. If there is any soft IRQ which needs to be serviced, function *do_softirq()* will be called to wake up thread *ksoftirqd* to handle it. Thread *ksoftirqd* has a higher priority than most user processes. Usually it will be scheduled to run immediately after the timer interrupt is serviced.

Whenever a user process performs a system call or an interrupt occurs, the system goes into the kernel mode. As shown in Figure 7.2, eventually *ret_from_sys_call* will be reached. The kernel will (1) pick a new process to run if *need_resched* flag is set; (2) handle all pending signals, put associated processes into wait/run queue; or (3) simply continue running the old process. At that point, the process is restored back to user mode.

7.3 RTVP Kernel Structure

The kernel structure of RTVP is shown in Figure 7.4. A RLS is inserted under the original Linux scheduler which becomes the TLS now. When a timer interrupt occurs, the RLS updates system time information and runs the RLS to activate a particular VP according to the partition table. Then TLS will schedule the tasks on that VP.

Each VP can have its own customized TLS or share a common TLS with other VPs. As mentioned above, all VPs share the original Linux scheduler as TLS in our prototypes. The original Linux scheduler provides 3 scheduling algorithms: POSIX real-time processes can be scheduled either by SCHED_FIFO or by SCHED_RR (round-robin) policy; other non-real-time processes are scheduled by SCHED_OTHER policy.

Each VP owns a dedicated process run queue. When a particular VP is active, TLS will schedule processes in the corresponding run queue. In this way the RLS and TLS are cleanly separated and any customized TLS can be easily plugged in without extra structural modification.

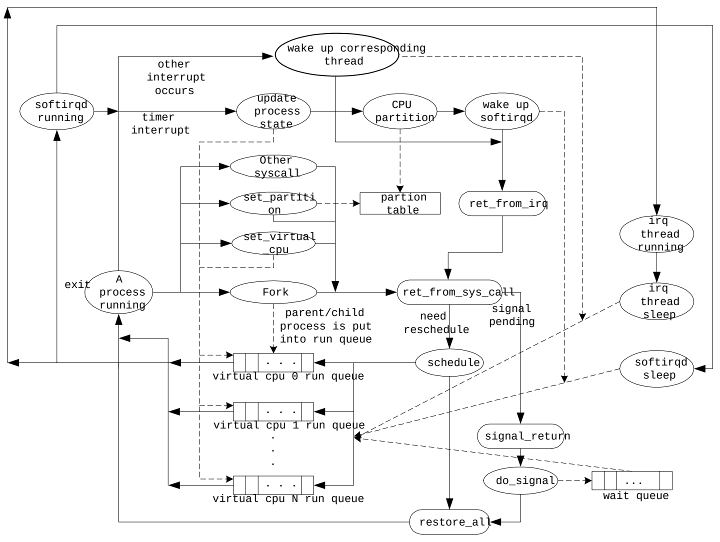


Figure 7.3: New Kernel Process Scheduling and State-Transition Diagram

Figure 7.4: New Linux Kernel State Transition Diagram

The original Linux kernel is not preemptive in the kernel mode in the sense that when an un-masked interrupt occurs, the kernel jumps into the corresponding ISR and services it immediately. This limitation makes accurate CPU partition impossible, which is required by RTVP. In our prototypes, an ISR thread is spawned for each non-timer-interrupt service routine by the *init* process during system initialization. In this way ISRs can be scheduled and run in appropriate time, which achieves accurate CPU partition.

ISR threads are assigned to the highest priority by default in our prototypes. Whenever an interrupt occurs, the corresponding ISR thread will be woke up and inserted to the head of a run queue which is the one associated with SVP by default. When the corresponding VP is activated by the RLS, TLS will guarantee to pick the ISR thread first and execute it. After the interrupt is serviced, the ISR thread will be de-queued from the run queue and put into sleep state. Since all ISR threads are assigned to run on SVP by default, applications running on other AVPs will not be disturbed by these interrupts, which is highly desirable for hard real-time applications. Users are allowed to change the priority of any ISR thread and set it to run on any AVP if necessary so that flexibility is also achieved.

7.4 Scheduling Implementation

We describe in this section the essential data structure of the dynamic RLS kernel. The static RLS kernel is simpler and can be viewed as a subset of the dynamic RLS kernel.

- structure *partition parameters [MAX PARTITION NUMBER]*; it contains:
  - float *rate*;
  - float *partition delay*;
- int *scheduling table[2][MAXIMAL LENGTH]*;
- int *effective table length*;

The first member data *partition parameters* is a collection of parameters of partitions. When a new partition is requested, an admission test will be performed based on the information of existing partitions. Once it is admitted, the parameters of an equivalent task are computed. Scheduling tables will be computed based on those task parameters as if a task were being scheduled. The actual scheduling and the admission test are dependent on the scheduling algorithm.

The second member data *scheduling table* shows the scheduling tables used in dynamic RLS. Scheduling according to a table is typically used in time-driven scheduling systems. It is conceptually simple and proven to be run-time efficient. Meanwhile, the event-driven scheduling algorithm which is used for dynamic RLS is notorious for its large scheduling overhead. Take the EDF scheduling algorithm for example, every time a new job is released or a job finishes the queue of deadlines is re-evaluated and the job with the earliest deadline will be chosen to run next. In order to utilize the advantages of both time-driven and event-driven scheduling, we choose to use tables in dynamic RLS.

Two sets of tables are employed in the implementation. The table computation task is assigned to the system partition exclusively, which is created to deal with system tasks. When the OS is scheduling according to one set of tables, the system partition will work on computing the other set using the scheduling scheme that we discussed in the previous section. EDF scheduling algorithm is used to schedule the tasks that correspond to partitions. When the table for scheduling is exhausted, the two tables will then alternate. This process is shown in Figure 7.5. In this way, scheduling decision is made in a batch fashion when the system partition is running, and all other real-time partitions will run in the efficient time-driven mode.

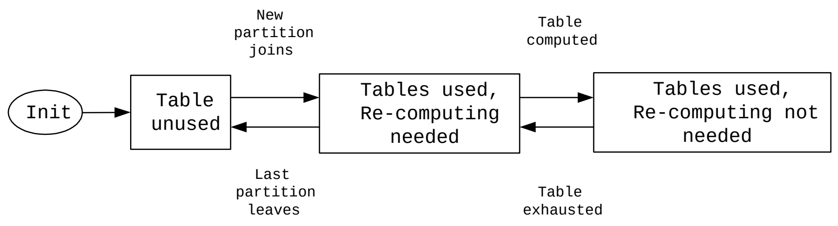


Figure 7.5: Scheduler State-Transition Diagram

There are several noteworthy issues related to the tables.

First, system partition has to finish computing one set of table before the other set is exhausted. By checking the length of tables and the partition delay parameter of the system partition, this would be easily done.

Second, adding or removing partitions will not be effective until a new set of tables is computed accordingly and is activated for scheduling. This would induce a bounded delay for partition related operations. Assuming that partitions in their nature are more static and are requested also in a more predictable fashion than tasks, this type of delay is acceptable. Even if the length of delay is too large for certain systems, the problem could be solved by changing the effective table length or by immediately recomputing a new set of tables and activating them.

Third, in the case that the system partition is the only active partition, the tables will not be used and the system partition will have exclusive access to the processor. The system performance would be the same as in the original Linux. The transition diagram is shown in Figure 7.5.

The third member data *effective table length* mandates the effective length of each table. There are basically three cases where the effective length might need a change. One is that the delay of partition related operation as mentioned above is too large. Second, if the scheduling period is less than the total length of these two tables, the effective length could be set accordingly as a way of optimization, thus eliminating the need to recompute the tables till any change happens to the partitions. Third, adjusting the effective length dynamically provides memory-size-sensitive systems another leverage to balance between memory usage and system performance.

7.5 Network Virtualization

So far, our discussion is limited to virtualizing processor. In an operating system, there are many other types of resources such as network, IO and storage. Among those resources, network is of particular interest to us because 1), network is preemptive on packets' level, 2), there is strong functionality support for network in Linux.

Actually, our implementation takes advantage of Linux network traffic control.

In Linux, traffic control provides queueing systems and scheduling mechanisms to manipulate the time and order of packets as they are transmitted onto the network. By default, Liunx has a single FIFO queue to collect packets and dequeue them as fast as the network hardware can accept. Other example traffic control options include Stochastic Fair Queuing (SFQ),Class Based Queuing (CBQ) and Generic Random Early Drop (GRED).

We implement a new queueing discipline called Real-Time Virtual Networking (RTVN). Similarly to RTVR, RTVN guarantees the amount of packets transmitted within a certain interval.

The implementation of RTVN is similar to that of Token Bucket Filter(TBF). The major difference between these two is that the purpose of TBF is to limit the highest speed of transmission while the purpose of RTVN is to limit the lowest speed of transmission. Partition delay in RTVN is mapped into the queue depth in TBF and admission control exists only in RTVN.

7.6 Memory Consumption Concern

Linux allows a maximum of 140 priorities in the system, which means each run queue associated with a VP has 140 sub-run queue link lists. Partition table(s) and process run queue link lists for each VP are statically compiled into the kernel.

Since the goal of our prototypes is to demonstrate on personal computing and embedded devices a computation model that is efficient and achieves a separation of concerns to real-time requirements among all applications sharing the same CPU, it is a reasonable expectation that there won't be many applications running simultaneously on such a small device. To save kernel memory, we set 6 AVPs in system. The maximal partition table length is set to 1024 which is long enough in most practical scenarios.

7.7 System Calls and Utilities

New system calls are built into the kernel of the prototypes. Utilities which performs appropriate system calls are also implemented to assist users to manage the partition table and to pass application real-time requirements to the kernel.

Add_Partition/Remove_Partition

For each VP, utility *add_partition* checks whether the real-time requirements specified by application tasks can be met. If not, an error message providing useful information will be printed out to the user. Otherwise, a new CPU partition table based on these requirements will be generated and passed to the kernel via *set_cpu_partition_table* system call. Note that in dynamic RLS, the utility will check whether the partition has the small-rate-small-delay attributes. If so, it will be added as a small partition.

Remove_partition will move a specified partition from the current partition table.

Get_Partition

This utility simply calls system call *get_cpu_partition_table* and prints out the current partition_table information.

Restore_Partition

This utility restores all application tasks on AVPs back onto SVP and resets the partition table so that CPU is exclusively utilized by SVP.

Set_Virtual_CPU

This utility calls system call *set_virtual_cpu_id* to assign a process running on a specified AVP.

Get_Virtual_CPU

This utility calls system call *get_virtual_cpu_id* and prints out the ID of the VP on which a given process is running.

7.8 SVP & ISR

SVP has to be given a fair amount of CPU utilization because all newly forked processes and system tasks such as ISR threads are running on it by default. A low CPU sharing utilization means a large IRQ response time which may cause the whole system to be unstable. For example, if SVP shares 5% of the CPU, the minimal interval between two neighboring partitions allocated to SVP will be 190 ms, assuming scheduling quantum is 10ms.. In the *add_partition* utility program, we enforce that SVP will share at least 10% of the CPU supply, which leaves enough CPU supply for applications on AVPs but is the minimum to ensure the system reliability.

A two-level resource partition is implemented in dynamic RLS. SVP can be configured as a small partition and grouped with other small partitions to form a normal partition so that CPU can be shared more efficiently in this way. For example, if a normal partition shares 10% of the CPU and SVP occupies 1 mini slot, it actually only shares 1% of CPU but the IRQ response time is almost the same as if it had shared 10% of CPU in static RLS. The implementation of two-level resource partitions is achieved by dynamically adjusting the scheduling quantum size which is the reverse of the timer interrupt frequency.

7.9 Experiments

Several experiments were carried out to study the performance of our prototypes. (1) The interrupt response time both on static and dynamic RLS were measured. (2)

We measured the CPU partition scheduling overhead. (3) An H.263 video decoder and a g-nibble game were selected as real-time applications that coexist to compete for CPU cycles. We qualitatively evaluate their performance on RTVR and compare it with that on Linux. (4) A video encoder application which requests mainly CPU cycles and network bandwidth was selected. We see how RTVR achieved better performance by appropriate co-partitioning of both resources.

Experiment (1)-(3) were carried on Toshiba Satellite 1100 laptop with an Intel Pentium III Celeron 1.33 GHz processor and 256 MB RAM. Experiment (4) were made on a desktop PC with Pentium III 600MHz processor and 256 MB RAM. After we present the measurement results, we discuss system performance-tuning problems.

7.9.1 Keyboard Response Time

To study the typical interrupt response on our RTVR, we measured keyboard response time. The keyboard uses IRQ 1 in our IBM-PC compatible laptop. The system is set to have a SVP and an AVP. An instrumentation code is inserted to measure the IRQ response time. Figure 7.6-(a) shows the average response time to IRQ 1 over 10 seconds.

In the static RLS prototype, the availability factor of AVP is set to $i/8$, where i is an integer in the range $[0, 7]$. The supply regularity is set to the minimal value which can accurately represent the corresponding availability factor. For example, if availability 7/8 is assigned to AVP, to accurately represent 7/8, supply regularity has to be at least 3 because $7/8 = 1/2 + 1/4 + 1/8$. In the dynamic RLS prototype, the AVP is set as a small partition. The rate of AVP is set to $i/10$, where i is also an integer in the range $[0, 9]$.

Unsurprisingly, dynamic RLS significantly reduces the IRQ response time by adding a small partition layer in RLS, which demonstrates that it can efficiently accommodate tasks with small rate and small delay.

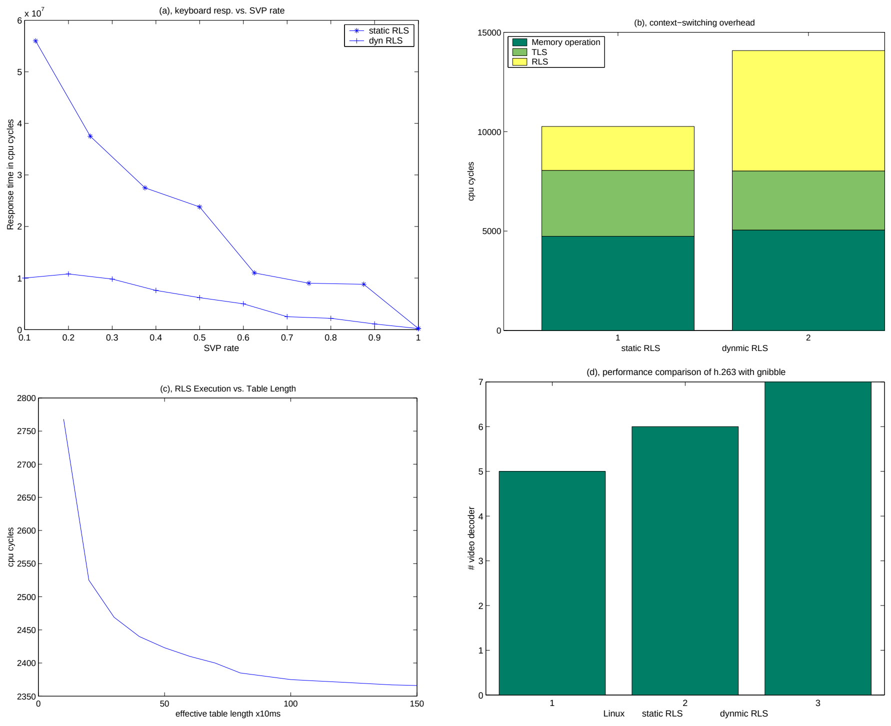


Figure 7.6: Experiment (1)-(3) Results

7.9.2 Scheduling Overhead

The typical CPU partition scheduling overhead consists of RLS execution, TLS execution and memory operation such as stack swap. Figure 7.6-(b) shows the maximal context-switching time measured over 10 seconds on both prototypes. Note that there are 1328940 CPU cycles/ms, which means that the context-switching overhead increased by adding a RLS layer is less than 0.4%.

The execution time of RLS in static RLS only consists of a table lookup operation, which is almost constant. In dynamic RLS, the maximal execution time also consists of a partition table generation, which is conducted once every effective table length. The average RLS execution time and context-switching time is directly related to it. Figure 7.6-(c) plots the average RLS execution time versus effective table length in dynamic RLS. Obviously, a longer table reduces scheduling overhead but consumes more memory. The trade-off between table length and memory consumption should be tuned on an individual platform basis.

7.9.3 Competing for a Single Resource

An H.263 video decoder and a g-nibble game were selected as real-time applications coexisting to compete for CPU cycles. We qualitatively measure their performance on RTVR and compare it with that on Linux. The H.263 decoder is a real-time application which decodes and displays 25 CIF (352 x 288 pixels) frames per second. The g-nibbles game is an interactive program which requests a small CPU rate but also a small delay. It can be used to subjectively test keyboard response. Figure 7.6-(d) shows the maximal number of video decoder threads running on each platform while the keyboard response is still satisfactory at the same time. Adding one more video decoder on any platform will either cause the video pictures to suffer from delays and/or jerkiness, or perceptibly degrade the g-nibble game.

The static RLS prototype is set up with 1 SVP and 3 AVPs. The availability factor and supply regularity of each AVP is set to 0.25 and 1, respectively. Two video decoder threads can be run on each AVP with good quality while still satisfying the g-nibble.

The dynamic RLS prototype is set up with a SVP and 2 AVPs, each of which is a small partition. The first one shares 40% CPU and runs 3 decoder threads. The second one shares 50% CPU and runs 4 decoder threads.

This experiment convincingly showed that our RTVR prototypes can accommodate more real-time tasks when they compete for the same resource (CPU).

7.9.4 Competing for Multiple Resources

In this experiment, three identical video encoders were executed simultaneously on our desktop PC for 10 seconds. Each one encodes the same video file and then sends the coded stream out. Thus, they mainly compete for CPU computation and network bandwidth. We show how RTVR achieves better performance by co-partitioning both resources.

To simplify the test, the video encoder is set to code 1 I frame and then 4 P frames at the normal rate of 25 frames/s. The average output bandwidth request by each stream is 50 Kbps. Each encoder can buffer 5 frames waiting to be sent at most. To see how network scheduling affects the performance, we set the total available bandwidth to 300 kbps. Each packet is 1k bytes in length. For each frame, we measure the time difference (in ms) between when it is actually sent and when it is supposed to be sent. The delay from previous frames will accumulate and result in the delay of later ones. We set the initial delay as 1000 ms; thus the deadline of frame k to be sent out can be calculated as $1000 + 40k$ ms.

Figure 7.7 showed the result with all four combinations of resource partition, that is, with/without CPU/network partition. Figure 7.7-1 can be seen as the result when they run on the original linux kernel. Each stream has about 50 % frames missing the deadline. Figure 7.7-2 is the performance measured with network partition only. We reserve 100 kbps bandwidth with latency 40 ms for each stream and the performance is much better than that in case 1, but still stream 2 and 3 have almost 20% deadline miss. Figure 7.7-3 is the performance measured with the CPU partition only. There is no deadline miss. The gap between deadline and the actual time when packets are sent increases as simulation continues, which means it is possible to support more stream(s). 7.7-4 is the performance measured with both CPU and network partition. As expected, there is no deadline miss. And, the"gap" is larger compared to case 3, which means the best performance is achieved by proper resource co-partitioning.

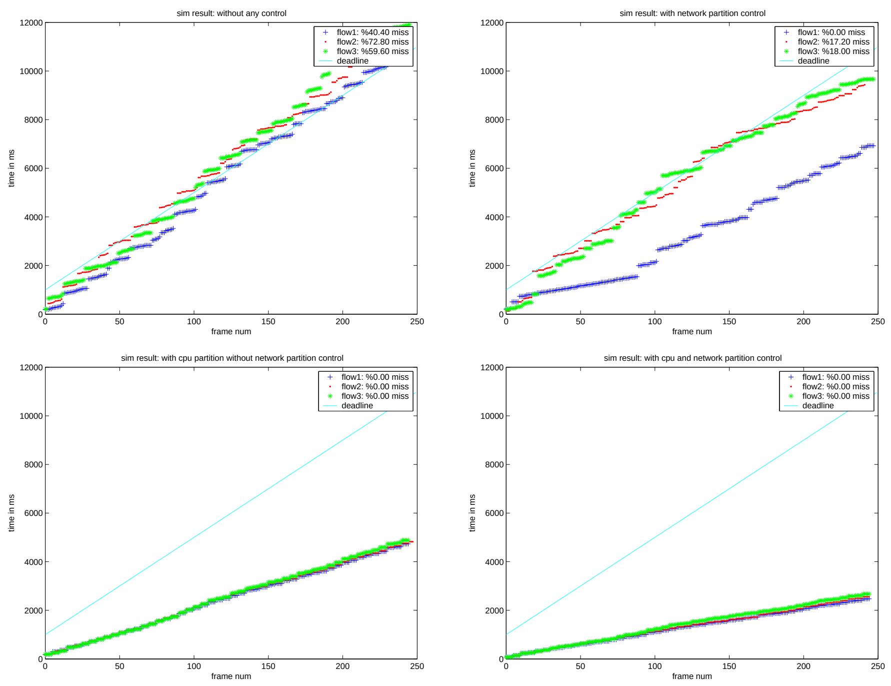


Figure 7.7: Experiment (4) Results

7.9.5 System Performance Tuning

We now discuss how to tune a RTVR system to achieve optimal performance.

The relationship between availability factor and partition delay is shown in Figure 7.8 in static RLS. For a given availability factor α, the actual partition delay D depends on the scheduling quantum Q and the supply regularity R_s. For example, given Q as 10 ms and α as 1/16, if R_s is 1, it is partitioned as 1 out of every 16 quantum which results in D equal to 150 ms. If R_s is 2, it is partitioned as 2 out of every 32 quanta and D can be as large as 300 ms. In Figure 7.8, the solid line and dash line represent the minimal and maximal partition delay for any given α in static RLS, respectively. In dynamic RLS, partition delay cannot be bounded by the rate only since the EDF algorithm is employed in RLS.

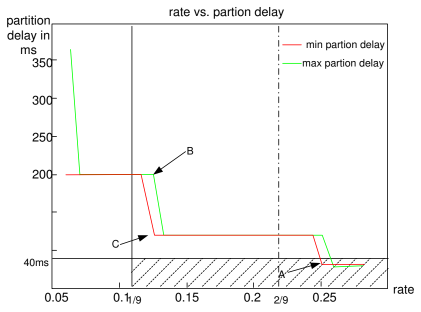


Figure 7.8: Partition Rate versus Partition Delay

Take the H.263 decoder used in the experiment as an example. We measured its periodic deadline as 40 ms and rate as 1/9. To guarantee the 40 ms deadline, α on static RLS should be at least 1/4 and R_s be 1 as the point A in Figure 7.8. Since each one only requires rate 1/9, two decoders can run on the same partition, which explains the experiment setup of static RLS. Clearly, in the static RLS case, the deadline of the H.263 decoder is the constraint.

In dynamic RLS, the scheduling quantum of a small partition can be as small as 1 ms, which makes the rate become the constraint. To maximize the number of the decoder threads running, we set two small partition AVPs. One shares 40% of CPU and accommodates 3 threads; the other shares 50% and accommodates 4. Note that theoretically we can group 8 decoder threads on an AVP sharing 90% CPU. However, as expected, too many threads grouped together will interfere with each other which makes some displays un-smooth.

Again, different supply regularity results in different partition delay even with the same availability factor in static RLS, such as point B and C in Figure 7.8. A larger supply regularity yields a larger partition delay which may violate the deadline of real-time tasks. On the other hand, it provides more scheduling flexibility, and possibly reduces scheduling overhead. For example, a RM scheduler can accommodate a task set $\{(C, P) \mid (4,7),(2,8),(2,32)\}$, where C is the computational time and P is the task period. However, it cannot accommodate a task set such as $\{(C, P) \mid (4,7),(2,8),(1,16)\}$.

Given a set of tasks with different real-time requirements, it is an interesting question how to tune a RTVR system to achieve optimal performance. We believe that all responsible factors such as partition delay, scheduling quantum, task grouping should be carefully investigated.

Chapter 8

Conclusion

8.1 Summary

In this dissertation, we present the architecture of Real-Time Virtual Resources (RTVRs). A RTVR operates at a fraction of the rate of the shared physical resource and its rate of operation varies with time but is bounded. One or more RTVRs are scheduled on the physical resource by a resource level scheduler. Tasks within the same task group are scheduled on a RTVR by a task level scheduler that is specialized to the real-time requirements of the tasks in the group.

This approach is well suited for an open system environment with a clean separation of concerns between task group scheduling and partition scheduling. Task group scheduling depends not on how partitions are scheduled but simply on the partition parameters. The bounded-delay partition model gives more flexibility on partition level scheduling as well. Scheduling of a real-time task group in a bounded-delay partition can be reduced to traditional scheduling on the normalized execution of a partition with deadlines earlier than their corresponding periods. Guaranteed jitter can be achieved with the bounded-delay partition model.

Specifically, we investigate the regularity-bounded resource partition model on integer domain. We characterize the rate variation of resource partitions from both temporal and supply dimensions using temporal regularity and supply regularity respectively. We analyze regularity-bounded partitions with respect to the schedulability of both the fixed priority scheduling and dynamic priority scheduling. We identify a certain set of partitions as regular partitions whereas the utilization bounds of both fixed and dynamic priority scheduling algorithms remain the same for dedicated resources. In fact, the task groups scheduled on regular partitions will not be able to distinguish whether they are executed on resource partitions or on dedicated resources. We also discuss the resource level scheduling of partitions by deriving a number of results that pertain to the construction of regularity-bounded partitions from regular partitions. These techniques are especially suitable for online scheduling and highly desirable for open systems.

We also extend RTVRs to a hierarchical structure in Chapter 4. Partitions on each level are scheduled as if they had access to a dedicated resource. Various cases of quantum size requirements are considered, thus providing a more practical approach to construct real-time virtual resources where context switching overheads can be substantial.

In Chapter 5, we further exploit the isolation property of the RTVR concept to solve the gang scheduling problem in parallel systems. End-to-end delay bound of RTVRs in distributed systems is also derived.

In Chapter 6, we investigate the resource locking issue in RTVR and presented a resource server solution with a highly efficient admission test and an optimization scheme called Partition Coalition that is based on the server solution and designed to substantially reduce the blocking time due to resource locking.

Finally in Chapter 7, we present our implementation of RTVRs on the Linux 2.4 kernel. Our implementation work gives some credence to a positive answer to whether RTVRs' theoretical advantages could actually be realized in practice. The design of resource level schedulers is intensively investigated. High level description of static resource level scheduling and dynamic resource level scheduling are presented. Modifications made to Linux in order to support RTVR are also explained in detail. The kernel scheduler implementation is emphasized. In addition, device virtualization with network as a prototype is also discussed. Several experiments are finally conducted to measure system performance in various aspects including IRQ response time, scheduling overhead and memory consumption.

8.2 Future Research

8.2.1 Model Refinement

There are still many open issues to be investigated. The possibility of interaction between tasks from different partitions may be pursued if the partition independence is not violated. Utilization bounds may be possible under some conditions. The scheduling of regular partitions could be improved by approaches similar to those used in the pinwheel problem. Our jitter bounds are not exact and tighter bounds may be possible with more sophisticated partitioning schemes.

8.2.2 Scheduling

We may consider further optimization in gang scheduling including (1) improving gang scheduling fragmentation so as to reduce the high concurrent context switching costs of parallel processes, (2) exploiting clock synchronization properties, (3) considering extensions to distributed systems that are more loosely-coupled parallel systems. Further research on resource locking in RTVR may yield optimization by utilizing the property of scheduling quantum when all critical sections are contained inside scheduling quanta [32, 71].

8.2.3 Implementation

We may also continue our work on improving the implementation. Even though we have shown that our implementation can deal with more than one resource type, our longer-term goal is to be able to apply the concept of RTVRs to all types of resources that an operating system may need to manage.

Appendix A

Notational Conventions and Acronyms

In this dissertation, we adhere to the following notational conventions.[1]

[**t**] denotes an instant or a period of time

[**J**] a real-time job (frame)

[**T**] a real-time task

[**n**] number of tasks in a system

[**C or c**] the execution time of a job

[**D**] the deadline of a job relative to its request time

[**P or p**] period of a partition or a task, or the minimum separation time of two consecutive jobs

[Π] a resource partition

[1]Note occasionally different notations are used in the research literature. For example, C is also used for task execution time; T may refer to the task period instead of a task.

[S] starting time

[S(t)] supply function with regard to time t

[E] end time

[$S^*(t)$] least supply function with regard to time t

[τ] task group

[α] availability factor (rate)

[Δ] partition delay

[V(t)] virtual time of actual time t

[P(t)] actual time of virtual time t

[Q] Scheduling Quantum

The following acronyms are used throughout the dissertation.

[CPU] Central Process Unit, often referred to as the "processor"

[PE] Processing Unit, as a conventional term used in parallel system.

[EDF] Earliest Deadline First Priority Assignment

[L&L] prefix, refer to the classical Liu & Layland paper [111]

[RMA] Rate-Monotonic Priority Assignment

[RTVR] Real Time Virtual Resource

[AAF] Adjusted Availability Factor

[EBD] Extended Bounded Delay

[**LCM**] Least Common Multiple

[**TLS**] Task Level Scheduling

[**RLS**] Resource Level Scheduling

[**FCFS**] First Come First Serve

[**VP**] Virtual Processor

[**SVP**] System Virtual Processor

[**AVP**] Application Virtual Processor

[**ISR**] Interrupt Service Routine

[**RTVN**] Real-Time Virtual Networking

Appendix B

Algorithm to Calculate Least Supply Function

Algorithm to get $S^*(t)$: We generate the critical pattern, from which $S^*(t)$ can be easily obtained.

```
Input: $W = (\{(S_0, E_0), (S_1, E_1), \ldots, (S_N, E_N)\}, P)$
Calculation_of_Critical_Partition {
  $\delta \leftarrow P - E_N$; // offset
  // Align to the first IBCP point.
  $W \leftarrow ((S_0 + \delta, E_0 + \delta), (S_1 + \delta, E_1 + \delta), \ldots, (S_N + \delta, E_N + \delta), P)$;
  $V \leftarrow W$;
  for $(i = 1; i \leq N; i++)$ {
    $V \leftarrow$ ShiftToNextIBCP$(V)$;
    Merge$(W, V)$;
  }
  return $W$;
}
```

	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=1
$V_a > V_b$	R	if($t_i - t + V_b) > V_a$ R[t,t+$V_a - V_b$],N[t+$V_a - V_b$,t_i] else R	R	R
$V_a \leq V_b$	N	N	if($t_i - t + V_a) > V_b$ N[t,t+$V_b - V_a$],R[t+$V_b - V_a$,t_i] else N	N

Table B.1: Computing Table

```
Merge($A, B$) {
// assume A is the generated partition so far, B is a partition based on an IBCP point.
  $T \leftarrow$ Merge all the time points of $A$ and $B$;
  $V_a \leftarrow 0$;
  $V_b \leftarrow 0$;
  $t \leftarrow 0$;
  Traverse $T${
    $t_i \leftarrow$ next element of $T$;
// Execute the update according to A is increasing or not and B is increasing or not.
Update values according to Table 1 where R means Replace $[t, t_i]$ in A with B and N means nothing need to be done.

    if(A=1) $V_a += t_i - t$;
    if(B=1) $V_b += t_i - t$;
    if($t_i$ belongs to A) A=A xor A;
    B=B xor B;
  }
  return T;
}
```

The time complexity of the algorithm is $O(N^2)$.

Bibliography

[1] Rtai:http://www.rtai.org/.

[2] Rtlinux:http://www.fsmlabs.com/.

[3] www.boeing.com. Technical report.

[4] www.evansdata.com. Technical report.

[5] Volvo technology report, no.1. Technical report, 1998.

[6] E. Amir, S. McCanne, and R. H. Katz. An active service framework and its application to real-time multimedia transcoding. In *SIGCOMM*, pages 178–189, 1998.

[7] N. Audsley and A. Wellings. Analysing apex applications. In *IEEE Real-Time Systems Symposium*, pages 39–44, December 1996.

[8] N. C. Audsley, A. Burns, M. Richardson, K. W. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):285–292, 1993.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[10] T. P. Baker. A stack-based allocation policy for real-time processes. In *IEEE Real-Time Systems Symposium*, 1990.

[11] S. Baruah. *The Uniprocessor Scheduling of Sporadic Real-Time Tasks.* PhD thesis, Department of Computer Science, The University of Texas at Austin, 1993.

[12] S. Baruah. Fairness in periodic real-time scheduling. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1995.

[13] S. Baruah. Overload tolerance for single-processor workloads. In *Real-Time Technology and Applications Symposium*, pages 2–11, 1998.

[14] S. Baruah and A. Bestavros. Pinwheel scheduling for fault-tolerant broadcast disks in real-time database systems. In *IEEE International Conference on Data Engineering*, pages 543–551, April 1997.

[15] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *The 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[16] S. Baruah and S. Lin. Improved scheduling of generalized pinwheel task systems.

[17] S. Baruah and S.-S. Lin. Pfair scheduling of generalized pinwheel task systems. *IEEE Transactions on Computers*, 47(7):812–816, July 1998.

[18] S. Baruah, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The complexity of periodic maintenance. In *International Computer Symposium*, 1990.

[19] S. K. Baruah. Fairness in periodic real-time scheduling. In *IEEE Real-Time Systems Symposium*, pages 200–209, December 1995.

[20] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multiframe tasks. *Real-Time Systems Journal*, 17(1):5–22, July 1999.

[21] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multiframe tasks. *Real-Time Systems Journal*, 17(1):5–22, July 1999.

[22] S. K. Baruah, D. Chen, and A. K. Mok. Jitter concerns in periodic task systems. In *IEEE Real-Time Systems Symposium*, 1997.

[23] S. K. Baruah, D. Chen, and A. K. Mok. Static-priority scheduling of multiframe tasks. In *Euromicro Conference on Real-Time Systems*, June 1999.

[24] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[25] S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the pre-emptive scheduling of periodic, real-time tasks on one processor. *The International Journal of Time-Critical Computing*, 2:301–324, 1990.

[26] S. K. Baruah, R. R. Howell, and L. E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, September 1993.

[27] S. K. Baruah, A. K. Mok, and L. E. Rosier. The preemptively scheduling of sporadic, real-time tasks on one processor. In *IEEE Real-Time Systems Symposium*, December 1990.

[28] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed Systems.* PhD thesis, the University of Illinios at Urbana-Champaign, 1994.

[29] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. Assigning real-time tasks to homogeneous multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–42, December 1995.

[30] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[31] M. Caccamo, G. Lipari, and G. Buttazzo. Sharing resources among periodic and aperiodic tasks with dynamic deadlines. In *IEEE Real-Time Systems Symposium*, December 1999.

[32] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *IEEE Real-Time Systems Symposium*, pages 161–170, 2001.

[33] M. Chan and F. Chin. General schedulers for the pinwheel problem based on double-integer reduction. *IEEE Transactions on Computers*, 41(6):755–768, June 1992.

[34] M. Y. Chan and F. Chin. Schedulers for larger classes of pinwheel instances. *Algorithmica*, 9:425–462, 1993.

[35] D. Chen. *Real-Time Data Management in the Distributed Environment.* PhD thesis, The University of Texas at Austin, 1999.

[36] D. Chen, A. K. Mok, and S. K. Baruah. On modeling real-time task systems. *Lecture Notes in Computer Science - Lectures on Embedded Systems*, 1494, October 1997.

[37] D. Chen, A. K. Mok, and T.-W. Kuo. Utilization bound re-visited. *IEEE Transactions on Computers*, 52(3):351–361, March 2003.

[38] S. Chen, J. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time Systems Journal*, 3:307–336, September 1991.

[39] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[40] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computer*, 39(9), 1990.

[41] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *SIGCOMM*, pages 14–26, 1992.

[42] J. Codd and M. Gouda. Flow theory. *IEEE/ACM Transactions on Networking*, 5(5):661–674, 1997.

[43] R. Davis and A. J. Wellings. Dual priority scheduling. In *IEEE Real-Time Systems Symposium*, pages 100–109, December 1995.

[44] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking Research and Experience*, pages 3–26, 1990.

[45] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[46] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[47] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transaction on Software Engineering*, 15:1497–1506, December 1989.

[48] R. Devillers and J. Goossens. General response time computation for the deadline driven scheduling of periodic tasks. *Fundamenta Informaticae*, 34, 1999.

[49] M. DiNatale and J. A. Stankovic. Applicability of simulated annealing methods to real-time scheduling and jitter control. In *IEEE Real-Time Systems Symposium*, pages 190–199, December 1995.

[50] B. P. Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems*. Addison-Wesley Publishing Company, 1998.

[51] D.Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. In *IEEE Conference on Communications Software: Communications for Distributed Applications and Systems*, pages 35–43, 1991.

[52] F. W. et al. A gang scheduling design for multiprogrammed parallel computing environments. *Job Scheduling Strategies for Parallel Processing, LNCS*, pages 1162:111–125, 1996.

[53] D. Feitelson. Job scheduling in multiprogrammed parallel systems. In *IBM Research Report RC Second Revision (1997)*, 1997.

[54] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. In *Journal of Parallel and Distributed Computing*, pages 16:306–318, 1992.

[55] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium*, pages 26–35, 2002.

[56] D. Ferrari. Client requirements for real-time communications services. *IEEE Communications*, 28:11, 1990.

[57] D. Ferrari and A. Gupta. Resource partitioning for real-time communication, 1993.

[58] J. Frederick. Microsoft's $40 billion bet. *Money Magazine*, May 2002.

[59] R. S. Garfinkel and W. J. Plotnicki. A solvable cyclic scheduling problem with serial precedence structure. *Operations Research*, 28(5):1236–1240, 1980.

[60] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1), January 1995.

[61] S. Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Real-Time Technology and Applications Symposium*, pages 60–71, June 1997.

[62] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated servicespacket switching networks. Technical report, Dept. of Computer Sciences, Univ. of Texas at Austin (ftp://ftp.cs.utexas.edu/pub/techreports/tr96-02.ps.Z), 1996.

[63] A. Gupta and D. Ferrari. Resource partitioning for multi-party real-time communication. Technical Report TR-94-061, Berkeley, CA, 1994.

[64] C.-C. Han. A better polynomial-time scheduleability test for real-time multiframe tasks. In *IEEE Real-Time Systems Symposium*, December 1998.

[65] C.-C. Han and K.-J. Lin. Scheduling distance-constraint real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 300–308, December 1992.

[66] C.-C. Han, K.-J. Lin, and C.-J. Hou. Distance-constrained scheduling and its applications to real-time systems. *IEEE Transactions on Computers*, 45(7):814–826, July 1996.

[67] C. C. Han and K. G. Shin. A polynomial-time optimal synchronous bandwidth allocation scheme for the timed-token mac protocol. In *INFOCOM'95*, volume 2, pages 875–882, 1995.

[68] C.-C. Han and H. ying Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *IEEE Real-Time Systems Symposium*, pages 36–45, December 1997.

[69] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transaction on Software Engineering*, 20(1):13–28, January 1994.

[70] S. Ho, T. Kuo, and A. Mok. Similarity-based load adjustment for static real-time transaction systems. In *IEEE Real-Time Systems Symposium*, December 1997.

[71] P. Holman and J. Anderson. Locking in pfair-scheduled mutliprocessor systems. In *IEEE Real-Time Systems Symposium*, pages 149 – 158, 2002.

[72] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *22th Hawaii International Conference on System Sciences*, January 1989.

[73] R. Holte, L. Rosier, I. Tulchinsky, and D. Varvel. Pinwheel scheduling with two distinct numbers. *Theoretical Computer Science*, 100:105–135, June 1992.

[74] C.-J. Hou and K. S. Tsoi. Dynamic real-time channel setup and tear-down in dqdb networks. In *IEEE Real-Time Systems Symposium*, pages 232–241, December 1995.

[75] C.-W. Hsueh and K.-J. Lin. An optimal pinwheel scheduler using the single-number reduction technique. In *IEEE Real-Time Systems Symposium*, pages 196 – 205, 1996.

[76] C.-W. Hsueh and K.-J. Lin. On-line schedulers for pinwheel tasks using the time-driven approach. In *10th Euromicro Workshop on Real-Time Systems*, pages 180–187, 1998.

[77] C.-W. Hsueh and K.-J. Lin. Scheduling real-time systems with end-to-end timing constraints using the distributed pinwheel model. *IEEE Transactions on Computers*, 50(1):51–66, January 2001.

[78] C.-W. Hsueh, K.-J. Lin, and N. Fan. Distributed pinwheel scheduling with end-to-end timing constraints. In *IEEE Real-Time Systems Symposium*, December 1995.

[79] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Real-time transaction processing: Design, implementation and performance evaluation. Technical Report Tech. Report 90-43, University of Massachusetts, May 1990.

[80] J. M. Hyman, A. A. Lazar, and G. Pacifici. Real-time scheduling with quality of

service constraints. *IEEE Journal of Selected Areas in Communications*, 9(7):1052–1063, 1991.

[81] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE Real-Time Systems Symposium*, pages 89–99, December 1992.

[82] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[83] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *IEEE Real-Time Systems Symposium*, pages 212–221, December 1993.

[84] M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Supercomputing'97*, 1997.

[85] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

[86] H. Kopetz. The time-triggered model of computation. In *IEEE Real-Time Systems Symposium*, December 1998.

[87] T. Kuo, K. Lin, and Y. Wang. An open real-time environment for parallel and distributed systems. In *20th International Conference on Distributed Computing Systems*, pages 206–213, 2000.

[88] T.-W. Kuo. *Real-Time Database - Semantics and Resource Scheduling*. PhD thesis, Department of Computer Science, The University of Texas at Austin, 1994.

[89] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open system architecture for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 256–267, 1999.

[90] T. W. Kuo and A. K. Mok. Load adjustment in adaptive real-time systems. In *IEEE Real-Time Systems Symposium*, pages 160–170, 1991.

[91] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *IEEE Real-Time Systems Symposium*, pages 151 –160, 2001.

[92] S. Lauzac, R. Melhem, and D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. In *International Parallel Processing Symposium*, pages 511–518, 1998.

[93] J. Lee, C. Lin, Y. Chang, and W. Shih. Real-time gang schedulings with workload models for parallel computers. In *International Conference on Parallel and Distributed Systems*, pages 114–121, 1998.

[94] K. Lee. Performance bounds in communication networks with variable-rate links. In *SIGCOMM*, pages 126–136, 1995.

[95] S. K. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology*, volume 2, pages 607–611, 1994.

[96] Y. Lee, D. Kim, M. Younis, and J. Zhou. Partition scheduling in apex runtime environment for embedded avionics software. In *The 5th International Conference on Real-Time Computing Systems and Applications*, pages 103–109, 1998.

[97] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, December 1990.

[98] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 14:44–53, 1986.

[99] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm - exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, December 1989.

[100] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[101] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[102] S. S. Lin and K. J. Lin. Pinwheel scheduling with three distinct numbers. In *EuroMicro Workshop on real-Time Systems*, June 1994.

[103] S.-S. Lin and K.-J. Lin. A pinwheel scheduler for three distinct numbers with a tight schedulability bound. *Algorithmica*, 019(04):411–426, 1997.

[104] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Real-Time Technology and Applications Symposium*, pages 166–175, December 2000.

[105] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Real-Time Technology and Applications Symposium*, pages 26–35, May 2001.

[106] G. Lipari and G. Buttazzo. Scheduling real-time multi-task applications in an open system. In *Euromicro Conference on Real-Time Systems*, pages 234–241, June 1999.

[107] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 217–226, 2000.

[108] T. D. C. Little and A. Ghafoor. Synchronization and storage models for multimedia objects. *IEEE Journal in Selected Areas of Communications*, 8(3):413–427, 1990.

[109] T. D. C. Little and A. Ghafoor. Multimedia synchronization protocols for broadband integrated services. *IEEE Journal in Selected Areas of Communications*, 9(9):1368–1382, 1991.

[110] C. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *Space Programs Summary 37-60*, II:28–37, 1969.

[111] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1), January 1973.

[112] J. W. S. Liu, K.-J. Lin, W.-K. Shih, and A. C. shi Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, pages 58–68, 1991.

[113] S. A. Maxwell. Linux core kernel 2nd edition. In *Coriolis Technology Press*, 2001.

[114] A. Mok. Task management techniques for enforcing ed scheduling on a periodic task. In *5th IEEE Workshop on Real-Time Software and Operating Systems*, May 1988.

[115] A. Mok and X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *IEEE Real-Time Systems Symposium*, pages 129–138, 2001.

[116] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Real-Time Technology and Applications Symposium*, pages 75–84, 2001.

[117] A. Mok and W. Poon. Non-preemptive robustness testing is not finite. In *Work in Process, Real Time System Symposium*, 2002.

[118] A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. Algorithms and complexity of the periodic maintenance problem. *Microprocessing and Microprogramming*, 27:657–664, 1989.

[119] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, MIT, 1983.

[120] A. K. Mok, P. Amerasinghe, M. Chen, S. Sutanthavibul, and K. Tantisirivat. Synthesis of a real-time message processing system with data-driven timing constraints. In *IEEE Real-Time Systems Symposium*, pages 133–143, December 1987.

[121] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transaction on Software Engineering*, 1997.

[122] A. K. Mok and X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. Technical report, Dept. of Computer Sciences, Univ. of Texas at Austin (ftp://ftp.cs.utexas.edu/pub/amok/UTCS-RTS-2001-02.ps), 2001.

[123] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. Technical report, Dept. of Computer Sciences, Univ. of Texas at Austin (ftp://ftp.cs.utexas.edu/pub/amok/UTCS-RTS-2001-01.ps), 2001.

[124] A. K. Mok and S. Sutanthavibul. Modeling and scheduling of dataflow real-time systems. In *IEEE Real-Time Systems Symposium*, December 1985.

[125] D.-W. Park. *A Generalized Utilization Bound Test for Fixed-Priority Real-Time Scheduling*. PhD thesis, Texas A&M University.

[126] D.-W. Park, S. Natarajan, A. Kanevsky, and M. J. Kim. A generalized utilization bound test for fixed-priority real-time scheduling. In *Second International Workshop on Real-Time Computing Systems and Applications*, pages 73–77, 1995.

[127] D.-T. Peng and K. G. Shin. A new performance measure for scheduling independent real-time tasks. *Journal of Parallel and Distributed Computing*, 19:11–26, 1993.

[128] R. Rajkumar, L. Abeni, D. D. Niz, S. Gosh, A. Miyoshi, and S. Saewong. Recent developments with Linux/RK. In *Proceedings of the Real Time Linux Workshop*, December 2000.

[129] P. V. Rangan, S. Ramanathan, and T. Kaeppner. Performance of inter-media synchronization in distributed and heterogeneous multimedia systems. *Computer Networks and ISDN Systems*, 27:549–565, 1995.

[130] J. Regehr and J. Stankovic. Hls: A framework for composing soft real-time schedulers. In *IEEE Real-Time Systems Symposium*, pages 3–14, 2001.

[131] P. Richardson and S. Sarkar. Adaptive scheduling: Overload scheduling for mission critical systems. In *Real-Time Technology and Applications Symposium*, pages 14–23, December 1999.

[132] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11:19–39, 1996.

[133] T. H. Romer and L. E. Rosier. An algorithm reminiscent of euclidean-gcd for computing a function related to pinwheel scheduling. *Algorithmica*, 17:1–10, 1997.

[134] J. Rushby. *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance.* NASA Contractor Report 209347. SRI International, Menlo Park, CA, 1999.

[135] O. Serlin. Scheduling of time critical processes. In *Spring Joint Computer Conference*, volume 41, pages 925–932, 1972.

[136] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[137] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39, 1990.

[138] S. Shigero, M. Takashi, and H. Kei. On the schedulability conditions on partial time slots. In *Real-Time Computing Systems and Applications Conference*, pages 166–173, 1999.

[139] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM J. Computing*, pages 537–552, July 1991.

[140] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *IEEE Real-Time Systems Symposium*, pages 2–13, 2003.

[141] D. Shuzhen, X. Qiwen, and Z. Naijun. A formal proof of the rate monotonic scheduler. In *Real-Time Computing Systems and Applications*, pages 500–503, 1998.

[142] F. Silva, E. Lopes, E. Aude, F. Mendes, J. Silveira, H.Serdeira, M. Martins, and W. Cirne. Response time analysis of gang scheduling for real time systems. In *Proceedings of the SPECTS 2002 - 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[143] M. Sjodin and H. Hansson. Improved response-time analysis calculations. In *IEEE Real-Time Systems Symposium*, pages 36–45, December 1998.

[144] S. H. Son. *Advances in Real-time Systems.* Prentice Hall, 1995.

[145] Space and N. W. S. Command. *Next Generation Computer Resources*, 1994.

[146] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 1:27–60, 1989.

[147] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling For Real-Time Systems - EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[148] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE Journal in Selected Areas of Communications*, 8(3):391–400, 1990.

[149] D. Stiliadis and A. Varma. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. *IEEE Transactions on networking*, 6(5):611–624, 1998.

[150] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

[151] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1), January 1995.

[152] Y. L. T. Kuo and K. Lin. Efficient on-line schedulability tests for priority driven real-time systems. In *Real-Time Technology and Applications Symposium*, pages 4–13.

[153] K. Team. Scheduling in k42. Technical report, IBM (http://www.research.ibm.com/K42/white-papers/Scheduling.pdf), 2001.

[154] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6:133–151, 1994.

[155] D. C. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. Technical Report TR-91-007, Berkeley, CA, 1991.

[156] R.-H. Wang. *Rule-Based Program Response-Time Analysis for Real-Time Applications*. PhD thesis, the University of Texas at Austin, 1997.

[157] Y.-C. Wang and K.-J. Lin. Implementing a general real-time scheduling framework in the RED-linux real-time kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255, 1999.

[158] S. A. Ward. An approach to real-time computation. In *7th Texas Conference on*

Computing Systems - Computing System for Real-Time Applications, pages 5.26–5.34, 1978.

[159] W. Wei and C. Liu. On a periodic maintenance problem. *Operations Research Letters*, 2(2):90–93, 1983.

[160] G. G. Xie and S. S. Lam. Delay guarantee of virtual clock server. *IEEE/ACM Transactions on Networking*, 3(6):683–689, 1995.

[161] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *IEEE Real-Time Systems Symposium*, pages 32–43, 1999.

[162] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints: exploiting data semantics. In *IEEE Real-Time Systems Symposium*, pages 240–251, 1996.

[163] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *SIGCOMM*, pages 113–121, 1991.

[164] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transaction on Computer Systems*, 9(2):101–124, 1991.

[165] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(8), 1987.

[166] L. Zhou and K. Shin. Rate-monotonic scheduling in the presence of timing unpredictability. In *Real-Time Technology and Applications Symposium*, pages 22–27, December 1998.

Vita

Xiang Feng, son of Mingsheng Feng and Liyun Wan, was born in Jingdezhen, the "Ceramics Metropolis" of China on April 30th, 1975. Under the unalloyed love and caring from his parents and his grandma, he enjoyed a happy childhood before attending school at the age of five. A naughty and rebellious boy by nature, he did not perform persistently well at school. His lowest grade was once 11/100 during the years at the Second High School of Jingdezhen. Despite of those "lows", he always ranked at the top of his class from primary school to college.(There are no similar ranking systems in graduate schools, fortunately.)

Exempted from the entrance exam at the age of sixteen, he was admitted to Tongji University. Then he got involved in various activities. He started as a part-time developer in a software firm. He later lead a group of his classmates to contract various software projects. He also served as a lecturer in the extension program of Tongji University for three years. Finally, he decided to pursue serious research because he enjoys being mentally challenged. In 1995, he was admitted to graduate school at Tongji University and again exempted from the entrance exam. In 1998, he came to the U.S. and started his Ph.D. study at the University of Texas at Austin where he met his wife, Yunda. Currently, Alex and Yunda live happily somewhere on this planet with their pretty daughter, Anita (Neenee).

Xiang (Alex) Feng has contributed to the following publications:

1 Al Mok, **Xiang (Alex) Feng**, "Real-Time Virtual Resource: A Timely Ab-

straction for Embedded Systems", Lecture Notes of Computer Science, 2002,pp. 182-196

2 **Xiang (Alex) Feng**, Al Mok, "A Model of Hierarchical Real-Time Virtual Resources", *IEEE Real Time System Symposium 2002*, Austin, Dec. 2002. pp. 26-35

3 Al Mok, **Xiang (Alex) Feng**, "Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds", *IEEE Real Time System Symposium 2001*, London, UK, Dec. 2003. pp. 129-138

4 Al Mok, **Xiang (Alex) Feng**, Deji Chen, "Resource partition for real-time systems", *IEEE Real-Time Technology and Applications Symposium 2001*, Taipei, Taiwan, May, 2001. pp. 75-84

5 **Xiang (Alex) Feng**, Al Mok, "Real-Time Gang Scheduling, Dynamic Partition Scheduling and Resource Locking for RTVR", submitted to conference publication.

6 **Xiang (Alex) Feng**, Zhengting He, Al Mok , "Implementation of Real-Time Virtual Resources on Linux", submitted to conference publication.

Permanent Address: 226 South Zhonghua Rd.
Jingdezhen, Jiangxi, 333000
China

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of