STRONGLY PARTITIONED SYSTEM ARCHITECTURE FOR INTEGRATION OF REAL-TIME APPLICATIONS

By

DAEYOUNG KIM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2001

To my parents, my wife Yoonmee, and my son Minjae

ACKNOWLEDGMENTS

I would like to thank many individuals for the care and support they have given to me during my doctoral studies. First of all, I would like to express my deep gratitude to Professor Yann-Hang Lee. As my advisor, he has provided me with invaluable guidance, and insightful comments and suggestions. I would also like to thank Professors Paul W. Chun, Douglas D. Dankel, Randy Y. Chow, and Jih-Kwon Peir for serving as my dissertation committee.

I would also like to thank the former and present members of the Real-Time Systems Research Group--Yoonmee Doh, Okehee Goh, Yan Huang, Vlatko Milosevski, James J. Xiao, Youngjoon Byun, Jaeyong Lim, and Kunsuk Kim--for their friendship and discussion. I am also grateful to my friend Sejun Song at CISCO systems, and Professor Youngho Kim at Pusan National University, for their encouragement and discussion. My thanks should also go to Dr. Mohamed Younis at the University of Maryland, in Baltimore County; Dr. Zeff Zhou and James McElroy, at Honeywell International; and Dr. Mohamed Aboutabl, at Lucent Technology. My deep gratitude also goes to my former directors of ETRI: Professor Munkee Choi, at Information and Communications University; and Hyupjong Kim, at Holim Technology.

Thanks also go to the CISE Department of the University of Florida, ETRI, Allied Signal Inc., Honeywell International, NASA, and the CSE Department of Arizona State University for providing financial and administrative support during my Ph.D. program.

I give special thanks to my parents, since they have always encouraged me in my studies, and believed in me. I am also grateful to my mother-in-law and my sisters, Mikyung and Mijung. My final acknowledgements go to my wife Yoonmee and my son Minjae for their love shown to me during the study.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xii

CHAPTERS

1 INTRODUCTION 1

1.1 Integrated Real-Time Systems 1
1.2 Research Background and Related Works 4
1.2.1 Fundamental Scheduling Theories 4
1.2.2 Model and Theories for Integration of Real-Time Systems 6
1.2.3 Real-Time Kernels 7
1.2.4 Real-Time Communication Networks 9
1.3 Main Contributions 10
1.4 Organization of the Dissertation 11

2 INTEGRATED SCHEDULING OF PARTITIONS AND CHANNELS 13

2.1 Introduction 13
2.2 Integrated Real-Time Systems Model 14
2.3 Fundamental Scheduling Theory for Integration 19
2.3.1 Schedulability Requirement 19
2.3.2 Characteristics of the Two-Level Scheduling Theory 22
2.3.3 Distance-Constrained Low-Level Cyclic Scheduling 25
2.4 Integrated Scheduling Theory for Partitions and Channels 30
2.4.1 Scheduling Approach 30
2.4.2 Algorithm Evaluation 42
2.5 Solving Practical Constraints 47
2.5.1 Incremental Changing Scheduling 48
2.5.2 Replicated Partition Scheduling 52
2.5.3 Fault Tolerant Cyclic Slot Allocation for Clock Synchronization 55
2.5.4 Miscellaneous Practical Constraints 57
2.6 Scheduling Tool 60
2.7 Conclusion 62

3 SOFT AND HARD APERIODIC TASK SCHEDULING 64

3.1 Introduction 64
3.2 Aperiodic Task Scheduling Model 65

3.3 Soft Aperiodic Task Scheduling 67
3.3.1 Left Sliding (LS) 67
3.3.2 Right Putting (RP) 69
3.3.3 Compacting 71
3.3.4 DC^2 Scheduler 72
3.4 Hard Aperiodic Task Scheduling 73
3.4.1 Low-Bound Slack Time 73
3.4.2 Acceptance Test Considering Future RP Operations 75
3.4.3 Dynamic Slack Time Management 76
3.4.4 Multi-Periods Aperiodic Server Scheduler 78
3.5 Simulation Studies of the DC^2 Scheduler 80
3.5.1 Response Time 80
3.5.2 Acceptance Rate 81
3.6 Conclusion 85

4 REAL-TIME KERNEL FOR INTEGRATED REAL-TIME SYSTEMS 86

4.1 Introduction 86
4.2 SPIRIT-μKernel Model and Design Concepts 88
4.2.1 Software Architecture model for Integrated Real-Time Systems 88
4.2.2 Design Concepts of the SPIRIT-μKernel 89
4.3 SPIRIT-μKernel Architecture 90
4.3.1 Memory Management 91
4.3.2 Partition Management 93
4.3.3 Timers/Clock Service 95
4.3.4 Exception Handling 95
4.3.5 Kernel Primitives Interface 96
4.3.6 Inter-Partition Communication 96
4.3.7 Device Driver Model 97
4.4 Generic RTOS Port Interface (RPI) 98
4.4.1 Event Delivery Object (EDO) 99
4.4.2 Event Server 102
4.4.3 Kernel Context Switch Request Primitive 103
4.4.4 Interrupt Enable/Disable Emulation 104
4.4.5 Miscellaneous Interface 104
4.5 Performance Evaluation 105
4.5.1 Kernel Tick Overhead 105
4.5.2 Partition Switch Overhead 107
4.5.3 Kernel-User Switch Overhead 108
4.5.4 TLB Miss Handling Overheads 108
4.6 Conclusion 109

5 REAL-TIME COMMUNICATION NETWORKS FOR INTEGRATED REAL-TIME SYSTEMS 110

5.1 Introduction 110
5.2 SPRETHER Architecture and Model 111
5.3 SPRETHER Protocols 114
5.3.1 Table-Driven Proportional Access Protocol 114
5.3.2 Single Message Stream Channel (SMC) 117
5.3.3 Multiple Message Streams Channel (MMC) 118
5.3.4 Non-Real-Time Channel (NRTC) 123
5.3.5 Packet and Frame Format and Management 124

5.3.6 SPRETHER API 125
5.4 Scheduling Real-Time Messages 126
5.4.1 Scheduling Requirement of the SMC and MMC Channels 126
5.4.2 Distance-Constrained Cyclic Scheduling 129
5.5 Prototype and Performance Evaluation 131
5.5.1 TDPA Protocol Overhead 133
5.5.2 Response Time of Non-Real-Time ICMP Message 134
5.5.3 Throughput of Non-Real-Time Connection 134
5.6 Conclusion 137

6 CONCLUSIONS AND FUTURE WORK 138

6.1 Contributions 138
6.2 Future Research Directions 139

LIST OF REFERENCES 141

BIOGRAPHICAL SKETCH 148

LIST OF TABLES

Table Page

2-1 Index Notations used in the SPIRIT Model 17
2-2 Notations used in the SPIRIT Model 18
2-3 Task Parameters for the Example Partitions 22
2-4 Task Parameters for the Example of Integrated Scheduling 37
2-5 Cyclic Time Schedule for Processor 1 39
2-6 Cyclic Time Schedule for Processor 2 39
2-7 Slot Allocation of Cyclic Scheduling for Bus of Major Frame Size 202 40
2-8 Notations for Incremental Changing Algorithm 48
3-1 Frame-to-Frame Slack Time Table 74
4-1 Partition Configuration Information 94
4-2 Measured Kernel Overheads 106
4-3 Partition Switch Overheads 107
4-4 TLB Miss Handling Overheads 109
5-1 Parameters Used in the SMC Protocol 117
5-2 Parameters Used in the MMC Protocol 120
5-3 The SPRETHER Packet Field Description 125
5-4 Synchronization (NIS Packet Processing) Overhead 133

LIST OF FIGURES

Figure Page

2-1 Model for Strongly Partitioned Integrated Real-Time Systems (SPIRIT)....15
2-2 Task Model and Deadline....16
2-3 Task and Partition Execution Sequence....17
2-4 Inactivity Periods of the Example Partitions....23
2-5 Maximum Partition Cycles for Different Processor Capacity Assignments....24
2-6 Maximum Partition Cycles for Partition 1 under Different Task Deadlines....26
2-7 Maximum Partition Cycles of Partition 1 under Different Processor Utilizations....26
2-8 Example Cyclic Schedule at the Lower Level....28
2-9 Distance-Constrained Scheduling Model....29
2-10 Processor Cyclic Schedule Example....30
2-11 Combined Partition and Channel Scheduling Approach....31
2-12 Deadline Decomposition Algorithm....32
2-13 Scheduling Example of Two Messages in Combined Channel....35
2-14 Heuristic Channel Combining Algorithm....36
2-15 Processor Capacity vs. Partition Cycle....38
2-16 Bus Capacity vs. Channel Server Cycle....41
2-17 Schedulability Test for Configuration (4,3,5) and (2,2,4)....43
2-18 Measures for Bus Utilization and Capacities....46
2-19 Ratio of Message Deadline to Task Deadline....47
2-20 Cyclic Schedule Example for Incremental Changing....49

2-21 Example of Attaching New Partition 50
2-22 Replicated Partitions in IMA Systems 53
2-23 Example of Fault Tolerant Cyclic Slot Allocation 56
2-24 Cycle Transformation Algorithm for Fixed Bus Major Frame Size Scheduling 58
2-25 Logical Channel Server Architecture 59
2-26 Usage of Scheduling Tool in Avionics System Design 60
2-27 The Structure of the Scheduling Tool 61
2-28 System and Scheduling Parameters in the Tool Interface 62
3-1 Example of Feasible Cyclic Schedule 67
3-2 Modified Schedule after LS 68
3-3 Modified Schedule after RP 70
3-4 DC^2 Scheduing Algorithm 73
3-5 Example of the sf, sl, and sr Parameters 74
3-6 Acceptance Test Algorithm 76
3-7 Six Possible Slack Time Adjustments after an RP Operation 78
3-8 Two-Level Hierarchy of Aperiodic Task Schedulers 79
3-9 Average Response Time 83
3-10 Average Acceptance Rate 84
4-1 Strongly Partitioned Integrated Real-Time System Model 88
4-2 Architecture of the SPIRIT-μKernel 91
4-3 EDO, Event Server, and Kernel-Context Switch Request Primitive Interaction 99
4-4 Structure of the Event Delivery Object 100
4-5 SPIRIT-μKernel's Generic Interrupt/Exception Handling Procedure 101
4-6 SPIRIT-μKernel's Generic Event Server Procedure 103
4-7 Kernel Tick Overheads 107
5-1 SPRETHER Real-Time Communication Model 112

5-2 Message Stream Model in the SPRETHER....113
5-3 Strongly Partitioned Real-Time Ethernet Architecture....114
5-4 Channels and Gaps Configuration According to a Cyclic Scheduling Table....115
5-5 Example of TDPA the Real-Time Ethernet Protocol....116
5-6 Analysis of SMC Channel Activity....117
5-7 Utilization of SMC Channel....118
5-8 Analysis of MMC Channel Activity....119
5-9 Utilization of MMC Channel....120
5-10 Utilization of MMC Channel when Packet Size is 64, 128, 256, and 512 Bytes....122
5-11 Protection Mechanism of the Real-Time Channel....124
5-12 SPRETHER Packet Format....124
5-13 Architecture of MMC Channel Server....127
5-14 Example of a Communication Frame Cyclic Schedule....131
5-15 SPRETHER Prototype Platform....132
5-16 Software Architecture of SPRETHER....132
5-17 Response Time of Non-Real-Time Packet (64Bytes ICMP Packet)....135
5-18 Throughput of Non-Real-Time Packet (FTP)....136

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

STRONGLY PARTITIONED SYSTEM ARCHITECTURE FOR INTEGRATION OF REAL-TIME APPLICATIONS

By

Daeyoung Kim

August 2001

Chairman: Dr. Yann-Hang Lee
Major Department: Computer and Information Science and Engineering

In recent years, the design and development of technologies for real-time systems have been investigated extensively in response to the demand for fast time-to-the-market and flexible application integration. This dissertation proposes a design approach for integrated real-time systems in which multiple real-time applications with different criticalities can be feasibly operated, while sharing computing and communication resources.

The benefits of integrated real-time systems are cost reduction in design, maintenance, and upgrades; and easy adoption of Commercial-Off-The-Shelf (COTS) hardware and software components. The integrated real-time applications must meet their own timing requirements and be protected from other malfunctioning applications. To guarantee timing constraints and dependability of each application, integrated real-time systems must be equipped with strong partitioning schemes. Therefore, we name them strongly partitioned integrated real-time systems (SPIRIT).

To prove the theoretical correctness of the model and provide a scheduling analysis method, we developed a fundamental two-level scheduling theory, an algorithm for integrated

scheduling of partitions and communication channels, and aperiodic task scheduling methods. We also augmented the scheduling model with practical constraints, which are demanded by ARINC 653 Integrated Modular Avionics standards. The scheduling algorithms are evaluated by mathematical analysis and simulation studies, and implemented as an integrated scheduling tool suite.

To provide a software platform for integrated real-time systems, we developed a real-time kernel, SPIRIT-μKernel, which implements strong partitioning schemes based on the two-level scheduling theory. The kernel also enables a generic mechanism to host heterogeneous COTS real-time operating systems on top of the kernel. The performance of critical parts of the kernel on the prototype kernel platform is measured in the prototype environment of a PowerPC embedded controller.

Finally, we propose a real-time Ethernet named Strong Partitioning Real-Time Ethernet (SPRETHER), which is designed for the communication network of integrated real-time systems. To overcome the lack of deterministic characteristics of Ethernet, we use a software-oriented synchronization approach based on a table-driven proportional access method. We performed scheduling analysis work for SPRETHER and measured performance by experiments on a prototype network.

CHAPTER 1
INTRODUCTION

1.1 Integrated Real-Time Systems

Advances in computer and communication technology have introduced new architectures for real-time systems, which emphasize dependability, cost reduction, and integration of real-time applications with different criticalities. Away from the traditional federated and distributed implementation for real-time systems, the new approach, referred to as Strongly Partitioned Integrated Real-time Systems (SPIRIT), uses multiple COTS processor modules and software components in building comprehensive real-time systems. It allows the real-time applications to be merged into an integrated system.

The benefits of integrated real-time systems are cost reduction in design, maintenance, and upgrades, and easy adoption of Commercial-Off-The-Shelf (COTS) hardware and software components. Applications running in an integrated real-time system must meet their own timing requirements and be protected from other malfunctioning applications, while sharing computing and communication resources. To guarantee timing constraints and dependability of each application, integrated real-time systems must be equipped with strong partitioning schemes.

Strong partitioning schemes include temporal and spatial partitioning to protect applications from potential interference. Temporal partitioning ensures that execution time and communication bandwidth reserved to an application would not be changed either by overrun or by hazardous events of other applications. Spatial partitioning guarantees that physical resources, such as memory and I/O space of an application, are protected from illegal accesses attempted by other applications. Providing strong partitioning schemes for the integration of real-time applications is a main goal of this dissertation.

The basic scheduling entities of integrated real-time systems are partitions and channels. A *partition* represents a real-time application that is protected from potential interference by means of temporal and spatial partitioning schemes. Multiple cooperating *tasks* are scheduled by a *partition server* within a partition. To facilitate communications among applications, each partition can be assigned one or more communication channels. An application can transmit messages using its channel, and can access the channel buffers exclusively. In this sense, a channel is a spatial and temporal partition of a communication resource and is dedicated to a message-sending application. It serves multiple real-time messages that belong to the same partition and are scheduled by a *channel server*. Groups of partitions and channels are scheduled by a *cyclic CPU scheduler* and *cyclic bus (network) scheduler*, respectively.

A good example of an integrated real-time system is the Integrated Modular Avionics (IMA), which is being embraced by the aerospace industry these days [1,2,3]. An application running within a partition can be composed of multiple cooperating tasks. For instance, Honeywell's Enhanced Ground Proximity Warning System (EGPWS) consists of tasks for map loading, terrain threat detection, alert prioritization, display processing, etc. With spatial and temporal partitioning, the EGPWS application can be developed separately and then integrated with other applications running in different partitions of an IMA-based system. Its execution cannot be affected by any malfunctions of other applications (presumably developed by other manufacturers) via wild writes or task overruns. As long as sufficient resources are allocated to the partition and the channels, the EGPWS application can ensure proper execution and meet its real-time constraints.

One apparent advantage of IMA-based systems with spatial and temporal partitioning is that each application is running in its own environment. Thus, as long as the partition environment is not changed, an application's behavior remains constant even if other applications are modified. This leads to a crucial advantage for avionics systems; i.e., when one application is revised, other applications don't need to be re-certified by the FAA. Thus, the integration of

applications in a complex system can be upgraded and maintained easily. It is conceivable that such architecture with spatial and temporal partitioning can be used for integrating general real-time applications.

To guarantee a strong partitioning concept in integrated real-time systems, we investigated a two-level hierarchical scheduling theory that adopts distance-constrained cyclic scheduling at lower levels and fixed-priority driven scheduling at higher levels. According to the two-level scheduling theory, each real-time application (partition) is scheduled by a cyclic scheduler; and tasks within an application are scheduled by a fixed-priority scheduler. Based on the two-level scheduling theory, we devised a heuristic deadline decomposition and channel-combining algorithm to schedule both partitions and channels concurrently. Soft and hard aperiodic tasks are also supported efficiently by reclaiming unused processor capacities with a distance-constrained cyclic scheduling approach. We also augmented the scheduling model with practical constraints, which are demanded by ARINC 653 Integrated Modular Avionics standards. The scheduling algorithms are evaluated by mathematical analysis and simulation studies, and are implemented as an integrated scheduling tool suite.

To establish a software platform for integrated real-time systems, we developed a real-time kernel, SPIRIT-μKernel. The goals of the SPIRIT-μKernel are to provide dependable integration of real-time applications, flexibility in migrating operating system personalities from kernel to user applications, including transparent support of heterogeneous COTS RTOS on top of the kernel, and high performance. To support integration of real-time applications that have different criticality, we have implemented a strong partitioning concept using a protected memory (resource) manager and a partition (application) scheduler. We also developed a generic RTOS Port Interface (RPI) for easy porting of heterogeneous COTS real-time operating systems on top of the kernel in user mode. A variety of operating system personalities, such as task scheduling policy, exception-handling policy and inter-task communication can be implemented within the partition according to individual requirements of partition RTOS. To demonstrate this concept,

we ported two different application level RTOS: WindRiver's VxWorks 5.3 and Cygnus's eCos 1.2, on top of the SPIRIT-μKernel. Performance results show that the kernel is practical and appealing because of its low overhead.

Finally, we propose a Strong Partitioning Real-Time Ethernet (SPRETHER) to enable real-time communication networks for integrated real-time systems using Ethernet technology. Currently, Ethernet is the dominant network technology because of its cost-effectiveness, high bandwidth, and availability. With a Carrier Sense Multiple Access / Collision Detect (CSMA/CD) MAC protocol, Ethernet is inherently incapable of guaranteeing deterministic accesses due to possible packet collision and random back-off periods. However, in order to take advantage of COTS Ethernet in real-time systems, it is preferred not to change the standard network hardware components and protocol, so that most advanced and inexpensive Ethernet products can be used without any modifications. The issue is then how to design an easily accessible, reliable, deterministic, and affordable real-time Ethernet for safety-critical real-time systems using COTS Ethernet hardware and software methods. In the dissertation, we propose a Table-Driven Proportional Access (TDPA) protocol in the standard Ethernet MAC layer to achieve guaranteed real-time communication. In addition to real-time traffic, non-real-time Ethernet traffic is also supported with embedded CSMA/CD operations in the TDPA protocol. We performed scheduling analysis work for SPRETHER and measured performance by experiments in a prototype network.

1.2 Research Background and Related Works

1.2.1 Fundamental Scheduling Theories

1.2.1.1 RMA, EDF, and time-driven scheduling

Rate Monotonic Analysis (RMA) is a collection of quantitative methods and algorithms, with which we can specify, analyze, and predict the timing behavior of real-time software systems. RMA grew out of the theory of fixed-priority scheduling. A theoretical treatment of the

problem of scheduling periodic tasks was first discussed by Serlin in 1972 [4] and then more comprehensively discussed by Liu and Layland in 1973 [5]. They assumed an idealized situation in which all tasks are periodic, do not synchronize with one another, do not suspend themselves during execution, can be instantly pre-empted by higher-priority tasks, and have deadlines at the end of their periods. The term "rate monotonic" originated as a name for the optimal task priority assignment in which higher priorities are accorded to tasks that execute at higher arrival rates. Rate monotonic scheduling is fixed-priority task scheduling that uses a rate monotonic prioritization.

Compared to Rate Monotonic scheduling, which is a static priority scheduling method, Earliest Deadline First (EDF) is a dynamic priority scheduling method. In EDF, the task with the earliest deadline is always executed first.

When scheduling is time-driven, or clock-driven, the decisions of what jobs execute at what times are made at specific time instants. These instants are chosen before the system begins execution. Typically, in a system that uses time-driven scheduling, all the parameters of hard real-time jobs are fixed and known. A schedule of the jobs is computed off-line and is stored for use at run-time. The scheduler activates job execution according to this schedule at each scheduling decision instance.

In our integrated real-time systems model, we used a two-level scheduling approach that adopts time-driven scheduling (cyclic scheduling) for the low-level scheduler; and rate monotonic scheduling (fixed-priority driven scheduling) for the high-level scheduler. In low-level cyclic scheduling, we provide distance constraints to guarantee independence between low- and high-level schedulers. The characteristics of distance constraints ensure that at any given invocation interval of the partition and channel, they will be allocated a pre-scheduled amount of processor or communication capacity.

1.2.1.2 Aperiodic task scheduling

There have been substantial research works in the scheduling of aperiodic tasks in fixed-priority systems. Examples include the sporadic server algorithm [6], the deferrable server algorithm [7], and the slack stealer algorithm [8]. To serve aperiodic tasks in our integrated real-time systems model, we may use one of the above aperiodic servers as a dedicated aperiodic server of each partition. But, it is not a good approach, as an aperiodic server cannot claim the spare capacities earned by the aperiodic servers in other partitions.

A number of algorithms that solve aperiodic task scheduling in dynamic priority systems using an EDF scheduler can be found in the literature [9]. The Total Bandwidth Server assigns feasible priority to the arrived aperiodic task, while guaranteeing deadlines of other periodic tasks. These solutions cannot be applied to cyclic scheduling which requires distance constraints. We can find useful concepts in Shen's resource reclaiming [10], Fohler's slot-shifting algorithm [11], and J. Liu's aperiodic task scheduling in cyclic schedule [12]. However, their algorithms are also not applicable to our integrated real-time systems model because distance constraints are not considered in their scheduling models.

1.2.2 Model and Theories for Integration of Real-Time Systems

A different two-level hierarchical scheduling scheme has been proposed by Deng and Liu in [13]. The scheme allows real-time applications to share resources in an open environment. The scheduling structure has an earliest-deadline-first (EDF) scheduling at the operating system level. The second level scheduling within each application can be either time-driven or priority-driven. For acceptance tests and admission of a new application, the scheme analyzes the application schedulability. Then, the server size is determined and the server deadline of the job at the head of the ready queue is set at run-time. Since the scheme does not rely on fixed allocation of processor time or fine-grain time slicing, it can support various types of application requirements, such as release-time jitters, nonpredictable scheduling instances, and stringent timing

requirements. However, their model does not support the Integrated Modular Avionics standards, nor follow the strong partitioning concepts. To remedy the problem, our first step is to establish scheduling requirements for the cyclic schedules, such that task schedulability, under given fixed-priority schedules within each partition, can be ensured. The approach we adopt is similar to the one in [13] of comparing task execution in the SPIRIT environment with that of a dedicated processor. The cyclic schedule then tries to allocate partition execution intervals by "stealing" task inactivity periods. This stealing approach resembles the slack stealer for scheduling soft-aperiodic tasks in fixed-priority systems [8].

The scheduling approach for avionics applications under the APEX interface of IMA architecture was discussed by Audsley and Wellings [14]. A recurrent solution to analyze task response time in an application domain is derived, and the evaluation results show that there is potential for a large amount of release jitter. However, the paper does not address the issues of constructing cyclic schedules at the operating system level.

1.2.3 Real-Time Kernels

The microkernel idea met with efforts in the research community to build post-Unix operating systems. New hardware (e.g., multiprocessors, massively parallel systems), application requirements (e.g., security, multimedia, and real-time distributed computing) and programming methodologies (e.g., object orientation, multithreading, persistence) required novel operating-system concepts. The corresponding objects and mechanisms—threads, address spaces, remote procedure calls (RPCs), message-based IPC, and group communication—were more basic, and more general abstractions than the typical Unix primitives. In addition, providing an API compatible with Unix or another conventional operating system was an essential requirement; hence implementing Unix on top of the new systems was a natural consequence. Therefore, the microkernel idea became widely accepted by operating-system designers for two completely different reasons: (1) general flexibility and power and (2) the fact that microkernels offered a technique for preserving Unix compatibility while permitting development of novel operating

systems. Many academic projects took this path, including Amoeba [15], Choices [16], Ra [17], and V [18]; some even moved to commercial use, particularly Chorus [19], L3 [20], and Mach [21], which became the flagship of industrial microkernels. We refer to them as first-generation microkernel architecture.

Compared to first-generation microkernel architecture, the second-generation microkernels, such as MIT's Exokernel [22] and GMD's L4 [23] achieve better flexibility and performance by following two fundamental design concepts. Firstly, the microkernels are built from scratch to avoid any negative inheritance from monolithic kernels. Secondly, the kernels are implemented as hardware dependent, maximizing the support from the hardware. Our real-time kernel, which implements strong partitioning concepts, follows the concepts of second-generation microkernel architecture.

The number of contemporary real-time operating systems (RTOSs) has grown, both in academia and in the commercial marketplace. Systems such as MARUTI [24] and MARS [25] represent a time-triggered approach in the scheduling and dispatching of safety-critical real-time applications, including avionics. However, these systems either provide no partitioning scheme, as in the case of MARUTI; or rely completely on proprietary hardware, as in the case of MARS.

Commercial RTOSs, such as OSE, VRTX, and Neutrino, provide varying levels of memory isolation for applications. However, applications written for any of them are RTOS specific. In addition, it is not possible for applications written for one RTOS to co-exist on the same CPU with a different RTOS, without a considerable effort of re-coding and re-testing. The approach that we present in this dissertation research overcomes these drawbacks by ensuring both spatial and temporal partitioning, while allowing the integration of applications developed using a contemporary RTOS.

The Airplane Information Management System (AIMS) on board the Boeing 777 commercial airplane is among the few examples of IMA based systems [26]. Although the AIMS and other currently used modular avionics setups offer strong partitioning, they lack the ability of

integrating legacy and third party applications. In our approach, a partition is an operating system encompassing its application. All previous architectures have the application task(s) as the unit of partitioning.

The SPIRIT-µKernel cyclic scheduler can be viewed as a virtual machine monitor that exports the underlying hardware architecture to the application partitions, similar to the IBM VM/370 [27]. The concept of virtual machines has been used for a variety of reasons, such as cross-platform development [28], fault tolerance [29], and hardware-independent software development [30]. Although some of these virtual machines restrict memory access to maintain partitioning, we are not aware of any that enforce temporal partitioning or support real-time operating systems.

Decomposing the operating system services into a µ-kernel augmented with multiple user-level modules has been the subject of extensive research, such as SPIN [31,32], VINO [33], Exokernel [22], and L4 [23,24]. The main objective of these µ-kernel-based systems is to efficiently handle domain-specific applications by offering flexibility, modularity, and extendibility. However, none of these systems is suitable for hard real-time applications.

RT-Mach and CVOE are among the few µ-kernel based RTOSs that support spatial and temporal partitioning. Temporal guarantees in RT-Mach are provided through an operating system abstraction called processor reserve [35]. Processor reservation is accepted through an admission control mechanism employing a defined policy. The CVOE is an extendable RTOS that facilitates integration through the use of a callback mechanism to invoke application tasks [36]. Yet tasks from different applications will be mixed, and thus legacy applications cannot be smoothly integrated without substantial modification and revalidation.

1.2.4 Real-Time Communication Networks

We look into several commercial products available that claim to provide a real-time performance guarantee over LANs. HP's 100VG-AnyLAN [37] uses advanced Demand Priority Access to provide users with guaranteed bandwidth and low latency, and is now the IEEE 802.12

standard for 100-Mbps networking. National Semiconductor's Isochronous Ethernet [38] includes a 10-Mbps P channel for normal Ethernet traffic, 96 64-Kbps B channels for real-time traffic, one 64-Kbps D channel for signaling, and one 96-Kbps M channel for maintenance. The 96 B channels can provide bandwidth guarantees to network applications because they are completely isolated from the CSMA/CD traffic. Isochronous Ethernet forms the IEEE 802.9 standard. 3COM's Priority Access Control Enabled (PACE) [39] technology enhances multimedia (data, voice and video) applications by improving network bandwidth utilization, reducing latency, controlling jitter, and supporting multiple traffic priority levels. It uses star-wired switching configurations and enhancements to Ethernet that ensure efficient bandwidth utilization and bounded latency and jitter. Because the real-time priority mechanism is provided by the switch, there is no need to change the network hardware on the desktop machines.

On the other hand, RETHER provides real-time guarantees for multimedia applications with a software approach. It uses a timed token similar to FDDI and P1394 to control the packet transmission in the network. The tokens contain all connection information, and are passed as handshaking signals for each data packet transmission. However, their scheme cannot fulfill the requirements of safety-critical real-time systems such as integrated modular avionics (IMA) standards. For example, RETHER does not support scalable multicast real-time sessions, which are necessary for the upgrade and evolution of the IMA system, or a multi-frame cyclic schedule, which provides a highly scalable scheduling capability. Also, all message transmissions must follow the noncontention-based round-robin polling method, even if messages have no real-time communication requirements

1.3 Main Contributions

The objective of the dissertation is to design integrated real-time systems supporting strong partitioning schemes. The main contributions of the dissertation are as follows:

A model for strongly partitioned integrated real-time systems. We propose a system model for integrated real-time systems, which can be used for safety-critical real-time systems

such as integrated modular avionics systems. The model concerns (1) integration of real-time applications with different criticalities, (2) integrated scheduling of processor and message communication, and (3) guaranteeing a strong partitioning concept.

Comprehensive scheduling approaches to integrated real-time systems. We devise comprehensive scheduling algorithms for integrated real-time systems. They include (1) a fundamental two-level scheduling theory, (2) integrated scheduling of partitions and channels, (3) soft and hard aperiodic task scheduling, (4) scheduling with practical constraints, and (5) a full scheduling tool suite that provides automated scheduling analysis.

A real-time kernel for integrated real-time systems. We establish a software platform for integrated real-time systems by means of SPIRIT-μKernel. The kernel implements (1) strong partitioning concepts with a cyclic scheduler and a strong resource protection mechanism, and (2) a generic real-time operating system port interface. (3) The kernel makes it possible to run real-time applications developed in different real-time operating systems on the same processor while guaranteeing strong partitioning.

A real-time Ethernet for integrated real-time systems. We design and prototype a real-time Ethernet for integrated real-time systems by means of SPRETHER. The SPRETHER includes (1) a table-driven proportional access protocol for implementing distance-constrained cyclic scheduling, (2) support for real-time messages, (3) a scheduling algorithm to meet real-time requirements, and (4) support for non-real-time messages by integrating original CSMA/CD MAC operations into the TDPA protocol.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 considers the modeling and scheduling analysis of strongly partitioned real-time systems. A basic two-level scheduling theory is presented to solve scheduling problems of integrated real-time systems. Based on the two-level scheduling theory, we developed an integrated scheduling approach to find feasible schedules of partitions and channels on a multiprocessor network platform. Then the schedulability and

behaviors of an integrated scheduling approach were studied through simulations. We also discuss the incorporation of several practical constraints that would be required in different application domains. To help a system integrator construct a schedule for integrated real-time systems, we provide a scheduling tool suite, which automates all scheduling analysis presented in the dissertation.

Chapter 3 presents a solution for supporting on-line scheduling of soft and hard aperiodic tasks in integrated real-time systems. The Distance Constraint guaranteed Dynamic Cyclic (DC^2) scheduler is proposed, which uses three basic operations named Left Sliding (LS), Right Putting (RP), and Compacting, to dynamically schedule aperiodic tasks within a distance-constrained cyclic schedule. We also developed an admission test algorithm and scheduling method for hard aperiodic tasks. With the simulation studies, we observed that the DC^2 could result in a significant performance enhancement in terms of the average response time of soft aperiodic tasks and the acceptance rate for hard aperiodic tasks.

Chapter 4 describes the design of a real-time kernel, SPIRIT-μKernel, which implements a strong partitioning scheme based on our two-level scheduling theory. The detailed implementation issues are presented, including a generic real-time operating system port interface, which is used to host heterogeneous COTS real-time operating systems on top of the kernel. The performance of the kernel on an experimental platform is evaluated.

Chapter 5 deals with a real-time Ethernet called SPRETHER (Strong Partitioning Real-Time Ethernet), which is designed for a communication network of integrated real-time systems. The SPRETHER uses a software-oriented synchronization approach based on the table-driven proportional access protocol to overcome the lack of deterministic characteristics of Ethernet. The scheduling algorithm and performance analysis are presented. Chapter 6 summarizes the main contributions of the dissertation, and discusses future research directions.

CHAPTER 2
INTEGRATED SCHEDULING OF PARTITIONS AND CHANNELS

2.1 Introduction

One of the essential issues of real-time systems is to provide scheduling algorithms that guarantee timing constraints to real-time applications. In this chapter, we present research results related to the partition (processor) and channel (communication) scheduling in integrated real-time systems. First, we discuss the scheduling algorithm for a two-level hierarchical scheduling model that is a fundamental model of integrated real-time systems. Because the priority-driven scheduling model is the most popular and demandable in real-time systems, we adopt the priority-driven model for applications in our two-level hierarchical scheduling model. In a priority-driven model, all tasks are modeled as hard periodic tasks. To guarantee strict temporal partitioning constraints among integrated applications, we adopt a distance-constrained cyclic scheduling model for the lower-level scheduling method.

To schedule processor execution, we need to determine which partition is active and to select a task from the active partition for execution. According to temporal partitioning, time slots are allocated to partitions. Within each partition, fixed priorities are assigned to tasks based on rate-monotonic or deadline-monotonic algorithms. A low-priority task can be preempted by high-priority tasks of the same partition. In other words, the scheduling approach is hierarchical, in that partitions are scheduled using a distance-constrained cyclic schedule; and tasks are dispatched according to a fixed-priority schedule. Similar hierarchical scheduling is also applied to the communication media, where channels are scheduled in a distance-constrained cyclic fashion and have enough bandwidth to guarantee message communication. Within each channel, messages are then ordered according to their priorities for transmission.

Given task-execution characteristics, we are to determine the distance-constrained cyclic schedules for partitions and channels under which computation results can be delivered before or on the task deadlines. The problem differs from typical cyclic scheduling since, at the partition and channel levels, we don't evaluate the invocations for each individual task or message. Only the aggregated task execution and message transmission models are considered. In addition, the scheduling for partitions and channels must be done collectively using a heuristic deadline decomposition, and channel-combining algorithm such that tasks can complete their computation and then send out the results without missing any deadlines. Soft and hard aperiodic tasks are also supported efficiently by reclaiming unused processor capacities with a distance-constrained dynamic cyclic (DC^2) scheduling approach. We also give a scheduling model and solutions to practical constraints that are required by Integrated Modular Avionics standards. The scheduling algorithms are evaluated by simulation studies and implemented as an integrated scheduling tool.

The chapter is organized as follows. Section 2.2 presents an integrated real-time systems model. Section 2.3 discusses the fundamental two-level scheduling theory and provides the characteristics of the scheduling approach. Section 2.4 describes the integrated scheduling approach for partitions and channels. An evaluation of the algorithms by simulation studies is also included. The algorithms to solve practical constraints and a scheduling tool suite are described in Section 2.5 and 2.6 respectively. The conclusion then follows in Section 2.7.

2.2 Integrated Real-Time Systems Model

The SPIRIT (Strongly Partitioned Integrated Real-Time Systems) model, as shown in Figure 2-1, includes multiple processors connected by a time division multiplexing communication bus, such as ARINC 659 [3]. Each processor has several execution partitions to which applications can be allocated. An application consists of multiple concurrent tasks that can communicate with each other within the application partition. Task execution is subject to deadlines. Each task must complete its computation and send out the result messages on time in order to meet its timing constraints. Messages are the only form of communication among

applications, regardless of whether their execution partitions are in the same processor or not. For inter-partition communication, the bandwidth of the shared communication media is distributed among all applications by assigning channels to a subset of tasks running in a partition. We assume that there are hardware mechanisms to enforce the partition environment and channel usage by each application, and to prevent any unauthorized accesses. Thus, task computation and message transmission are protected in their application domain. The mechanisms could include a hardware timer, memory protection controller, slot/channel mapping, and separate channel buffers.

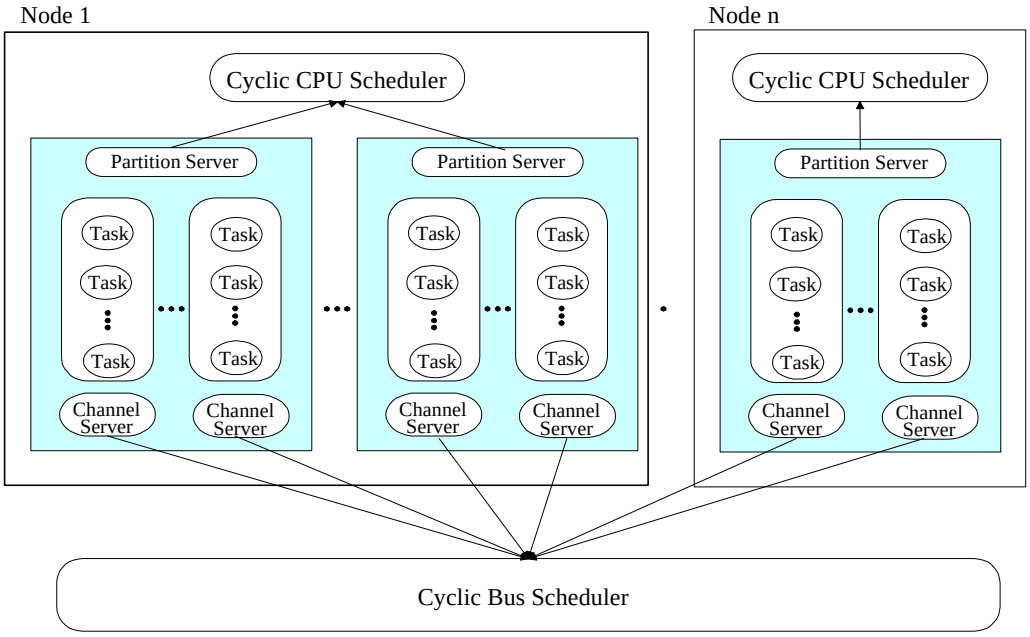


Figure 2-1 Model for Strongly Partitioned Integrated Real-Time Systems (SPIRIT)

In our task model, we assume that each task arrives periodically and needs to send an output message after its computation. Thus, as illustrated in Figure 2-2, tasks are specified by several parameters, including invocation period (T_i), worst-case execution time (C_i), deadline (D_i) and message size (M_i). Note that, to model sporadic or aperiodic tasks, we can assign the parameter T_i as the minimum inter-arrival interval between two consecutive invocations.

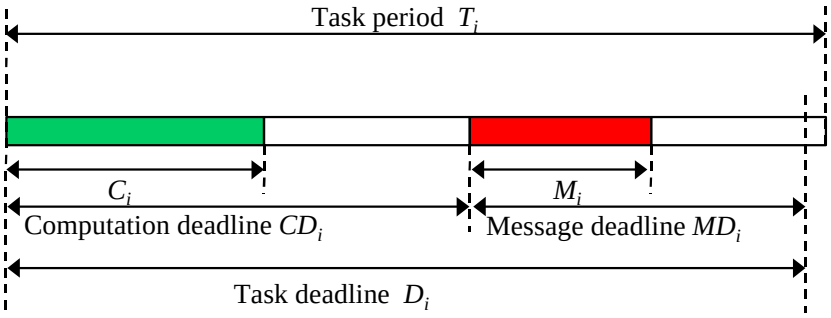


Figure 2-2 Task Model and Deadline

In order to schedule tasks and messages at processors and communication channels, the task deadline, D_i, is decomposed into message deadline (MD_i) and computation deadline (CD_i). The assignment of message deadlines influences the bandwidth allocation for the message. For example, when the message size, M_i, is 1 Kbytes, and the message deadline is 10 *ms*, then the bandwidth requirement is 0.1 Mbytes per second. In the case of a 1 *ms* message deadline, the bandwidth requirement becomes 1 M bytes per second. However, a tradeoff must be made, since a long message deadline implies that less bandwidth needs to be allocated, and the task computation has to be completed immediately.

For each processor in the SPIRIT architecture, the scheduling is done in a two-level hierarchy. The first level is within each partition server where the application tasks are running, and a higher-priority task can preempt any lower-priority tasks of the same partition. The second level is a distance-constrained cyclic partition schedule that allocates execution time to partition servers of the processor. In other words, each partition server, S_k, is scheduled periodically with a fixed period. We denote this period as the *partition cycle*, h_k. For each partition cycle, the server can execute the tasks in the partition for an interval $a_k h_k$ where a_k is less than or equal to 1 and is called *partition capacity*. For the remaining interval of $(1-a_k)h_k$, the server, S_k, is blocked. Figure 2-3 shows an example execution sequence of a partition that consists of three tasks. During each partition cycle, h_k, the tasks, t_1, t_2, and t_3, are scheduled to be executed for a period of $a_k h_k$. If

there is no active task in the partition, the processor is idle and cannot run any active tasks from other partitions.

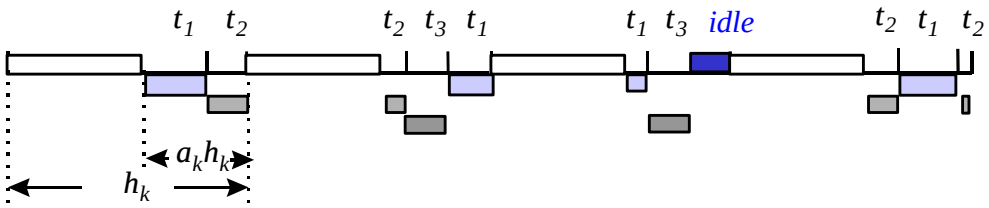


Figure 2-3 Task and Partition Execution Sequence

Similarly, a two-level hierarchical scheduling method is applied to the message and channel scheduling. A channel server provides fixed-priority preemptive scheduling for messages. Then, a distance-constrained cyclic schedule assigns a sequence of communication slots to each channel server according to its channel cycle, m_k, and channel capacity, b_k. A channel may send out messages using $b_k m_k$ slots during every period of m_k slots. Note that we use the unit of "slot" to show both message length and transmission time, with an assumption that communication bandwidth and slot length are given. For instance, a 64-bit slot in the 30 MHz 2-bit wide ARINC 659 bus is equivalent to 1.0667 ms, and a message of 1000 bytes is transmitted in 125 slots. For convenience purposes, we define the conversion factors *ST* as a slot-to-time ratio based on slot length and bus bandwidth.

Table 2-1 Index Notations used in the SPIRIT Model

Notation	Description
$n(N)$	Number of nodes in the system
$n(P_i)$, $1 \le i \le n(N)$	Number of partitions in node *i*
$n(t_{i,j})$, $1 \le i \le n(N)$, $1 \le j \le n(P_i)$	Number of tasks in partition j of node *i*
$n(Q_{i,j})$, $1 \le i \le n(N)$, $1 \le j \le n(P_i)$	Number of channel servers in partition *j* of node *i*

For all the notations shown in Table 2-2, we use the index such that $1 \le i \le n(N)$, $1 \le j \le n(P_i)$, $1 \le k \le n(t_{i,j})$, and $1 \le l \le n(Q_{i,j})$. For most of the notations, we also provide a simplified form such as A_k for $A_{i,j}$.

Table 2-2 Notations used in the SPIRIT Model

Notation	Description
N_i	Node i
$P_{i,j}$	Partition j of node i
$A_{i,j}$ or A_k	Application j of node i
$t_{i,j,k}$ or t_k	Task k of partition j of node i
$S_{i,j}$ or S_k	Periodic partition server for $P_{i,j}$
$Q_{i,j,l}$ or Q_k	Channel server l of partition j of node i
$a_{i,j}$ or a_k	Partition capacity of partition server $S_{i,j}$
$a^h_{i,j}$ or a^h_k	Partition capacity of partition server $S_{i,j}$ after specialization of partition cycle
$h_{i,j}$ or h_k	Partition cycle of partition server $S_{i,j}$
$h_{i,j}$ or h_k	Partition cycle of partition server $S_{i,j}$ after specialization
$r_{i,j}$ or r_k	Processor utilization of partition server $S_{i,j}$
$b_{i,j,l}$ or b_k	Channel capacity of channel server $Q_{i,j,l}$
$b^h_{i,j,l}$ or b^h_k	Channel capacity of channel server $Q_{i,j,l}$ after specialization of channel cycle
$m_{i,j,l}$ or m_k	Channel cycle of channel server $Q_{i,j,l}$
$m_{i,j,l}$ or m_k	Channel cycle of channel server $Q_{i,j,l}$ after specialization
$T_{i,j,k}$ or T_k	Period of task $t_{i,j,k}$
$C_{i,j,k}$ or C_k	Worst case execution time of task $t_{i,j,k}$
$D_{i,j,k}$ or D_k	Deadline of task $t_{i,j,k}$ (includes message transmission)
$CD_{i,j,k}$ or CD_k	Decomposed computation deadline of task $t_{i,j,k}$
$MD_{i,j,k}$ or MD_k	Decomposed message deadline of task $t_{i,j,k}$
$M_{i,j,k}$ or M_k	Message transmission size(#of slots) of task $t_{i,j,k}$
ST	Slot to Time conversion factor
TS	Time to Slot conversion factor

2.3 Fundamental Scheduling Theory for Integration

In this section, we discuss the fundamental scheduling theory for the SPIRIT two-level hierarchical scheduling model. We are only concerned with schedulability analysis for partitions here. The next section shows integrated scheduling of both partitions and channels.

2.3.1 Schedulability Requirement

We consider the scheduling requirements for a partition server, S_k that executes the tasks of an application partition A_k according to a fixed-priority preemptive scheduling algorithm and shares the processing capacity with other partition servers in the operating system level. Let application A_k consist of t_1, t_2,..., t_n tasks. Each task t_i is invoked periodically with a period T_i and takes a worst-case execution time (WCET) C_i. Thus, the total processor utilization demanded by the application is $r_k = \sum_{i=1}^{n} \frac{C_i}{T_i}$. Also, upon each invocation, the task t_i must be completed before its deadline period D_i, where $C_i \le D_i \le T_i$.

As modeled in SPIRIT, the tasks of each partition are running under fixed-priority preemptive scheduling. Suppose that there are n tasks in A_k listed in priority ordering $t_1 < t_2 < ... < t_n$ where t_1 has the highest priority and t_n has the lowest. To evaluate the schedulability of the partition server S_k, let's consider the case that A_k is executed at a dedicated processor of speed a_k, normalized with respect to the processing speed of S_k. Based on the necessary and sufficient condition of schedulability [40,41], task t_i is schedulable if there exists a $t \in H_i = \{ lT_j \mid j = 1,2,...i; l = 1,2,..., \lfloor D_i / T_j \rfloor \} \cup \{D_i\}$, such that

$$W_i(a_k, t) = \sum_{j=1}^{i} \frac{C_j}{a_k} \left\lceil \frac{t}{T_j} \right\rceil \le t .$$

The expression $W_i(a_k, t)$ shows the worst cumulative execution demand made on the processor by the tasks with a priority higher than or equal to t_i during the interval [0,t]. We now

define $B_i(a_k) = \max_{t \in H_i} \{t - W_i(a_k, t)\}$ and $B_0(a_k) = \min_{i=1,2,..,n} B_i(a_k)$. Note that, when t_i is schedulable, $B_i(a_k)$ represents the total period in the interval [0, D_i] that the processor is not running any tasks with a priority higher than or equal to that of t_i. It is equivalent to the level-i inactivity period in the interval [0, D_i] [8].

By comparing the task executions at server S_k and at a dedicated processor of speed a_k, we can obtain the following theorem:

Theorem 1. The application A_k is schedulable at server S_k that has a partition cycle h_k and a partition capacity a_k, if

a) A_k is schedulable at a dedicated processor of speed a_k, and

b) $h_k \le B_0(a_k)/(1-a_k)$

Proof: The task execution at server S_k can be modeled by tasks t_1, t_2,..., t_n of A_k and an extra task t_0 that is invoked every period h_k and has an execution time $C_0=(1-a_k)h_k$. The extra task t_0 is assigned with the highest priority and can preempt other tasks. We need to show that, given the two conditions, any task t_i of A_k can meet its deadline even if there are preemptions caused by the invocations of task t_0. According to the schedulability analysis in [40,41], task t_i is schedulable at server S_k if there is a $t \in H_i \cup G_i$, such that

$$\sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil + C_0 \left\lceil \frac{t}{h_k} \right\rceil \le t \, .$$

where $G_i = \{ lh_k \mid l = 1,2,\cdots,\lfloor D_i/h_k \rfloor \}$.

If t_i is schedulable on a processor of speed a_k, there exists a $t_i^* \in H_i$ such that $B_i(a_k) = t_i^* - W_i(a_k, t_i^*) \ge B_0(a_k) \ge 0$ for all $i=1,2,...,n$. Note that $W_i(a_k, t_i)$ is a non-decreasing function of t_i. Assume that $t_i^*=mh_k+d$, where $d < h_k$. If $d \ge B_0(a_k)$,

$$\sum_{j=1}^{i} C_j \left\lceil \frac{t_i^*}{T_j} \right\rceil + C_0 \left\lceil \frac{t_i^*}{h_k} \right\rceil = a_k W_i(a_k, t_i^*) + (m+1)C_0$$
$$\leq a_k \left(t_i^* - B_0(a_k)\right) + (m+1)C_0$$
$$= a_k \left(t_i^* - B_0(a_k)\right) + (m+1)(1-a_k)h_k$$
$$\leq a_k \left(t_i^* - B_0(a_k)\right) + (1-a_k)(t_i^* - d) + B_0(a_k)$$
$$= t_i^* + (1-a_k)(B_0(a_k) - d)$$
$$\leq t_i^*$$

The above inequality implies that all tasks t_i are schedulable at server S_k.

On the other hand, if $d < B_0(a_k)$, then, at $t_i' = mh_k < t_i^*$, we have

$$\sum_{j=1}^{i} C_j \left\lceil \frac{t_i'}{T_j} \right\rceil + C_0 \left\lceil \frac{t_i'}{h_k} \right\rceil \leq a_k \left(t_i^* - B_0(a_k)\right) + mC_0$$
$$\leq a_k \left(t_i^* - d\right) + m(1-a_k)h_k$$
$$= a_k t_i' + (1-a_k)t_i'$$
$$= t_i'$$

Since $t_i' \in G_i$, the application A_k is schedulable at server S_k.

When we compare the execution sequences at server S_k and at the dedicated processor, we can observe that, at the end of each partition cycle, S_k has put the same amount of processing capacity to run the application tasks as the dedicated processor. However, if the tasks are running at the dedicated processor, they are not blocked and can be completed earlier within each partition cycle. Thus, we need an additional constraint to bound the delay of task completion at server S_k. This bound is set by the second condition of the Theorem and is equal to the minimum inactivity period before each task's deadline. ■

An immediate extension of Theorem 1 is to include the possible blocking delay due to synchronization and operating system overheads. Assume that the tasks in the partition adopt a priority ceiling protocol to access shared objects. The blocking time is bounded to the longest critical section in the partition in which the shared objects are accessed. Similarly, additional delays caused by the operating system can be considered. For instance, the partition may be

invoked later than the scheduled moments since the proceeding partition just enters an O/S critical section. In this case, we can use the longest critical section of the operating system to bound this scheduling delay. These delay bounds can be easily included into the computation of $W_i(a_k, t)$.

2.3.2 Characteristics of the Two-Level Scheduling Theory

Theorem 1 provides a solution to determine how frequently a partition server must be scheduled at the O/S level and how much processor capacity it should use during its partition cycle. It is easy to see that $B_0(a_k)$ and h_k are increasing functions of a_k. This implies that if more processor capacity is assigned to a partition during its partition cycle, the tasks can still meet their deadlines even if the partition cycle increases. To illustrate the result of Theorem 1, we consider an example in Table 2-3 where four application partitions are allocated in a processor. Each partition consists of several periodic tasks and the corresponding parameters of (C_i, T_i) are listed in the Table. Tasks are set to have deadlines equal to their periods and are scheduled within each partition according to a rate-monotonic algorithm [5]. The processor utilizations demanded by the 4 partitions, r_k, are 0.25, 0.15, 0.27, and 0.03, respectively.

Table 2-3 Task Parameters for the Example Partitions

	Partition 1 (utilization=0.25)	Partition 2 (utilization=0.15)	Partition 3 (utilization=0.27)	Partition 4 (utilization=0.03)
tasks (C_i, T_i)	(4, 100) (9, 120) (7, 150) (15, 250) (10, 320)	(2, 50) (1, 70) (8, 110) (4, 150)	(7,80) (9,100) (16,170)	(1,80) (2,120)

Following Theorem 1, the minimum level-i inactivity period is calculated for each partition and for a given capacity assignment a_k, i.e., $B_0(a_k) = \min_i \max_{t \in H_i} (t - \sum_{j=1}^{i} \frac{C_j}{a_k} \left\lceil \frac{t}{T_j} \right\rceil)$. The resulting inactivity periods are plotted in Figure 2-4 for the four partitions. It is easy to see that,

when a_k is slightly larger than the processor utilization, the tasks with a low priority (and a long period) just meet their deadlines, and thus have a small inactivity period. On the other hand, when a_k is much larger than the processor utilization of the partition, the inactivity period is bounded to the smallest task period in each partition. This is due to the fact that the tasks with a short period cannot accumulate more inactivity period before their deadlines. The curves in the figure also show that an increase of a_k after the knees wouldn't make the inactivity periods significantly longer.

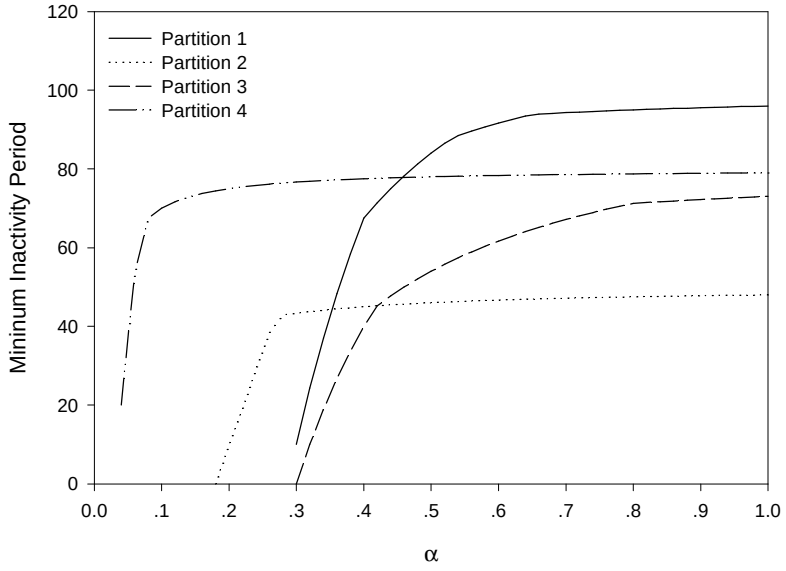


Figure 2-4 Inactivity Periods of the Example Partitions

In Figure 2-5, the maximum partition cycles are depicted with respect to the assigned capacity a_k. If the points below the curve are chosen to set up cyclic scheduling parameters for each partition at the O/S level, then the tasks in the partition are guaranteed to meet their deadlines. For instance, the curve for partition 2 indicates that, if the partition receives 28% of processor capacity, then its tasks are schedulable as long as its partition cycle is less than or equal to 59 time units. Note that the maximum partition cycles increase as we assign more capacity to

each partition. This increase is governed by the accumulation of inactivity period when a_k is small. Then, the growth follows by a factor of 1/(1-a_k) for a larger a_k.

Figure 2-5 suggests possible selections of (a_k, h_k) for the 4 partitions subject to a total assignment of processor capacity not greater than 1. From the figure, feasible assignments for (a_k, h_k) are (0.32, 36), (0.28, 59), (0.34, 28), and (0.06, 57), respectively. In the following subsection, we shall discuss the approaches of using the feasible pairs of (a_k, h_k) to construct cyclic schedules.

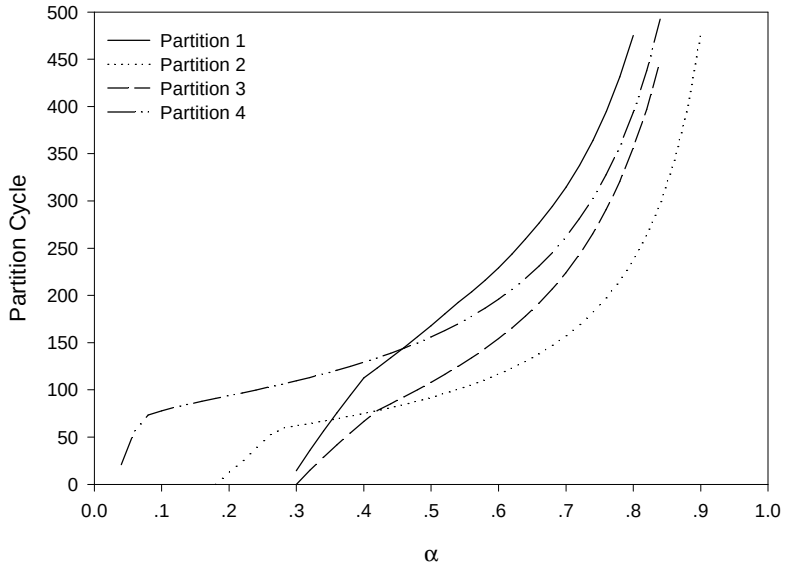


Figure 2-5 Maximum Partition Cycles for Different Processor Capacity Assignments

To reveal the properties of the parameters (a_k, h_k) under different task characteristics, we present two evaluations on the task set of Partition 1 in Table 2-3. We first alter the task deadlines such that D_i is equal to $1.0T_i$, $0.8T_i$, $0.6T_i$, and $0.4T_i$ for all tasks. The tasks are then scheduled according to the deadline monotonic algorithm within the partition. The maximum partition cycles are plotted in Figure 2-6. As the deadlines become tighter, the curves shift to the bottom-

right corner. This change suggests that either the partition must be invoked more frequently or should be assigned with more processor capacity. For instance, if we fix the partition cycle at 56 time units and reduce the task deadlines from $1.0T_i$ to $0.4T_i$, then the processor capacity must be increased from 0.34 to 0.56 to guarantee the schedulability.

The second experiment, shown in Figure 2-7, concerns partition schedulability under different task execution times. Assume that we upgrade the system with a high speed processor. Then, the application partitions are still schedulable even if we allocate a smaller amount of processor capacity or extend partition cycles. In Figure 2-7, we show the maximum partition cycles for Partition 1 of Table 2-3. By changing task execution times proportionally, the processor utilizations are set to 0.25, 0.20, 0.15 and 0.1. It is interesting to observe the improvement of schedulability when a_k is slightly larger than the required utilization. For instance, assume that the partition cycle is set to 56. Then, for the 4 different utilization requirements, the processor capacities needed for schedulability can be reduced from 0.34, to 0.28, 0.212, and 0.145, respectively. On the other hand, once a more than sufficient capacity is assigned to the partition, the maximum partition cycle is relatively independent of the processor utilization, and is mainly affected by task deadlines and periods.

2.3.3 Distance-Constrained Low-Level Cyclic Scheduling

With the result of the scheduling analysis, we can assign a fixed priority to each task, and a pair of partition capacity and partition cycle to each partition. In this subsection, we discuss how to construct a distance-constrained cyclic schedule from the result of scheduling analysis.

Given the schedulability requirement of (a_k, h_k) for each partition server S_k, a cyclic schedule must be constructed at the lower-level scheduler. Notice that the pair of parameters (a_k, h_k) indicates that the partition must receive an a_k amount of processor capacity at least every h_k time units. The execution period allocated to the partition needs to be not continuous, or to be restricted at any specific instance of a scheduling cycle.

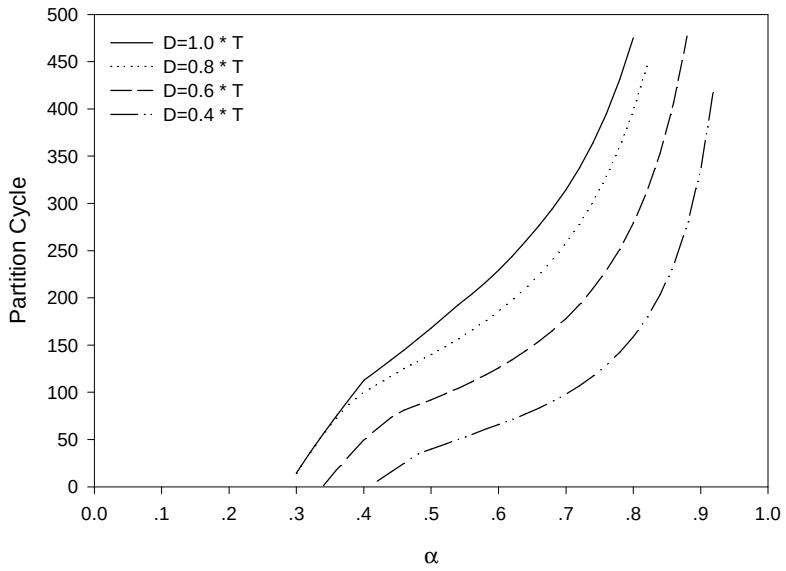


Figure 2-6 Maximum Partition Cycles for Partition 1 under Different Task Deadlines

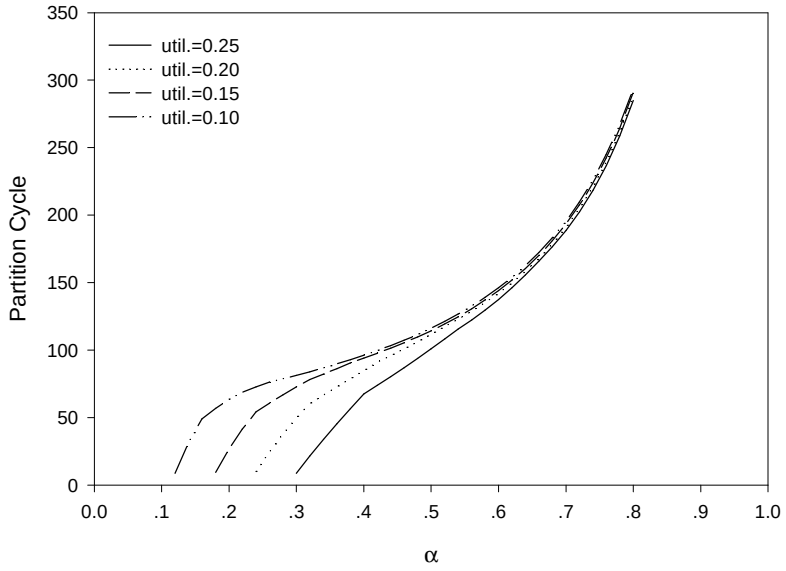


Figure 2-7 Maximum Partition Cycles of Partition 1 under Different Processor Utilizations

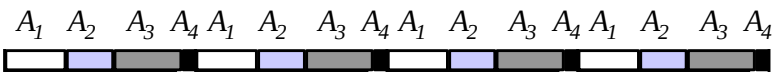
This property makes the construction of the cyclic schedule extremely flexible. In the following, we will use the example in Table 2-3 to illustrate two simple approaches.

1. *Unique partition cycle approach:* In this approach, the O/S schedules every partition in a cyclic period equal to the minimum of h_k and each partition is allocated an amount of processor capacity that is proportional to a_k. For instance, in the example of Table 2-3, a set of feasible assignments of (a_k, h_k) can be transferred from {(0.32, 36), (0.28, 59), (0.34, 28), (0.06, 57)} to {(0.32, 28), (0.28, 28), (0.34, 28), (0.06, 28)}. The resulting cyclic schedule is shown in Figure 2-8(a) where the O/S invokes each partition every 28 time units and allocates 8.96, 7.84, 9.52, and 1.68 time units to partitions 1, 2, 3, and 4, respectively.

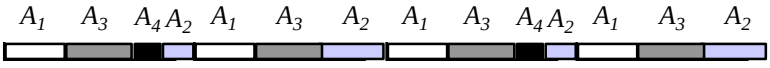
2. *Harmonic partition cycle approach:* When partition cycles are substantially different, we can adjust them to form a set of harmonic cycles in which h_j is a multiple of h_i, if $h_i < h_j$ for all *i* and *j*. Then, the O/S cyclic schedule runs repeatedly every major cycle which is equal to the maximum of h_k. Each major cycle is further divided into several minor cycles with a length equal to the minimum of h_k. In Figure 2-8 (b), the cyclic schedule for the example is illustrated where the set of (a_k, h_k) is adjusted to {(0.32, 28), (0.28, 56), (0.34, 28), (0.06, 56)}. The major and minor cycles are 56 and 28 time units, respectively. Note that partitions 2 and 4 can be invoked once every major cycle. However, after the processing intervals for partitions 1 and 3 are assigned in every minor cycle, there doesn't exist a continuous processing interval of length 0.28 * 56 time units in a major cycle. To schedule the application tasks in partition 2, we can assign an interval of length 0.22 * 28 in the first minor cycle and the other interval of length 0.34 * 28 in the second minor cycle to the partition. This assignment meets the requirement of allocating 28% of processor capacity to the partition every 56 time units.

Comparing Figure 2-8 (a) and (b), there are fewer context switches in the harmonic partition cycle approach. The reduction could be significant if there are many partitions with different partition cycles. In such a case, an optimal approach of constructing a set of harmonic cycles and assigning processing intervals should be sought in order to minimize the number of context switches. The other modification we may consider is that, when a partition cycle is reduced to fit in the cyclic schedule, the originally allocated capacity becomes more than sufficient. We can either redistribute the extra capacity equally to all partitions, or keep the same allocation in the partition for future extensions.

In addition to the flexible cycle schedules, the choice of a_k is adaptable as long as the sum of all a_k is less than or equal to 1. The parameter a_k must be selected to ensure partition schedulability at a dedicated processor of speed a_k. For instance, if the task set is scheduled according to a rate monotonic algorithm, we can define a minimum capacity equal to $r_k/n(2^{1/n}-1)$ which guarantees the partition schedulability. Additional capacity can be assigned to the partition such that the partition cycle can be prolonged. In the case that new tasks are added into the partition and the modified task set is still schedulable with the original capacity assignment, we may need to change the partition cycle and construct a new cyclic schedule at the O/S level. However, as long as the new cyclic schedule is subject to the requirement of (a_k, h_k) for each partition, no change to other partitions is necessary to ensure their schedulability.



(a) Unique Partition Cycle



(b) Harmonic Partition Cycle

Figure 2-8 Example Cyclic Schedule at the Lower Level

In distance-constrained scheduling, all partitions must be allocated an exact capacity at any instance of partition cycle period as shown in Figure 2-9. So the tasks of a partition, which arrive arbitrarily, can be allocated sufficient capacity with which they can be scheduled.

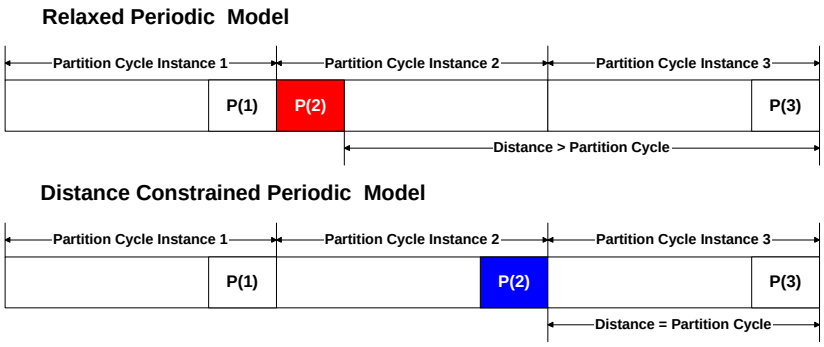


Figure 2-9 Distance-Constrained Scheduling Model

Let a feasible set of partition capacities and cycles be $(a_1, h_1), (a_2, h_2), \ldots, (a_n,h_n)$ and the set be sorted in the non-decreasing order of h_k. The set cannot be directly used in a cyclic schedule that guarantees the distance constraint of assigning a_k processor capacity for every h_k period in a partition. To satisfy the distance constraint between any two consecutive invocations, we can adopt the pinwheel scheduling approach [42,43] and transfer $\{h_k\}$ into a harmonic set through a specialization operation. Note that, in [42], a fixed amount of processing time is allocated to each task and would not be reduced even if we invoke the task more frequently. This can lead to a lower utilization after the specialization operations. For our partition-scheduling problem, we allocate a certain percentage of processor capacity to each partition. When the set of partition cycles $\{h_k\}$ is transformed into a harmonic set $\{h_k\}$, this percentage doesn't change. Thus, we can schedule any feasible sets of (a_k, h_k) as long as the total sum of a_k is less than 1.

A simple solution for a harmonic set $\{h_k\}$ is to assign $h_k=h_1$ for all k. However, since it chooses a minimal invocation period for every partition, a substantial number of context switches between partitions could occur. A practical approach to avoiding excessive context switches is to

use Han's S_X specialization algorithm with a base of 2 [42]. Given a base partition cycle h, the algorithm finds a h_i for each h_i that satisfies:

$$h_i = \eta * 2^j \le \eta_i < \eta * 2^{j+1} = 2*h_i,$$

To find the optimal base h in the sense of processor utilization, we can test all candidates h in the range of ($h_1/2$, h_1] and compute the total capacity $\sum_k a_k^h$. To obtain the total capacity, the set of h_k is transferred to the set of h_k based on corresponding h and then the least capacity requirement, a_k^h, for partition cycle h_k is obtained from Theorem 1. The optimal h is selected in order to minimize the total capacity. In Figure 2-10, we show a fixed-cyclic processor scheduling example that guarantees distance constraints for the set of partition capacities and cycles, *A(0.1,12), B(0.2,14), C(0.1,21), D(0.2,25), E(0.1,48),* and *F(0.3,50).* We use the base of 10 to convert the partition cycles to *10, 10, 20, 20, 40*, and *40*, respectively.

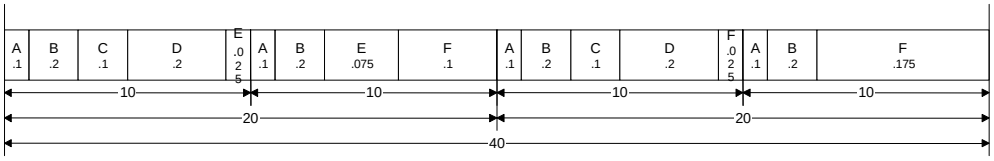


Figure 2-10 Processor Cyclic Schedule Example

2.4 Integrated Scheduling Theory for Partitions and Channels

2.4.1 Scheduling Approach

The objective of an integrated scheduling approach is to find feasible cyclic schedules for partition and channel servers which process tasks and transmit messages according to their fixed priorities within the servers. With proper capacity allocation and frequent invocation at each server, the combined delays of task execution and message transmission are bounded by the task deadlines. In Figure 2-11, we show the overall approach which first applies a heuristic deadline decomposition to divide the problem into two parts: partition-scheduling and channel-scheduling. If either one cannot be done successfully, the approach iterates with a modified

deadline assignment. We also assume that the initial task set imposes a processor utilization and a bus utilization of less than 100% and each task's deadline is larger than its execution time plus its message transmission time, i.e., $D_i \geq C_i + ST*M_i$ for task i.

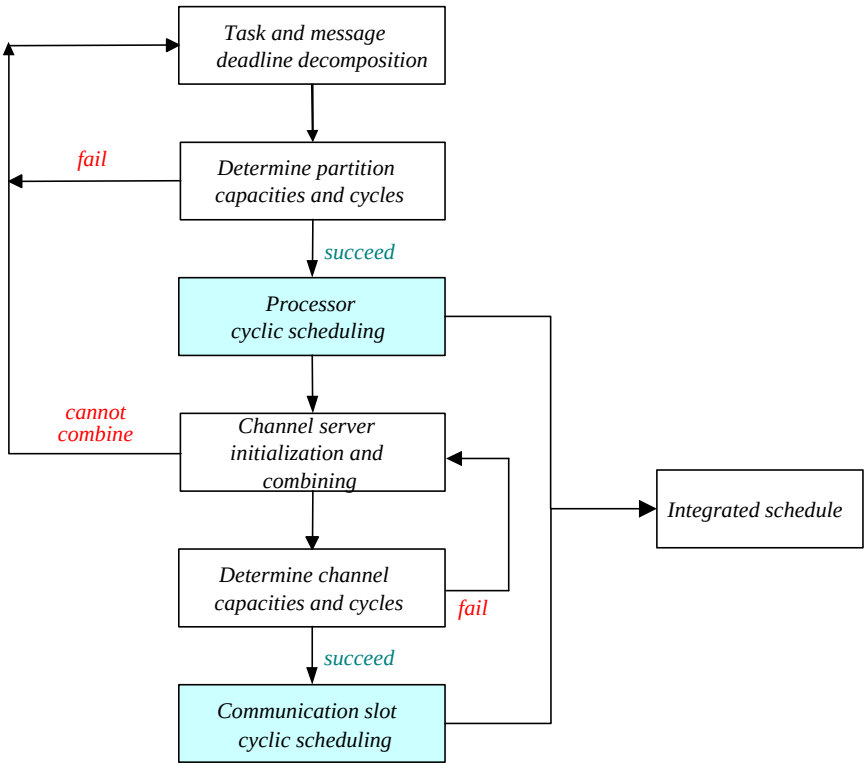


Figure 2-11 Combined Partition and Channel Scheduling Approach

2.4.1.1 Deadline decomposition

It is necessary to decompose the original task deadline, D_i, into computation and message deadline, CD_i and MD_i, for every task, before we can schedule the servers for partition execution and message transmission. A deadline decomposition algorithm is used to assign these deadlines in a heuristic way. If we assign tight message deadlines, messages may not be schedulable. Similarly, if tasks have tight deadlines, processor scheduling can fail. The following equation is used to calculate the message deadline and computation deadline for each task:

$$\text{Message Deadline, } MD_i = (D_i - \frac{ST * M_i}{C_i + ST * M_i}) f_i$$

Computation Deadline, $CD_i = D_i - MD_i$

where f_i is an adjusting factor for each task. The main idea of deadline decomposition is that it allocates the deadlines, CD_i and MD_i, proportionally to their time requirements needed for task execution and message transmission. In addition, the adjusting factor f_i is used to calibrate the computation and message deadlines based on the result of previous scheduling attempts and the utilization of the processor and communication bus. Since the message and task deadlines must be lower-bounded to the transmission time ($ST*M_i$) and computation time (C_i), respectively, and upper-bounded to D_i, we can obtain the lower bound and upper bound of the adjusting factor f as

$$\frac{1}{D_i(\frac{1}{C_i + ST * M_i})} \le f_i \le \frac{D_i - C_i}{D_i(\frac{ST * M_i}{C_i + ST * M_i})}$$

Since an adjusting factor of 1.0 is a fair distribution and always included in the range of f_i, we set the initial value of f_i to be 1. The heuristic deadline decomposition, as show in Figure 2-12, is similar to a binary search algorithm in attempting to find the right proportion of task and message deadlines. If we reach the situation that it cannot assign a new value for all tasks, we declare the input set of tasks to be unschedulable.

---

initialization for all tasks

$$MinF = \frac{1}{D_i(\frac{1}{C_i + ST * M_i})},\ MaxF = \frac{D_i - C_i}{D_i(\frac{ST * M_i}{C_i + ST * M_i})},\ f_i = 1.0;$$

iterative change of f_i when either partition or channel scheduling fails

```
if (partition scheduling fails) {
    MaxF = fi; fi = (MinF + fi) / 2.0;
}
else if (channel scheduling fails) {
    MinF = fi; fi = (MaxF + fi) / 2.0;
}
```

Figure 2-12 Deadline Decomposition Algorithm

2.4.1.2 Partition and channel scheduling

In SPIRIT, partitions and channels are cyclically scheduled. The partition cyclic schedule is based on partition cycle, $h_{k,}$ and partition capacity, a_k. Similarly, a channel cyclic schedule with parameters b_k and m_k implies that the channel can utilize $b_k m_k$ slots during a period of m_k slot interval. While tasks and messages are scheduled according to their priority within the periodic servers, the cyclic schedule determines the response time of task execution and message transmission.

We can use the same scheduling method of for channel scheduling as for partition scheduling. A channel server, Q_k, transmits its messages according to a fixed-priority preemptive scheduling method. It provides a bandwidth of $b_k m_k$ slots to the messages in the channel during every channel cycle, m_k, where $b_k \pounds 1$. For the remaining slots of $(1-b_k)m_k$, the channel server is blocked. Since each channel server follows the identical two-level hierarchical scheduling as partition servers, Theorem 1 can be directly applied to obtain the pair of parameters (b_k, m_k). However, there are several differences. First, only integer number of slots can be assigned to a channel server. Thus, we can use either $\lceil b_k m_k \rceil$ slots or restrict $b_k m_k$ to be integer. The second difference is that the message arrivals are not always periodic due to possible release jitters. Release jitters can be included in the schedulability test if they are bounded by some maximum values [44].The release jitter can also be eliminated if the communication controller incorporates a timed message service that becomes active immediately after the computation deadline is expired. The last difference is the assignment of messages into a channel. According to the principle of partitioning, tasks from different partitions cannot share the same channel for message transmission. For the tasks in a partition, we can group a subset of tasks and let them share a channel server. The grouping can be done based on the semantics of the messages or other engineering constraints. Also, the multiplexing of messages in a shared channel may lead to a saving of bandwidth reservation.

2.4.1.3 Channel combining

For a channel server that transmits a periodic message with a deadline MD_i and a message size M_i, we must allocate a minimum bandwidth of M_i/MD_i. Since there is a limitation in the total bus bandwidth, we may not always assign one channel server to each message. However, we may be able to combine some messages and let them share a common channel server. This can lead to a bandwidth reduction, since the reserved bandwidth can be better utilized by the messages of different deadlines. For example, given two messages 1 and 2 with parameters (M_1, MD_1, T_1) and (M_2, MD_2, T_2), respectively, the minimum bandwidth requirements, in terms of slots per time unit, for separate channels of messages 1 and 2, and for the combined channel, can be computed as following:

$$CB_1 = \frac{M_1}{MD_1}, CB_2 = \frac{M_2}{MD_2}$$

$$CB_{12} = \max\{\frac{M_1}{MD_1}, \frac{M_2 + M_1 * \left\lceil \frac{MD_2}{T_1} \right\rceil}{MD_2}\}$$

We assume that message 1 has a higher priority than message 2 in the computation of CB_{12}, i.e. the required bandwidth if messages 1 and 2 share a common channel server. The cost of message preemption is ignored, which can be at most one slot per preemption, since we assume that slots are the basic transmission units in the communication bus. Notice that CB_{12} is not always less than CB_1+CB_2. However, if message 1 has a much shorter deadline compared to its period and message 2 has a longer deadline than message 1's period, then the bandwidth reduction $CB_1+CB_2-CB_{12}$ becomes substantial. While we reserve a proper amount of bandwidth for an urgent message, the channel is only partially utilized if the message arrives infrequently. This provides a good chance to accommodate additional messages in the same channel and results in a reduction in the required bandwidth.

To help understanding channel combining, we give a simple example. Let's assume that there are two messages Msg1 and Msg2 with parameters (2,5,10) and (1,10,20), respectively. When we assume that deadline D_i equals perioid T_i, the message deadline MD_i must be less than T_i due to the deadline decomposition, which makes the equation, $CD_i+MD_i = D_i\ (=T_i)$, true. According to the above equations, we obtain CB_1= 2/5, CB_2=1/10, CB_1+CB_2 = 1/2 and CB_{12} = 2/5. So the bandwidth reduction is 1/10 after combining two messages in single channel. We illustrate the correctness of the channel combining in Figure 2-13 where both messages are released at the same time.

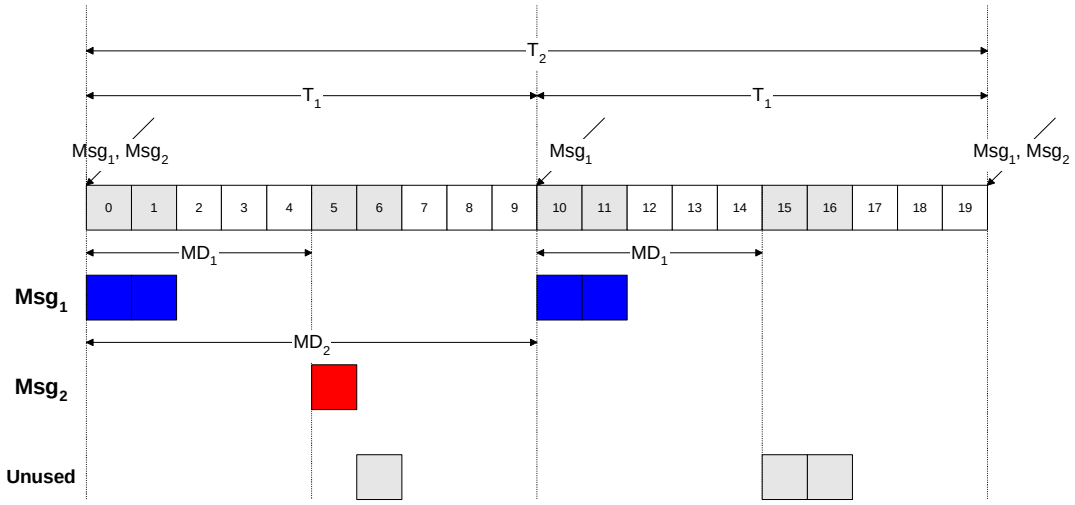


Figure 2-13 Scheduling Example of Two Messages in Combined Channel

The above equation also implies that the maximum bandwidth reduction can be obtained by combining the message with a long deadline and the message with a short deadline, where the period of the latter should be greater than the message deadline of the former. With this observation, we devise a heuristic channel-combining algorithm that is shown in Figure 2-14. The computation of the minimum bandwidth requirement of a channel consisting of messages *1,2,…,k-1*, and *k*, is:

$$CB_{12\ldots k} = \max_{j=1\cdots k}\{((\sum_{i=1}^{j-1} M_i * \left\lceil \frac{MD_j}{T_i} \right\rceil + M_j)/MD_j)\}$$

where we assume that message *j* has a higher priority then message *j+1*. Note that the real bandwidth allocation must be determined according to the choice of channel cycle as described in Theorem 1. However, in order to calculate channel cycle and capacity, the messages in each channel must be known. The channel-combining algorithm outlined in Figure 2-14 is developed to allocate messages to channels for each partition, and to reduce the minimum bandwidth requirement to a specific threshold. If the combined channels cannot be scheduled, we can further decrease the target threshold until no additional combining can be done.

---

initialization (channel combining is allowed for tasks in the same partition)

assign one channel server $\boldsymbol{Q_k}$ to the message of each task

iterate the following steps until the sum of total $\boldsymbol{CB_k}$ is less than the target threshold

determine all pairs of combinable channel server $\boldsymbol{Q_k}$ and $\boldsymbol{Q_j}$ where the max. message deadline in $\boldsymbol{Q_k}$ is larger than the min. task period in $\boldsymbol{Q_j}$

for every pair of combinable channel servers $\boldsymbol{Q_k}$ and $\boldsymbol{Q}_j$ {
calculate the bandwidth reduction $\boldsymbol{CB_k}+\boldsymbol{CB_j}-\boldsymbol{CB_{kj}}$
}
combine $\boldsymbol{Q_j}$ with the server $\boldsymbol{Q_k}$ that results in the maximum bandwidth reduction

---

Figure 2-14 Heuristic Channel Combining Algorithm

The basic method of cyclic scheduling for channel servers is the same as that of partition server scheduling. The only difference is that we need to consider that channel bandwidth allocation must be done based on an integer number of slots. Let the feasible bus bandwidth capacity allocation set be $(b_1, m_1), (b_2, m_2), \ldots, (b_n, m_n)$. Using the S_X specialization, the set $\{m_k\}$ will be transformed to a harmonic set $\{m_k\}$. Then, based on Theorem 1 and the reduced m_k, we can adjust the channel capacity b_k to $b_k^{\,h}$ subject to $\sum_{i=1}^{n}\left(\left\lceil b_i^{\,h} m_i \right\rceil / m_i\right) \le 1.0$. There will be $\lceil b_k^{\,h}\, m_k \rceil$ slots allocated to the channel server Q_k.

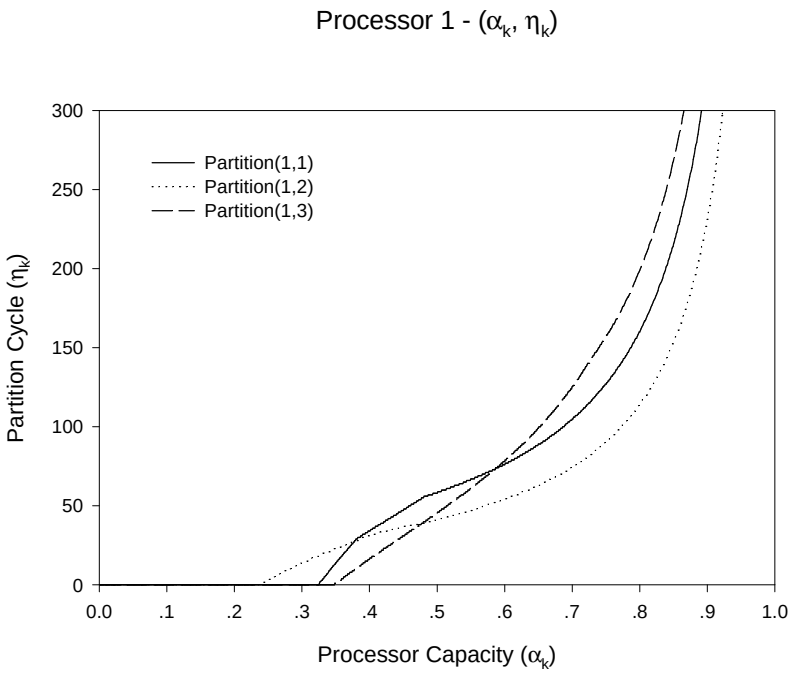
2.4.1.4 Example of integrated scheduling

To illustrate how the scheduling algorithm works, we considered the example given in Table 2-4 in which a total of five partitions are allocated to two processors. For simplicity, we assume that deadlines of tasks are equal to their periods. In this example, we considered the APEX/OASYS mission critical avionics real time system as a platform. APEX/OASYS has features such that the total bandwidth of the bus is 2,000,000 slots per second, and one time unit is capable of 2,000 slots. Therefore, the conversion factor *ST* is 0.0005 and TS is 2,000.

The processor utilizations for each processor, and bus utilizations, are also given in Table 2-4. Since both processor and bus share the same deadline of each task, schedulable utilization will be less than compared to a single resource scheduling problem. The processor utilization r and bus utilization s are given in the parenthesis below the processor and partition identifications, in the form of $(r\ ,\ s\ )$.

Table 2-4 Task Parameters for the Example of Integrated Scheduling

Processor 1 (0.65, 0.30)	Partition (1,1) (0.26,0.14)	Partition (1,2) (0.13,0.09)	Partition (1,3) (0.26, 0.07)
Tasks (C_k, M_k, T_k)	(4,*7000*,90) (9,*9000*,120) (7,*9000*,150) (15,*15000*,250) (10,*2000*,320)	(2,*2000*,50) (1,*2000*,70) (8,*14000*,110)	(7,*4000*,80) (9,*4000*,100) (10,*5000*,120)
Processor 2 (0.55, 0.17)	Partition (2,1) (0.25,0.04)	Partition (2,2) (0.30, 0.13)	
Tasks (C_k, M_k, T_k)	(5,*4000*,80) (9,*3000*,100) (11,*0*,120)	(6,*8000*,60) (10,*7000*,90) (14,*6000*,150)	



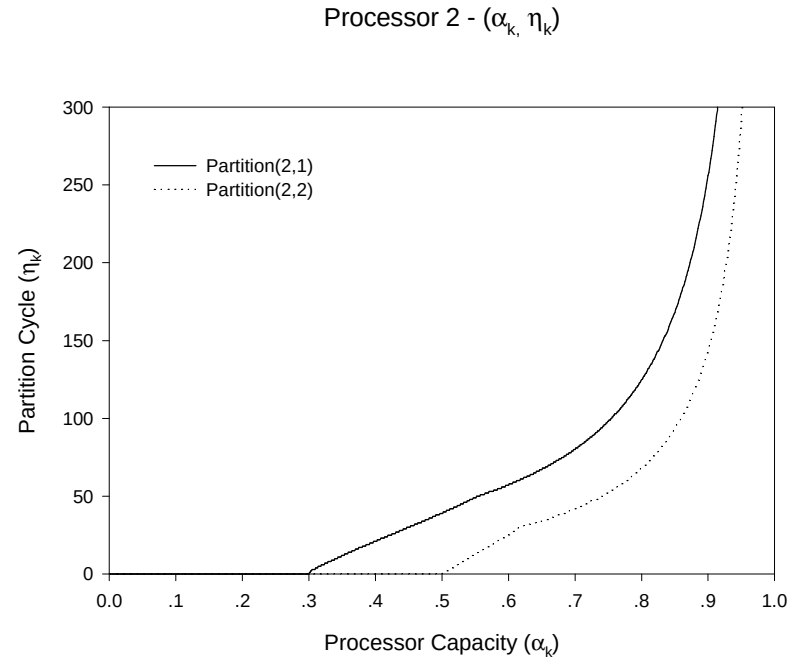


Figure 2-15 Processor Capacity vs. Partition Cycle

In Figure 2-15, the maximum partition cycles are depicted with respect to the processor capacity a_k. If the points below the curve are chosen to set up cyclic scheduling parameters for each partition in the processor cyclic scheduler, the tasks in the partition are guaranteed to meet their task deadlines. From the result of the scheduling algorithm, we can obtain sets of (a_k, h_k) as follows: (0.356, 17), (0.262, 6), (0.381, 11) and (0.375,17), (0.624,31) for each five partition. By the specialization technique of distance-constrained scheduling, we can transform the above sets to harmonic sets as follows (0.356, 16), (0.262, 4), (0.381, 8) and (0.375,15), (0.624,30) for each of the five partitions.

In Table 2-5 and Table 2-6, we show the result of cyclic scheduling for the chosen harmonic sets of parameters.

Table 2-5 Cyclic Time Schedule for Processor 1

start	0.000	1.048	4.000	5.048	5.144	8.000
	-	-	-	-	-	-
end	1.048	4.000	5.048	5.144	8.000	9.048
partition	2	3	2	3	1	2
start	9.048	12.000	13.048	13.144	15.984	
	-	-	-	-	-	
end	12.00	13.048	13.144	15.984	16.000	
partition	3	2	3	1	IDLE	

Table 2-6 Cyclic Time Schedule for Processor 2

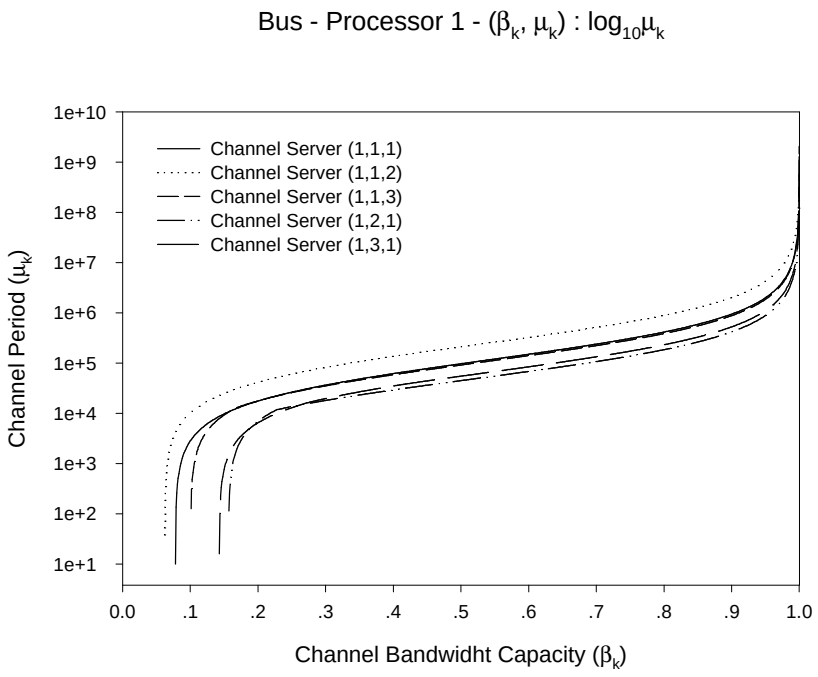
start	0.000	5.625	15.000	20.625	29.970
	-	-	-	-	-
end	5.625	15.000	20.625	29.970	30.000
partition	1	2	1	2	IDLE

In terms of message scheduling, we have sixteen messages that share the bus for message transmission. The scheduling algorithm combines tasks to reduce the whole bandwidth

requirement to fit the shared bus. In this example, the result of the message (channel) combining algorithm is: $Q_{1,1,1} = \{t_{1,1,2}, t_{1,1,5}\}$, $Q_{1,1,2} = \{ t_{1,1,4} \}$, $Q_{1,1,3} = \{t_{1,1,1}, t_{1,1,3}\}$, $Q_{1,2,1} = \{t_{1,2,1}, t_{1,2,2}, t_{1,2,3}\}$, $Q_{1,3,1} = \{t_{1,3,1}, t_{1,3,2,} t_{1,3,3} \}$, $Q_{2,1,1} = \{t_{2,1,1}, t_{2,1,2}\}$, $Q_{2,2,1} = \{t_{2,2,1}, t_{2,2,2\ ,} t_{2,2,3} \}$. In Figure 2-16, the maximum channel cycles are depicted with respect to the bus capacity b_k. In this example, since the channel server period is exponential to the capacity b_k, we use the logarithmic value instead. If the points below the curve are chosen to set up cyclic scheduling parameters for each channel server in the bus cyclic scheduler, the messages in the channel server are guaranteed to meet their message deadlines. From the result of the scheduling algorithm, we can obtain sets of (b_k, m_k) as follows: (0.079, 136), (0.063, 297), (0.101, 127), (0.157, 114), (0.144, 122) and (0.068, 172), (0.179, 101) for each nine channel servers. Applying the harmonic transformation algorithm we obtained a harmonic set of (b_k, m_k) as follows : (0.079, 101), (0.063, 202), (0.101, 101), (0.157, 101), (0.144, 101) and (0.068, 101), (0.179, 101) for nine channel servers, respectively. In Table 2-7, we show the result of cyclic scheduling in terms of slot numbers in a major frame of 202 slots.

Table 2-7 Slot Allocation of Cyclic Scheduling for Bus of Major Frame Size 202

start - end	channel	start - end	channel	start - end	channel
000 - 018	$Q_{2,2,1}$	069 - 075	$Q_{2,1,1}$	136 - 150	$Q_{1,3,1}$
019 - 034	$Q_{1,2,1}$	076 - 088	$Q_{1,1,2}$	151 - 161	$Q_{1,1,3}$
035 - 049	$Q_{1,3,1}$	089 - 100	IDLE	162 - 169	$Q_{1,1,1}$
050 - 060	$Q_{1,1,3}$	101 - 119	$Q_{2,2,1}$	170 - 176	$Q_{2,1,1}$
061 - 068	$Q_{1,1,1}$	120 - 135	$Q_{1,2,1}$	177 - 201	IDLE



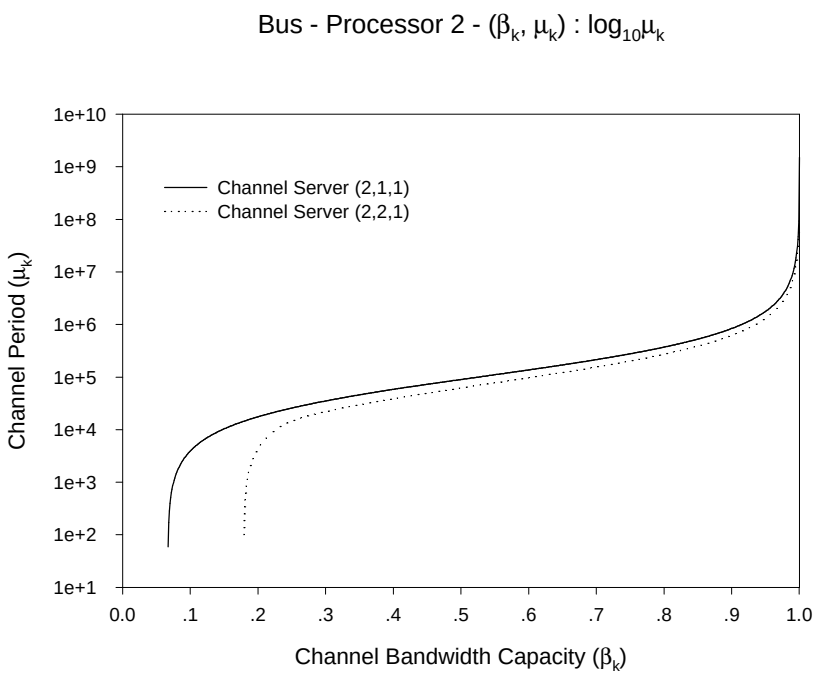


Figure 2-16 Bus Capacity vs. Channel Server Cycle

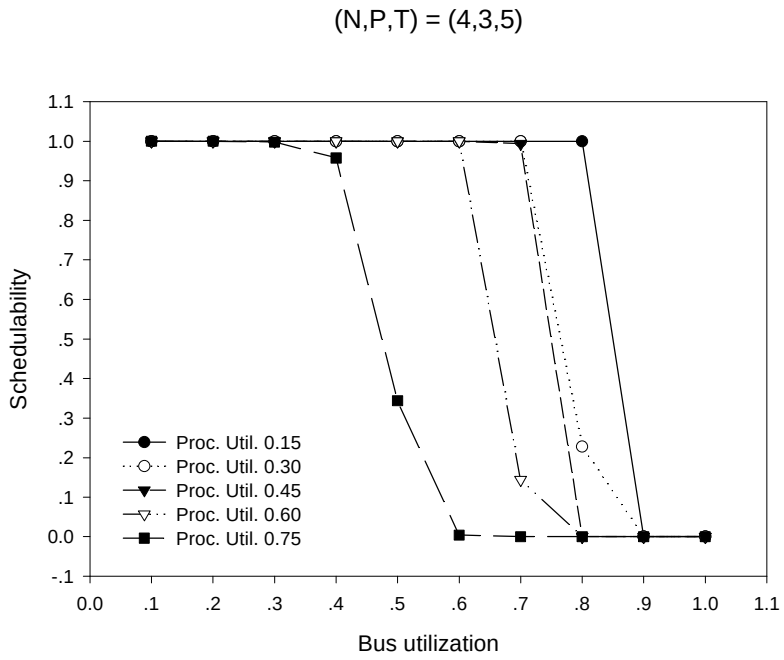
2.4.2 Algorithm Evaluation

In this section, we present the evaluation results of the integrated scheduling algorithms. Firstly, we show the percentage of schedulable task sets in terms of processor and bus utilization under the two-level scheduling, deadline decomposition and channel combining algorithms. Then, we show that the penalty of the harmonic transformation is negligibly small. Finally, the characteristic behavior of deadline decomposition is illustrated. The evaluations are done with random task and message sets that are generated with specific processor and bus utilization.

2.4.2.1 Schedulability test

A schedulability test of the algorithm is obtained using the simulations of a system model that consists of four processors, three partitions per processor and five tasks per partition, i.e., a configuration of (4, 3, 5). The simulations use random task sets that result in variable processor utilization of 15%, 30%, 45%, 60% and 75%. The task periods are uniformly distributed between the minimum and maximum periods. The total processor utilization is randomly distributed to all tasks in each processor and is used to compute the task execution times. To create message sets, we vary the total bus utilization from 10% to 90%. Message lengths are computed with a random distribution of the total bus utilization and task periods.

Using the scheduling procedure of Figure 2-11 we first assign task and message deadlines for each task. Then the partition capacity and cycle for each partition are computed and the cyclic schedule for each processor is constructed. To schedule message transmission, messages are combined into channels in order to reduce bandwidth requirement. After channel cycle and capacity are determined, a cyclic schedule is formed. For the priority schedules within partitions and channels, we adopt the deadline monotonic approach to order the task and message priorities. With all randomly created task sets, we report the percentage of schedulable task sets among all sets in Figure 2-17. The figure shows the algorithms are capable of finding proper deadline assignments and, then, determining feasible partition and channel cyclic schedules.



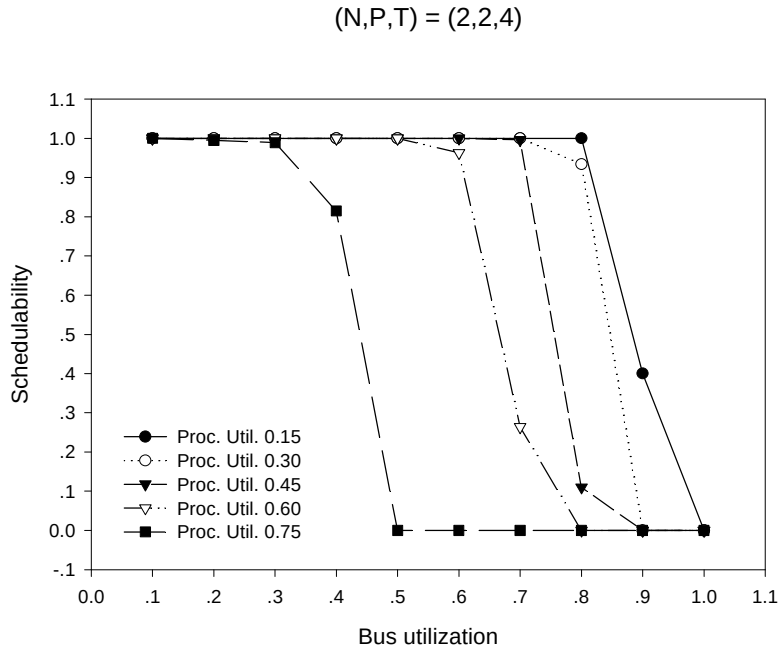


Figure 2-17 Schedulability Test for Configuration (4,3,5) and (2,2,4)

For instance, consider the case of 60% processor and bus utilization. Even if the deadlines are less than the task periods, almost 100% of task sets are schedulable. Figure 2-17 also reports the test results of the configuration (2, 2, 4). The curves have similar trends to that of the configuration (4, 3, 5).

2.4.2.2 Effects of deadline decomposition and channel combining algorithms

It is worthwhile looking into how the bus is utilized in the channel schedules resulting from the heuristic algorithms of deadline decomposition and channel combining. Consider the following measures:

1. *Measure1* is the bus utilization which equals the sum of $(ST*M_i)/T_i$ for all tasks. No real-time constraint of message delivery is considered in this measure.

2. *Measure2* is the total bus capacity needed to transmit messages on time with no channel combining (i.e., each task has a dedicated channel). This capacity will be equal to the summation of $(ST*M_i)/MD_i$ for all tasks and can be computed after message deadlines are assigned.

3. *Measure3* is the minimum bus capacity needed to schedule channels. This measure is equal to the summation of minimum b_k for all channels. Note that, according to Theorem 1, the minimum b_k for a channel is defined as the minimum capacity that results in a zero inactive period for at least one message in the channel. It can be determined after message deadlines are assigned and messages are combined into the channel.

4. *Measure4* is the total bus capacity selected according to Theorem 1. This measure can be formulated as the summation of b_k for all channels.

5. *Measure5* is the final bus capacity allocated to all channels based on a harmonic set of channel cycles and the integer number of slots for each channel. The capacity is equal to the summation of $\lceil b_k^h m_k \rceil / m_k$ for all channels.

We can expect an order of *Measure2*> *Measure5*> *Measure4*> *Measure3*> *Measure1* among the measures. *Measure2* should be much higher than other measures as we allocate bandwidth for each message independently to ensure on-schedule message delivery. With the message multiplexing within each channel, the on-schedule message delivery can be achieved with a smaller amount of bandwidth. However, a bandwidth allocation following *Measure3* cannot be practical since the channel cycles must be infinitely small. According to Theorem 1, *Measure4* contains additional capacity that is added to each channel to allow temporary blocking of message transmission during each channel cycle. Furthermore, in *Measure5*, an extra capacity is allocated as we make an integer number of slots for each channel, and construct a cyclic schedule with harmonic periods.

The simulation results of the above measures are shown in Figure 2-18. The results confirm our expectation of the order relationship. However, when we change the bus utilization from 0.1 to 0.8, the curves are not monotonically increasing (except the curve of *Measure1*). This is the consequence of the deadline decomposition (*DD*) algorithm. When channels don't have enough bandwidth to meet short message deadlines, the algorithm adjusts the factor f_k and assigns longer deadlines for message transmission. As shown in Figure 2-12, the *DD* algorithm uses an approach similar to a binary search algorithm, and makes a big increase to f_k initially. This results in long deadlines and the reduced capacity allocations in *Measure2-5*. In fact, when the bus utilization is less than 30%, the average number of iterations performed in the *DD* algorithm is slightly larger than 1, i.e., only the initial f_k is used to allocate deadlines. When the bus utilization is raised to 40% - 70%, the average number of iterations jumps to 1.6, 1.98, 2.0, and 2.04, respectively. It further increases to 11.09 when the bus utilization is set to 80%.

Figure 2-18 also illustrates the magnitude of the measures and the differences between them. The gap between *Measure3* and *Measure2* is very visible. This difference is the product of the channel combining algorithm. In order to meet a tight message deadline, we have to reserve a large amount of bandwidth. With channel combining, messages of different deadlines share the

allocated slots. As long as the message with a shorter deadline can preempt the on-going transmission, the slots in each channel can be fully utilized by multiplexing and prioritizing message transmissions.

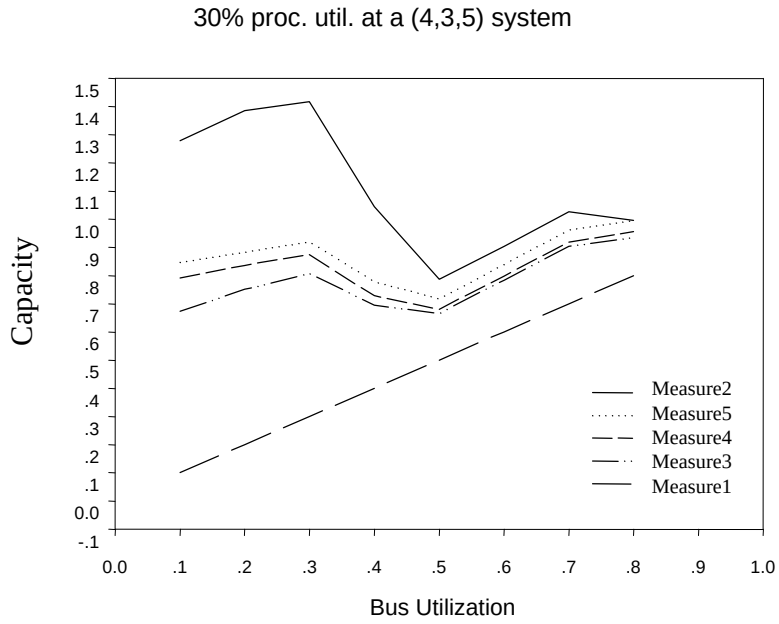


Figure 2-18 Measures for Bus Utilization and Capacities

There is a moderate gap between *Measure3* and *Measure4.* As indicated in Theorem 1, we search for a channel capacity and a channel cycle located in the knee of the curve $h_k \pounds B_0(a_k)/(1-a_k)$ after the initial sharp rise. This implies that a small increase of b_k will be added to *Measure3* in order to obtain a reasonable size of channel cycle. Finally, the difference between *Measure4* and *Measure5* is not significant at all. It is caused by the process of converting h_k to a harmonic cycle m_k, and by allocating an integer number of slots $\lceil b_k{}^h m_k \rceil$ for each channel.

The other way of looking into the behavior of the deadline decomposition algorithm is to investigate the resultant decomposition of task deadline, D_i. In Figure 2-19, we showed the average ratio of message deadline to task deadline, under different processor and bus utilization.

If the adjustment factor f_i is constant, the ratio, $\frac{MD_i}{D_i} = (\frac{ST * M_i}{C_i + ST * M_i}) f_i$, should follow a concave curve as we increase bus utilization (by increasing message length, M_i). For instance,

when the processor utilization is 15%, there are two segments of concave curves from bus utilization 10% to 70% and from 70% to 90%. The segmentation indicates a jump in the adjustment factors resulting from the deadline decomposition algorithm. In Figure 2-19, the concavity and the segmentation can also be seen in other curves that represent the message deadline ratios of different processor utilization. When the processor utilization is high, f_i may be modified gradually and partition scheduling may fail if we introduce a sharp increase to f_i. Thus, the concavity and the segmentation are not so obvious as the deadline ratio in an underutilized processor.

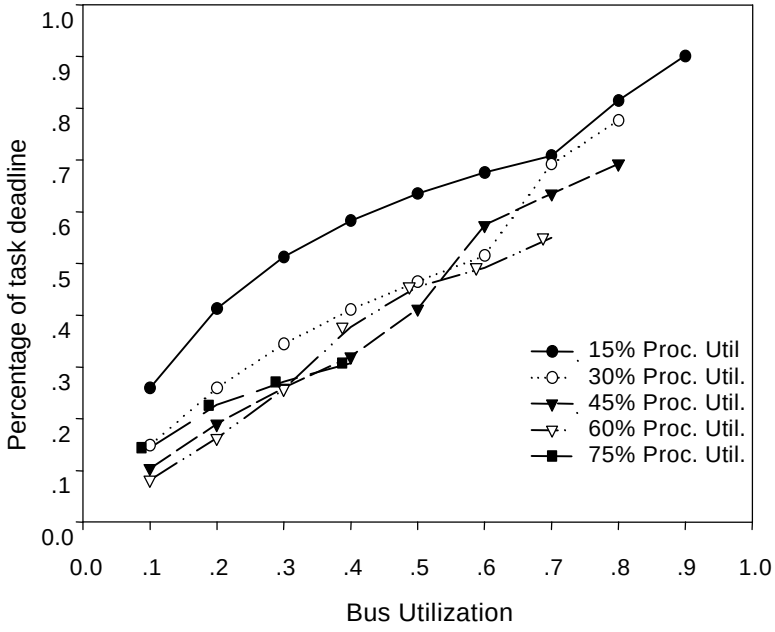


Figure 2-19 Ratio of Message Deadline to Task Deadline

2.5 Solving Practical Constraints

Since real-time systems are usually fault-tolerant, long-lived, and hardware/software restricted systems, there are practical issues to be considered as well as the fundamental scheduling problem. In this section, we present six additional practical issues and propose enhanced scheduling algorithms to solve these problems. To achieve efficient and economic lifetime maintenance of the IMA systems, we provide incremental changing algorithms. To

support replication based fault tolerance, we supplement replication scheduling algorithms. For meeting clock synchronization requirements, we add fault tolerant cyclic slot allocation algorithms. To solve the practical constraints due to hardware and software limitations, we have also devised algorithms for fixed bus major frame size, fixed message size, and time tick based scheduling.

2.5.1 Incremental Changing Scheduling

Since an aircraft can usually be in service for several tens of years, it is very likely that the system will be upgraded during its lifetime. In the current practice and FAA requirement, validation and certification processes for the whole system must be done after each upgrade. Incremental changing algorithms are to seek a minimal modification of the previous schedule when a subsystem is changed. According to the algorithms, it is just needed to re-allocate the resources for either modified partitions or new partitions only. We can reuse the original schedule such as processor capacity, channel allocation, channel bandwidth, major and minor periods, etc., for unaffected partitions and channel servers. Thus, we can avoid significant system re-design efforts and reduce the costs for re-certification. We define the parameters for describing incremental changing algorithms in Table 2-8.

Table 2-8 Notations for Incremental Changing Algorithm

Notation	Description
$avPC_i$	available processor capacity of node i $(1-\sum_{i=1}^{n} a_i^h)$
$avCC$	available communication bus capacity $(1-\sum_{i=1}^{n} b_i^h)$
$HP\text{-}P_i$, $HP\text{-}C$	major (hyper) period of each node i and communication bus
$mp\text{-}P_i$, $mp\text{-}C$	minor period of each node i and communication bus
avp_{ij}	accumulated available free capacity for harmonic period size j of node i
avc_k	accumulated available free capacity for harmonic period size k of communication bus

We introduce the concept of *available capacity* of each harmonic period for both processor and bus cyclic scheduling. These available capacities can be assigned to newly attached partition and channel servers. To achieve the flexibility in cyclic scheduling and allocation of new servers, we have to carefully manage the available capacities. For example in Figure 2-20, four partition servers are allocated processor capacities of (1/7, 7), (1/7, 14), (1/7, 14), and (1/7, 28) respectively. We evenly distribute the available capacities, 3/7, to every harmonics periods, 7, 14, and 28. In this example, we can obtain the value of parameters such that 3/7 for $avPC_i$, 1/7 for avp_{i7}, 4/14 for avp_{i14} and 12/28 for avp_{i28}. AS_k stands for available server for period k.

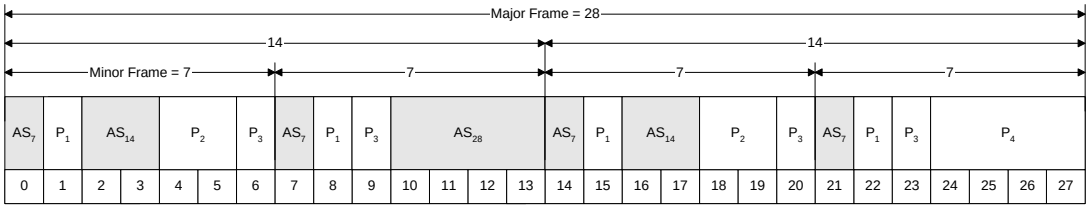


Figure 2-20 Cyclic Schedule Example for Incremental Changing

2.5.1.1 Attach algorithm

The *attach* algorithm finds a feasible schedule for a new partition while not affecting the original schedule of existing partitions. Since the *attach* algorithm has a flow similar to the basic two-level scheduling algorithm, we discuss only the necessary modifications in this subsection.

In terms of utilization, the sum of processor and bus utilization of all tasks in the new partition must be less than the current available capacity of node i, $avPC_i$ and *avCC* respectively. There should be modification in selecting a pair of partition capacity and cycle, (a_k, h_k), from schedule-graph (a_k, h_k), which is obtained from Theorem 1, of the new partition. From the schedule-graph, we choose possible candidates for a set of (a_k, h_k), that satisfies the following criterion.

[***Criterion* 1**.] Set of (α_k, η_k) such that $\eta_k \mid HP\text{-}P_i$, $h_k \geq mp\text{-}P_i$, and $\alpha_k \leq avp_{ihk}$

For example, we assume that we have found two candidates (1/7, 7) and (3/14, 14) for a new partition. The longer the partition cycle, the higher the partition capacity required. Since the periods 7 and 14 are included in the original harmonic set {7, 14, 28} and there is sufficient available processor capacity for the two periods, we can insert the new partition into the original schedule. Figure 2-21 shows the result of inserting P_{new}(1/7, 7) and P_{new}(3/14, 14) in (a) and (b) respectively. The selection between them is up to the system designer. But, in this example, P_{new}(1/7, 7) is better in terms of processor utilization.

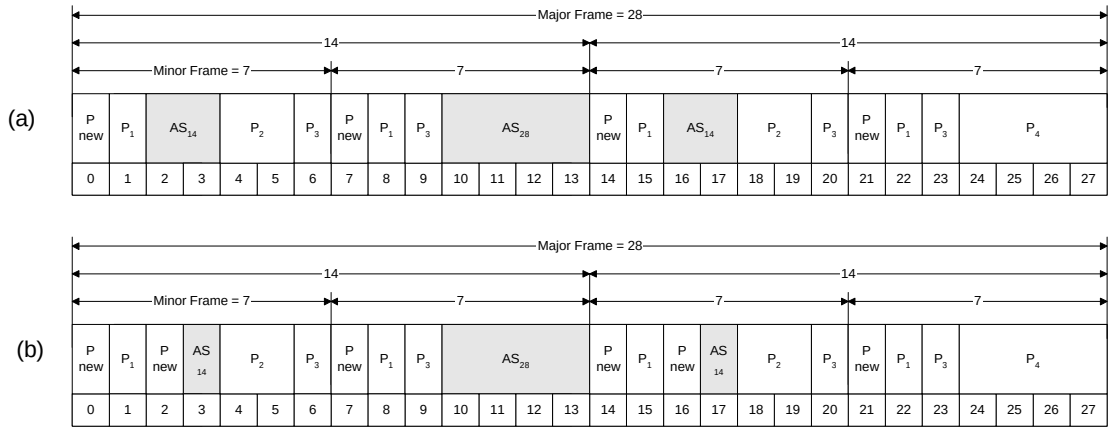


Figure 2-21 Example of Attaching New Partition

After processor (partition) scheduling, we assign a channel server to each message (task). And we set the requested channel bandwidth for the new partition to *CThres*. *CThres* is an important parameter of our channel combining algorithm. It is a maximum allowable bus utilization for the partition. Obviously, *CThres* must be less than the available bus capacity, *avCC*. We combine channel servers until the sum of utilization does not exceed *CThres*. If it succeeds in reaching *CThres*, we can proceed to the channel schedulability check and channel slot allocation.

The following criterion is used to choose a candidate set of (b_k, m_k) from the schedule-graph of channel cycle, m_k, vs. the channel capacity, b_k , after channel combining.

[***Criterion* 2.**] Set of (b_k, m_k) such that $m_k \mid HP\text{-}C$, $m_k \geq mp\text{-}C$, and $\lceil b_k * m_k \rceil \leq \lfloor avp_{imk} * \mu_k \rfloor$

As a result, we can obtain a candidate set of (b_k, m_k) for each channel server. The next step is to select a feasible (b_k, m_k) from the candidate set for each channel server. We used DFS (Depth First Search) to search for a feasible solution. Let's assume that we have *n* channel servers, and each channel server has a multiple number of feasible candidate pairs of (b_k, m_k).

Firstly, we build a graph from root level 0 to leaf level n. The root node at level 0 is a virtual initiating node that keeps full available bus capacities per each harmonic period. From level 1 to n-1, each level i has nodes that represent the channel server candidate set of (b_i, m_i). At level n, the channel server n's feasible candidate set will be constructed as leaf nodes. We can construct a graph by connecting every node except leaf nodes to all the nodes at the next level. Secondly, the search starts from the root until it reaches a leaf node. As a search goes into a node at the next level, it determines whether this node can be included in the current feasible set or not by comparing the required capacity of this node and that of the parent node's available capacity. If a check fails, it returns to the upper level and continues DFS. If it succeeds, it allocates the current (b_i, m_i) of the node to the corresponding channel server i. The remaining available capacities of the current node for all harmonic periods are set by copying from a parent node. Only the capacity for the period that is assigned to the current channel server *i* will be adjusted by subtracting the current assigned capacity from that of the parent's. Finally, if it reaches any leaf node, the current feasible set, from root to this leaf node, is one of the feasible sets. We can select an optimal feasible set from these feasible sets.

2.5.1.2 Detach & modify algorithms

The detach algorithm is used to remove capacity allocation from the cyclic schedule. The *detach* algorithm is relatively simple, for two reasons. Firstly, there is no dependency among partition servers, so we de-allocate the processor capacity originally allocated to the partition being detached. Secondly, the channel server only serves the messages in the same partition so we can de-allocate the bus capacity allocated to the channel servers belong to the partition.

To maintain correct system information after *detach*, we adjust the system parameters as follows. Let's assume that the detached partition had a processor capacity allocation of (a_k, h_k).

$$avP_i = avP_i + a_k$$

$$avp_{ihj} = avp_{ihj} + a_k, \text{ where } h_j \geq h_k \text{ and } h_j \leq HP\text{-}P_i$$

The basic *modify* algorithm is performed by sequential applications of the *detach* and *attach* algorithms. First it removes the old partition by the *detach* algorithm. Then it attaches the changed partition as a new partition. Since the two-level scheduling algorithm can give extra capacity to servers if necessary, we do not need to change the original schedule when the existing capacity is enough.

In the case of deleting tasks from the existing partition, we do not need to change allocation. By deleting tasks, the existing partition is over provided with processor capacity. It is also possible that as we delete tasks, some channel servers become empty. We can either leave it as it is or remove those channel servers. If we decide to remove those channel servers, we should de-allocate bus capacity and adjust *avCC* and corresponding avc_k. Now, let's look into the case of changing parameters of the original tasks. Since we do not delete any partition server or channel servers in this case, we can leave the original schedule if all schedulability checks pass with changed parameters.

2.5.2 Replicated Partition Scheduling

To achieve fault tolerance, safety-critical real time systems usually adopt a redundancy mechanism by replicating the application on multiple processors. It is required that the replicated partition instances in the same group must be guaranteed to meet the same real time constraints while executing on different processors. In Figure 2-22, we show an example of two replicated partition groups in an IMA system composed of three multiprocessors.

For simple explanation, we assume that each group possesses only one channel server, Ch. 1 and Ch.2 respectively. The replicated partition 1 must be feasibly scheduled in both node 1

and node 2. In the processor execution model, each replicated partition instance in node 1 and node 2 may be scheduled differently because they must be run in different node configurations.

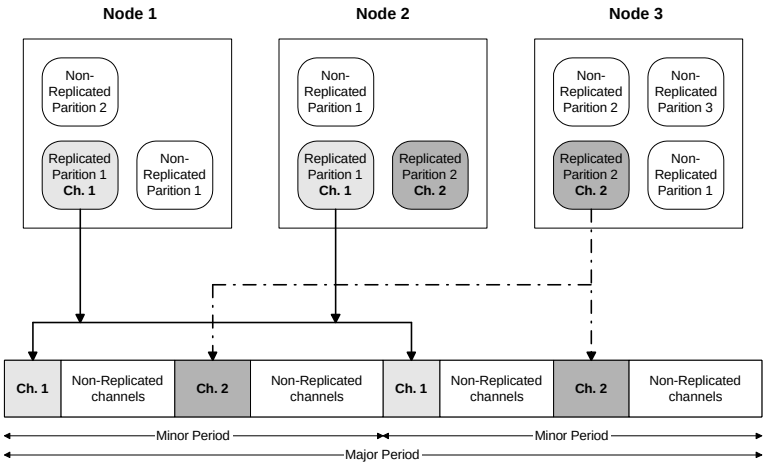


Figure 2-22 Replicated Partitions in IMA Systems

However, in the case of channel scheduling, both instances should be scheduled in the same manner because they have to share one channel server, Ch. 1. If we only consider replicated partition 1, the scheduling objective is to find a feasible processor schedule (α_k, η_k) and (α'_k, η'_k) for node 1 and node 2, and channel schedule (β_k, μ_k) for Ch.1. As long as these three capacity assignments for a replicated partition 1 are kept in global system scheduling, both replicated partition instances will meet the same timing constraints.

The overall scheduling takes two steps; replicated partition scheduling and global scheduling.

Firstly, we schedule each replicated partition group with expected processor and bus utilizations. We apply the basic two-level scheduling algorithm to replicated partitions as if total processor and bus capacities are the same as the expected processor and bus utilizations. The following heuristic equation is used to find the expected replicated-partition processor and bus utilization for each replicated partition group. With these utilization bounds, the scheduler makes its best efforts to find the feasible schedule for replicated partitions. But, there is still a chance to

miss the expected utilization of all replicated partitions. We allow this minor miss, because we can achieve the total system utilization bounds in non-replicated partition scheduling later.

$$ERPU_k = \min_{i=1}^{n(N)} \{TPU_i \cdot \frac{PU_k}{\sum_{j=1}^{n(P_i)} PU_{i,j}}\}$$

$$ERBU_k = \{TBU \cdot \frac{BU_k}{TotalPureBusUtil.}\}$$

The expected processor utilization of replicated partition k, $ERPU_k$, can be obtained by taking the minimum utilization requirement among all nodes for the replicated partition k. TPU_i is the total target processor utilization of the node, with a default of 100%. $PU_{i,j}$ is the processor utilization of partition j of node i. PU_k is the processor utilization of the corresponding replicated partition group. The expected bus utilization of replicated partition k, $ERBU_k$, can be obtained by taking the proportional utilization of the replicated partition from the total pure bus utilization. We only count the bus utilization of the replicated partition group once in calculating the total pure bus utilization. This is because replicated partition instances in the same group share one channel.

With the two calculated parameters, $ERPU_k$ and $ERBU_k$ per replicated partition group, we can find feasible schedules, i.e., schedule-graphs of (α_k, η_k) per replicated partition server and (β_k, μ_k) per channel server, using the basic two-level scheduling algorithm. While replicated partition scheduling, every deadline of each task is decomposed into computation and message deadline, and messages are combined properly to achieve expected bus utilization.

Secondly, we schedule the remaining non-replicated partitions without changing the results of the replicated partition scheduling, such as deadline decomposition, priority assignment, and channel bandwidth allocation, etc. It means that we use the same schedule-graphs of (α_k, η_k) and (β_k, μ_k) that were obtained in replicated partition scheduling. The global

scheduling algorithm finally selects feasible (α_k, η_k) and (β_k, μ_k) for all replicated partition and channel servers from the schedule-graphs while scheduling non-replicated partitions.

2.5.3 Fault Tolerant Cyclic Slot Allocation for Clock Synchronization

The ARINC 659 fault tolerant TDM bus is designed for safety-critical IMA systems that require very high reliability. It is equipped with four redundant data serial buses and four clock lines for fault tolerance. Clock synchronization is achieved by bit- and frame-level synchronization messages in a distributed way. In this subsection, we propose a fault tolerant cyclic slot allocation algorithm that can be used in the distributed clock synchronization environment adopted in ARINC 659.

We assume that in our target distributed clock synchronization environment, when one of the bus controllers gets bus mastership and transmits either real or idle data, it also distributes a reference clock to all other bus controllers simultaneously. Other bus controllers try to synchronize their local clock to the current reference clock. This approach might cause global system clock failure if the current bus master fails and no reference clock is provided for longer than a certain threshold time. Our new algorithm ensures that a bus master cannot be allocated slot time longer than this threshold. Also, the failed node is prevented from re-gaining bus mastership within another pre-determined threshold time. With this approach, bus synchronization can be recovered from single node failure without protection switching.

We use two parameters to specify the reliability requirement of a distributed clock synchronization mechanism and introduce Idle Clock Synchronization Channel Server (ICSCS). The two parameters are *MaxH* (Maximum Clock Holding Time) and *MinR* (Minimum Clock Re-sync Time) that are decided by the reliability requirement of the implemented hardware. The two parameters imply that the number of consecutive slots allocated to a single channel must be less than or equal to *MaxH* and be greater than or equal to *MinR*.

We present the basic idea with a simple example in Figure 2-23. In the example, there are three nodes, which have a channel server of bandwidth *CH1* (2,10), *CH2* (3,10), and *CH3* (3,20) respectively. The two parameters, *MaxH* and *MinR*, are both 1.

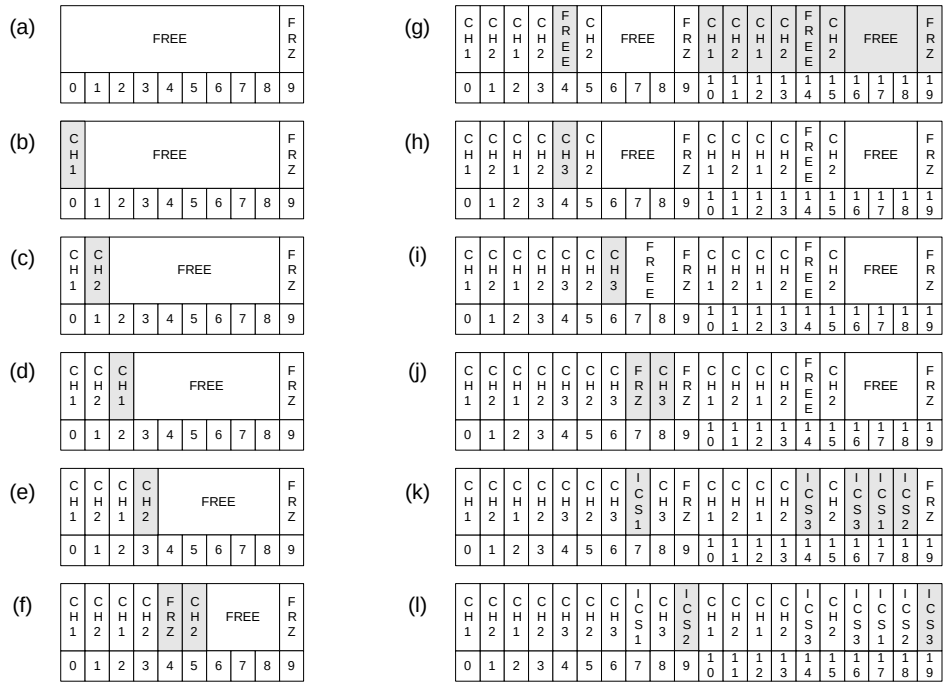


Figure 2-23 Example of Fault Tolerant Cyclic Slot Allocation

The basic idea of the algorithm is that it allocates slots to the channel servers in non-decreasing order of channel cycle. During allocation, it maintains a temporary bus schedule *List* that is an ordered set of allocated intervals. The *List* is initialized with only two interval entries. The first entry is a type *FREE* interval of size 9 (minimum period – *MinR*) and the second is a type *FREEZE* interval of size 1, i.e., *MinR*, as in Figure 2-23-(a). For the channel severs of the minimum channel cycle,i.e., *CH1* and *CH2*, we put their allocation interval into the *FREE* interval while meeting *MaxH* and *MinR* requirements, as in Figure 2-23-(b)(c)(d)(e). If it is not possible to allocate any channel server of the corresponding channel cycle for the specific slot interval due to the *MaxH* and *MinR* requirement, we temporally designate it as a type *FREEZE*, as in Figure 2-23-(f). After finishing the current channel cycle, the scheduling list is copied to fit

with the length of the next channel cycle as in Figure 2-23-(g). When we copy the *List*, we reset the intervals of type *FREEZE* that were generated in the previous allocation to *FREE*. Thus, the newly freed intervals can be allocated for the next channel cycle as in Figure 2-23-(h). Figure 2-23-(k) and (l) show that non-allocated FREE and FREEZE slot intervals are assigned to an ICSCS of each node properly. The allocation can be done easily, when we maintain two temporal vector variables *CAS* and *waitSlots* for each dynamically created time interval. *CAS* stands for consecutively allocated slots, and *waitslots* stands for the wait time after yielding bus mastership for that interval per node.

2.5.4 Miscellaneous Practical Constraints

2.5.4.1 Fixed bus major frame size scheduling

In the original scheduling algorithm, we tried to find the optimal major frame size for TDM bus with given channel bandwidth requirements. But, in a practical situation, it is also needed to schedule channels with a fixed major frame size, because major frame size can be tightly coupled with bus hardware design. In fixed bus major frame size scheduling, it is limited to choosing the base minor frame size, denoted as μ, in distance-constrained cyclic scheduling. The algorithm requires that the fixed major frame size must be multiples of the base minor frame size. The following algorithm is to find a feasible pair of channel capacity and cycle for each channel server to satisfy the major frame size constraint. We formalized a given problem such that:

(input) MF (Major Frame Size), a set of $(b_1, m_1), (b_2, m_2), \ldots, (b_n, m_n)$

(output) set of $(b^*_1, m_1), (b^*_2, m_2), \ldots, (b^*_n, m_n)$, $m_i \mid m_j$, for $i < j$ and $m_i \mid MF$,

$$i=1,2,\ldots,n, \text{ where } \sum_{i=1}^{n} \left\lceil b_i^* \cdot m_i \right\rceil \cdot \frac{1}{m_i} \le 1.0$$

We show the self-explanatory algorithm in Figure 2-24.

---

for ($\mu=\mu_1$; $\mu >= \mu_1 /2$; μ--) { // check all possible candidates of μ
  if (μ divides MF) { // base minor frame size must divide MF
    // finds MPS (Multi-Periods Set)
    MPS = $\{\mu*2^i \mid \mu*2^i <= \mu_n$ AND $\mu*2^i$ divides MF, i=0,1,2,…$\}$;
    set $m_i = \max_{m \in MPS}\{ m \mid m <= \mu_i\}$, i=1,2,…,n;

  }
  $if\ (\sum_{i=1}^{n} \lceil b_i^* \cdot m_i \rceil \cdot \frac{1}{m_i} \le 1.0)\ found = TRUE$
}

Figure 2-24 Cycle Transformation Algorithm for Fixed Bus Major Frame Size Scheduling

2.5.4.2 Fixed message size scheduling

Since each channel server schedules its assigned messages in a fixed-priority driven method, the logical architecture of a channel server can be depicted as in Figure 2-25. A channel server schedules the current set of arrived messages and puts the highest priority message into the top of the buffer. One of the important design criteria is to select a fixed buffer size, MSIZE, in Figure 2-25. In the case of ARINC 659, the minimum resolution of MSIZE supported by underlying hardware is a word (32bits). However, it may be impractical to choose a word for MSIZE, provided that a bus is much faster than a channel server. So depending on the bus speed and performance of the channel server, we must decide on a practically feasible MSIZE. In general, the smaller the preemption unit, the better schedulability we can achieve. It is also improper to permit a larger MSIZE because it can cause higher-priority messages to wait for the completion of lower-priority message of intolerably long size. It is an important job of the system engineer to find the optimal MSIZE.

In the original message scheduling algorithm, we assume that the unit of TDM time slot (32bit data in ARINC 659) is a preemption unit for message scheduling, i.e., MSIZE equals to 1. The fixed message size scheduling algorithm permits arbitrary values of MSIZE.

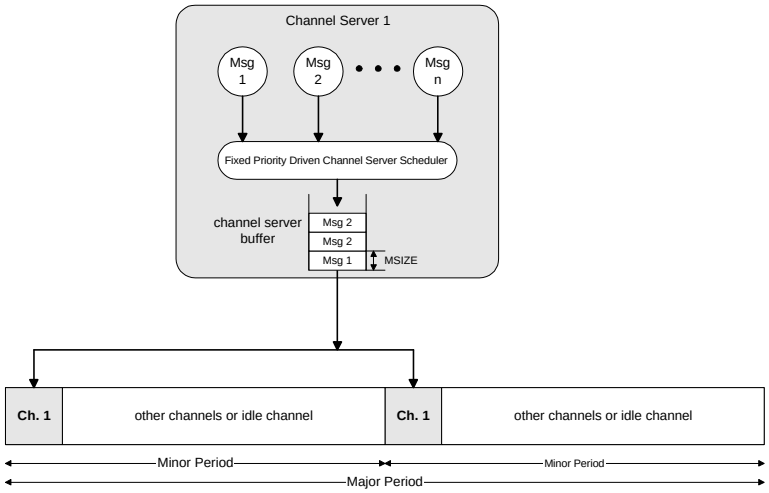


Figure 2-25 Logical Channel Server Architecture

The difference from the original schedulability condition is caused from that the higher-priority messages should wait for MSIZE amount of time, in worst case, to allow a formerly dispatched lower-priority message be completed. The modified scheduling equation, $W_i(s)$, is as follows.

$$W_i(s) = \sum_{j=1}^{i-1}\left(M_j \cdot \left\lceil \frac{s \cdot ST}{T_j} \right\rceil \right) + M_i + MSIZE \le \lfloor s * b_k \rfloor$$

2.5.4.3 Time tick based scheduling

The basic two-level scheduling algorithm assumes a high-precision real clock driven method for processor scheduling. For example, the basic two-level scheduler is allowed to allocate any real value for execution time, like 2.3571 (ms) in the partition cycle of 10 (ms) for a partition. But most practical systems, like Honeywell's GPACS IMA system, use a fixed resolution of time tick method to schedule partitions. Similar to the message scheduling algorithm that uses a fixed slot as a scheduling unit, the processor scheduling algorithm must be modified to allocate an integral number of time ticks to each partition. The disadvantage of the time tick based scheduling is that its processor utilization is poorer than that of the high-precision real

clock driven method, because it has to allocate time based on time ticks of relatively low resolution.

2.6 Scheduling Tool

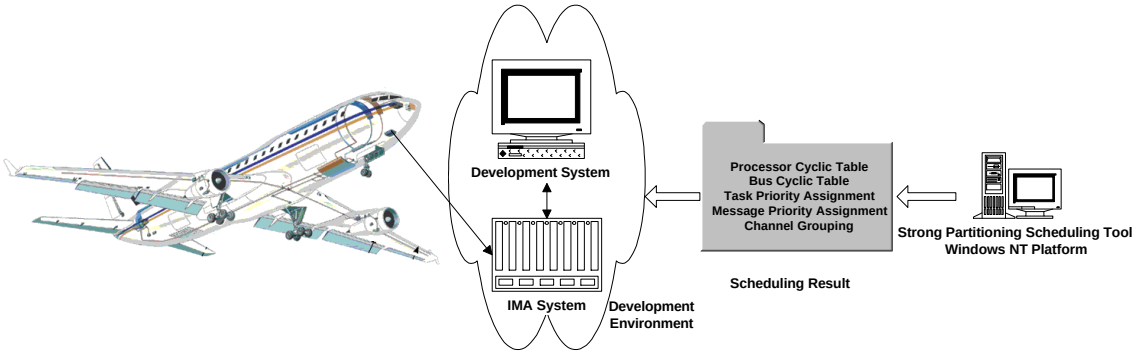


Figure 2-26 Usage of Scheduling Tool in Avionics System Design

We have developed a GUI-based integrated scheduling tool which implements all previously described scheduling algorithms for IMA systems on the Windows NT platform. Figure 2-26 shows the role of the tool in developing IMA systems. For guaranteeing timing requirements of all integrated real-time applications in the IMA system, system integrators need information such as the processor cyclic scheduling table in each processor, task priority assignment in each partition, message to channel server mapping, bus cyclic scheduling table, and message priority assignment in each channel. The tool prepares this scheduling information at the system integration stage. The tool provides convenient user interface functions, such as managing projects, editing files, performing scheduling algorithms, and browsing scheduling results.

Figure 2-27 shows the simplified structure of the tool, which is composed of three main components; the windows based configuration process, the command-line based integrated scheduler, and the graphical viewer. The windows based configuration-process-module is a top-level user interaction environment. It provides a graphical user interface with which the user can

configure the target system, input application information and scheduling options. In order to track the evolution of the system, it also provides project and history management.

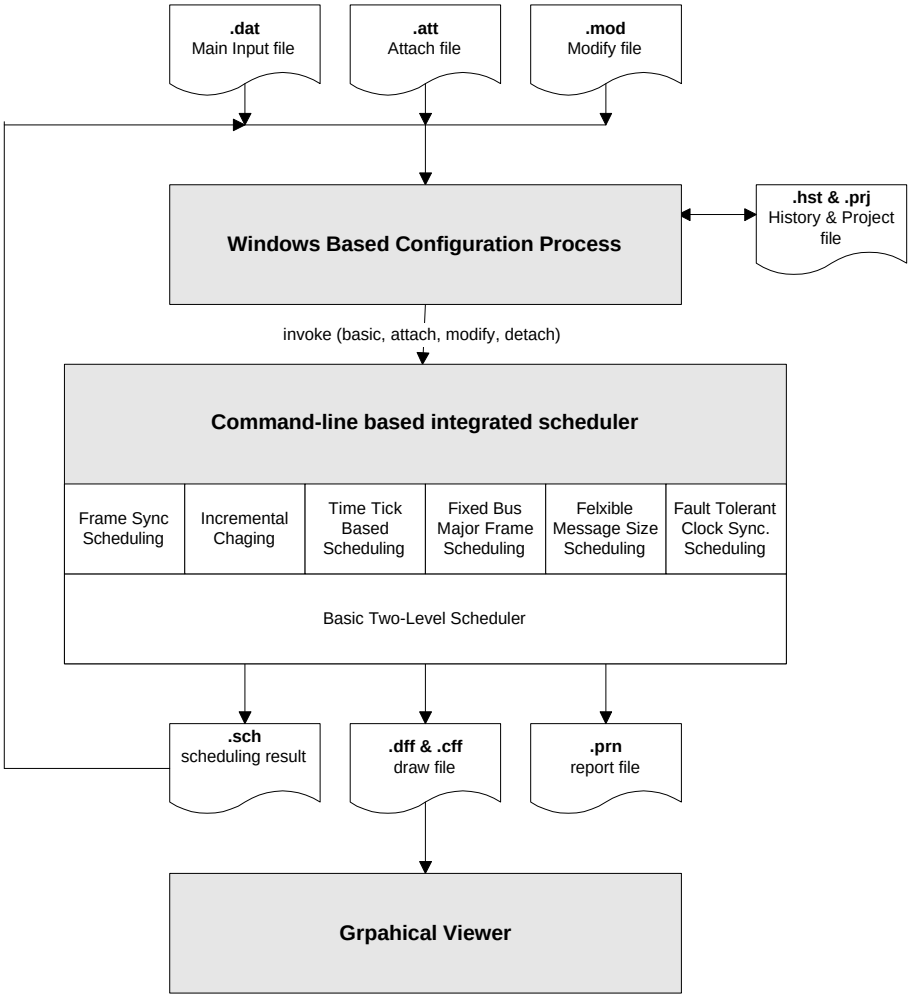


Figure 2-27 The Structure of the Scheduling Tool

Figure 2-28 shows the snapshot of the system configuration dialog box with the top-level screen as a background. The command-line based integrated scheduler is responsible for executing our scheduling algorithms with the given configuration and inputs either at the command line or in the integrated environment. A detailed result of scheduling is stored in a *.prn file. Additionally, the user can view the scheduled timing diagram of processor and bus allocation with the graphical viewer.

The scheduling algorithms for solving six additional practical constraints are implemented as plug-in modules of the basic two-level scheduling algorithm. So any combination of six constraints can be easily applied to the application.

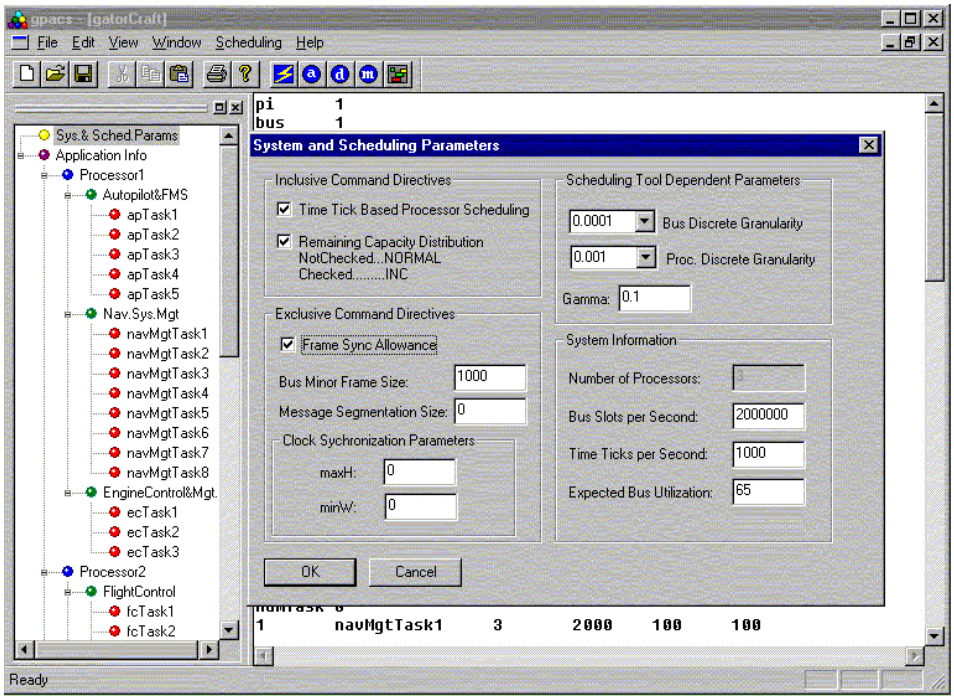


Figure 2-28 System and Scheduling Parameters in the Tool Interface

2.7 Conclusion

In this chapter, we presented scheduling algorithms to produce cyclic partition and channel schedules for the two-level hierarchical scheduling model of strongly partitioned integrated real-time systems. The system model supports spatial and temporal partitioning in all shared resources. Thus, applications can be easily integrated and maintained.

The main idea of our approach is to allocate a proper amount of capacity and to follow a distance constraint on partition and channel invocations. Thus, the tasks (messages) within a partition (channel) can have an inactive period longer than the blocking time of the partition (channel). We also use a heuristic deadline decomposition technique to find feasible deadlines for both tasks and messages. To reduce the bus bandwidth requirement for message transmission, we

develop a heuristic channel-combining algorithm, which leads to highly utilized channels by multiplexing messages of different deadlines and periods. The simulation analysis shows promising results in terms of schedulability and system characteristics.

We defined and solved an additional six practical constraints in scheduling Integrated Modular Avionics systems. We also developed a GUI-based integrated scheduling tool suite, which automates all scheduling analysis presented in the dissertation.

CHAPTER 3
SOFT AND HARD APERIODIC TASK SCHEDULING

3.1 Introduction

The integrated real-time systems model supports real-time applications of various types, including non- and less-critical applications. These kinds of applications usually adopt soft and hard aperiodic tasks. In this chapter, we focus on the design of dynamic scheduling algorithms for soft and hard aperiodic tasks in integrated real-time systems. We assume that there is an aperiodic task server in each partition to execute aperiodic tasks that belong to the same partition. To ensure the schedulability of the periodic tasks, the server would not consume any processing capacity allocated to the periodic tasks in the partition. Nevertheless, there exists available processing capacity, such as unused or unallocated capacity at the partition level, that can be used by the aperiodic task server. Also there exists pre-allocated processing capacity for aperiodic tasks, which can be shared by partitions. Our objective then is to utilize the available capacity to maximize the performance of aperiodic tasks, while meeting the distance and capacity constraints of each partition.

To maintain the distance constraints while scheduling aperiodic tasks, we propose the Distance Constraint guaranteed Dynamic Cyclic (DC^2) scheduler, which uses three basic operations; *left-sliding, right-putting,* and *compacting*. The goal of these operations is to obtain a maximum amount of slack time upon the arrival of an aperiodic task. The first operation, *left-sliding,* is to save any slack time by scheduling other partitions earlier when there is no pending task in the current partition. The second operation, *right-putting,* steals slack time as much as possible by swapping the existing cyclic schedule for partitions, while guaranteeing the distance constraints. Since the *right-putting* operation may divide the cyclic schedule into small segments,

the third operation, *compacting*, is designed to merge together the segments allocated to the same partition.

We also propose a hard aperiodic task scheduling approach. When a hard aperiodic task arrives, the scheduler must invoke an acceptance test algorithm. Basically, the algorithm checks the available slack time between the current time f and the task deadline $f+D_i$. An acceptance can only be made if the task's timing requirement can be met, and none of the hard aperiodic tasks admitted previously may miss their deadlines. With the help of the dynamic behavior of the DC^2 scheduler, we can take advantage of *future right-putting* operations in addition to the slack times of the current static schedule.

The chapter is organized as follows. Section 3.2 presents an aperiodic task scheduling model in integrated real-time systems. Section 3.3 presents the algorithms for soft aperiodic scheduling. Section 3.4 describes the algorithm for hard aperiodic scheduling. The evaluation of the algorithms by simulation studies is presented in Section 3.5. The conclusion then follows in Section 3.6.

3.2 Aperiodic Task Scheduling Model

Let's assume that there are an infinite number of aperiodic tasks, $\{J_i \mid i=0,1,2,\ldots\}$. Each aperiodic task has associated it with a worst case execution time of C_i. A hard aperiodic task additionally has a deadline D_i. To maximize the performance for aperiodic tasks, we are concerned with any available execution capacity that we can extract from the low-level cyclic schedule of a SPIRIT system. As long as the distance and capacity constraints are satisfied for each partition, this available execution capacity can then be assigned to any arriving aperiodic tasks.

We introduce *Multi-Periods Aperiodic Servers* (MPAS) that are virtual execution servers for aperiodic tasks. When it is activated, an aperiodic task server at one of the partitions can use the capacity to execute existing aperiodic tasks. A MPAS can be constructed using the following scheme. Let a finally scheduled set of partition servers be

$P = \{P_1(a_1^h, h_1), P_2(a_2^h, h_2), \cdots, P_n(a_n^h, h_n)\}$, the set being sorted in the non-decreasing order of h_k. By removing the duplicate periods in the set of h_i, $1 \le i \le n$, and sorting the remaining set in increasing order, we can obtain the non-duplicative set MPS = $\{H_1, H_2, \ldots, H_m\}$ where $m \le n$ and $H_i \mid H_j$ for $i < j$. We then define a set of $MPAS_{H_i}$ servers, where $1 \le i \le m$, that share the remaining idle processor capacity, $(1 - \sum_{i=1}^{n} a_i^h)$, evenly, i.e., we assign processor capacity $cp_{H_i} = (1 - \sum_{j=1}^{n} a_j^h)/m$, to each $MPAS_{H_i}$.

Then, the total set of servers in the low-level cyclic schedule, including partition servers *P* and multi-period aperiodic servers *MPAS*, can be formulated as follows:

$$CS = P \cup MPAS = CS_1[MPAS_{H_1}(cp_{H_1}, H_1), P_1(a_1^h, h_1), \cdots, P_{E_1}(a_{E_1}^h, h_{E_1})]$$

$$CS_2[MPAS_{H_2}(cp_{H_2}, H_2), P_{E_1+1}(a_{E_1+1}^h, h_{E_1+1}), \cdots, P_{E_2}(a_{E_2}^h, h_{E_2})] \cup \cdots \cup$$

$$CS_m[MPAS_{H_m}(cp_{H_m}, H_m), P_{E_{m-1}+1}(a_{E_{m-1}+1}^h, h_{E_{m-1}+1}), \cdots, P_{E_m}(a_{E_m}^h, h_{E_m})]$$

where all server partition cycles in the subset CS_i are the same and E_k is the last index of the partition server in the set CS_k, $E_1 < E_2 < \ldots < E_m = n$.

For instance, if we have input partition servers of $P = \{P_1(1/7,7), P_2(1/7,14), P_3(1/7,14), P_4(1/7,28)\}$, we can obtain a set of *MPAS* as $\{MPAS_7(1/7,7), MPAS_{14}(1/7,14), MPAS_{28}(1/7,28)\}$. Finally, we can obtain a cyclic schedule set CS of $\{\{MPAS_7(1/7, 7), P_1(1/7, 7)\}, \{MPAS_{14}(1/7, 14), P_2(1/7, 14), P_3(1/7, 14)\}, \{MPAS_{28}(1/7, 28), P_4(1/7, 28)\}\}$. Using the distance-constrained cyclic scheduling algorithm, we can obtain the feasible cyclic schedule shown in Figure 3-1. Since it allocates the *MPAS* with the minimum period first, for example, $MPAS_7(1/7, 7)$, it is guaranteed that every partition server does not cross a minor frame boundary. This characteristic can lead to a flexible relocation of cyclic schedule entries. Note that, in the Figure, we use *AS* as an abbreviation of *MPAS*. Given the cyclic schedule of *CS*, the straight-forward approach of

scheduling aperiodic tasks is to give CPU control to aperiodic tasks when the current timeline belongs to *MPAS*. However, for a better response time for soft aperiodic tasks and a higher acceptance rate for hard aperiodic tasks, we should look for ways to adjust the table entries dynamically.

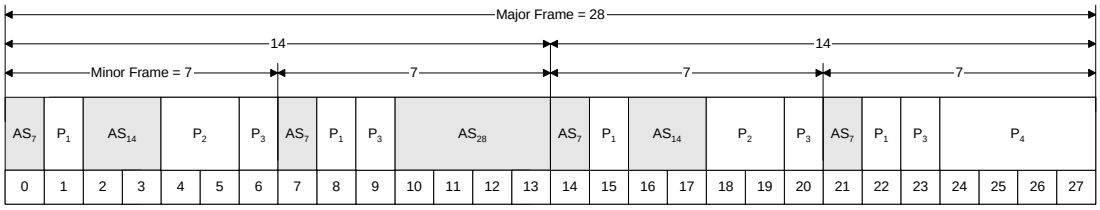


Figure 3-1 Example of Feasible Cyclic Schedule

3.3 Soft Aperiodic Task Scheduling

To dynamically schedule aperiodic tasks in a distance-constrained cyclic schedule, we propose to use three basic operations to manipulate the table entries. These are *LS* (Left Sliding), *RP* (Right Putting) and *Compacting* operations. The operations are applied to feasible cyclic schedules in run time when an aperiodic task arrives or a server finishes earlier than scheduled. In this section, we specify the details of each operation, and show that the deadlines of periodic tasks in each partition will not be missed when the operations are applied. Using these basic operations, we can then compose a distance-constrained dynamic cyclic (DC^2) scheduling algorithm.

3.3.1 Left Sliding (LS)

The *LS(x)* operation slides the current phase of the cyclic schedule left by *x* units of time in $O(1)$ time. There are two situations at which *LS(x)* will be applied. One is when the current partition server or MPAS has finished *x* units of time earlier than the allocated capacity. The other case is when the scheduler invokes a partition server or a *MPAS* with an allocated capacity of *x*, but finds no pending periodic task or aperiodic task, respectively. Figure 3-2 shows an example of a *LS(1)* operation.

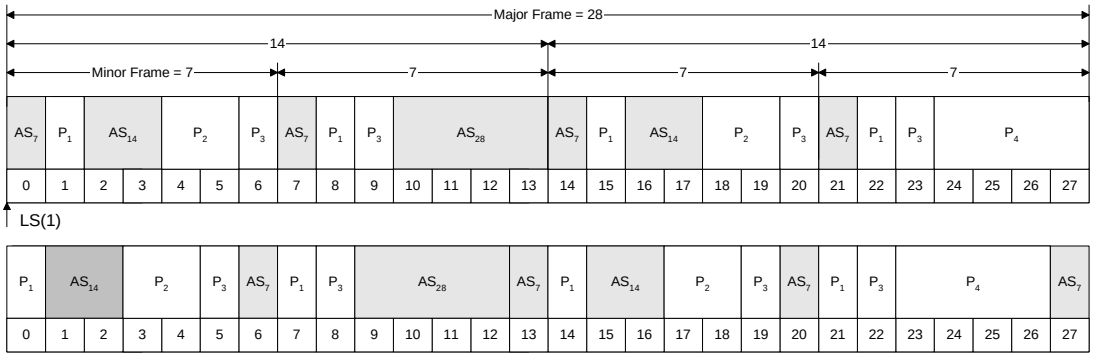


Figure 3-2 Modified Schedule after LS

To reduce the average response time of aperiodic tasks, we can use *LS* to eliminate idle processing during assigned slots. The advantage, compared to a simple polling aperiodic server algorithm, comes from its dynamic behavior. In a polling aperiodic server algorithm [12], when an aperiodic task arrives, the task has to wait until the next *MPAS* interval. Using *LS*, both partition server and *MPAS* can begin their execution earlier than the instances defined statically in the cyclic table, a better average response time should be expected. In addition, the following theorem shows the distance and capacity constraints for periodic tasks in all partitions are still valid under the *LS* operations.

Theorem 2. The modified cyclic schedule after the application of *LS(x)* is feasible, i.e., it guarantees the hard deadlines of periodic tasks in all partition servers.

Proof: We assume that a *LS(x) operation* takes place at instance f. Thus, we need to show that every periodic task that has already arrived before f, or will arrive after f, can meet its deadline. Firstly, we consider the tasks that arrive after instance f. If the *LS* operation doesn't occur, then there will be a processor capacity of a_i^h allocated to P_i in every period $[f+d, f+d+h_i]$, where $d \geq 0$. After the execution of *LS(x)*, the processing capacity allocated during $[f+d, f+d+h_i]$ under the modified schedule is equivalent to that allocated during $[f+d+x, f+d+x+h_i]$ in the original schedule. As a consequence, there will be exactly the required processor capacity for all partition servers in any periods after instance f. New periodic tasks will be able to meet

their deadlines. Secondly, we need to show that the periodic tasks that have already been released before instance f shall meet their deadlines. For the tasks that do not belong to the suspended partition, they will be given their preserved processor capacity earlier than the instance originally scheduled so that they can meet their deadlines. For the tasks that belong to the current partition, they have already finished before instance f because this partition becomes idle at f in spite of its unused remaining capacity. Therefore, we can conclude that the modified cyclic schedule after *LS(x)* is feasible. ■

3.3.2 Right Putting (RP)

The Right Putting (*RP*) operation exchanges the remaining allocation of the current partition server with a future MPAS of the same period. In order to meet the deadlines of periodic tasks and to satisfy the distance constraint in the cyclic schedule after *RP*, a *RP* operation is done based on two parameters: $PIST_i$ (Partition Idle Starting Time) and PL_i (Putting Limit) for partition server P_i. $PIST_i$ is the instance at which the partition becomes idle due to either no pending task or an early completion of tasks. PL_i is defined as the sum of $PIST_i$ and the invocation period h_i of P_i. Initially, $PIST_i$ is set to zero. According to the execution of tasks, $PIST_i$ will be reset to the instances when P_i is invoked and finds no waiting periodic tasks. Obviously, $PIST_i$ is always less than or equal to the current instance.

An *RP* operation is initiated when either an aperiodic task arrives or there is still a pending aperiodic task after the completion of a partition server. To reschedule the remaining capacity of the current partition server at a later interval, the operation brings in an equivalent processing capacity from a *MPAS* of the same period that lies between the current time instance and PL_i. If no such *MPAS* exists, the operation would not take place at all. Note that this capacity exchange leads to a revision of table entries in the cyclic schedule. This implies that the exchange becomes persistent for all future invocations of the partition server and the *MPAS*. Since the RP

operation should track all time interval entries in the cyclic schedule, its time complexity is *O*(N), where N is the total number of time table entries.

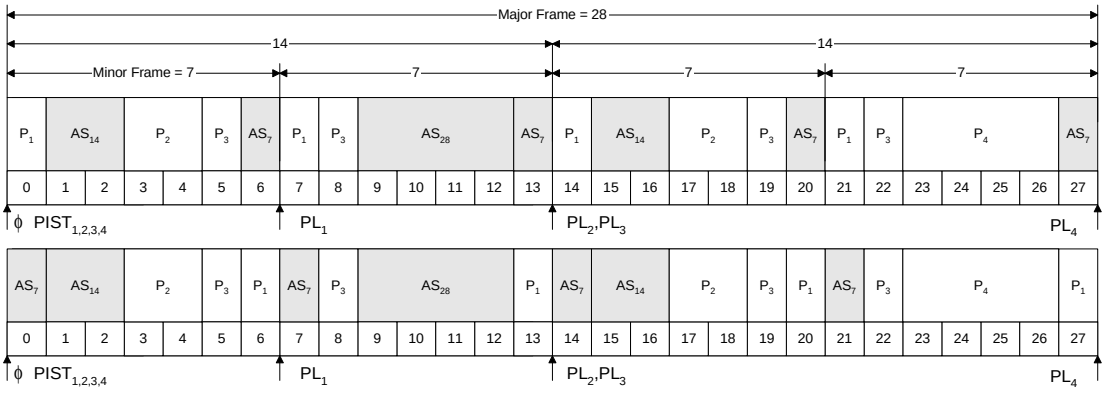


Figure 3-3 Modified Schedule after RP

In Figure 3-3, we show an example of an *RP* operation, which assumes that an aperiodic task arrives at time 0. It exchanges the current partition server P_1 with $MPAS_7$ that is scheduled between $[0, PL_1]$. This *RP* operation results in the immediate execution of the aperiodic task without any violation of distance constraints. However, the *RP* operation may split a scheduled interval into segments. This will increase the number of allocation entries in the cyclic schedule table, and the number of context switches between partitions. To reduce the problem of segmentation, a compaction algorithm can be applied.

Theorem 3. The modified cyclic schedule after the application of *RP* operations is feasible, i.e., it guarantees to meet the hard deadlines of periodic tasks in all partition servers.

Proof: We assume that an *RP* event occurred at instance f and the invocation of the current partition server P_i is postponed. Firstly, we consider a periodic task that arrives after instance f. If the task does not belong to P_i, it can always meet the deadline. This is because the schedules for all partition servers except P_i are not changed after the *RP*. For P_i there was exactly a_i capacity in every period of h_i before the *RP* operation. Assume that the *RP* exchanges y units of time for P_i from $[f, f+y]$ to $[f+d, f+d+y]$, where d is a real number. Due to $f+d+y \leq PL_i$ =

$PIST_i + h_i \leq f + h_i$, there exists a capacity a_i^h allocated to P_i in $[f, f+h_i]$. We can do the same exchange in all iterative periods after f given that the corresponding partition server and MPAS are in the same relative position in the cyclic schedule. Therefore, for every period h_i of $[f,¥]$, i.e. $[f+d, f+d+h_i]$, where $d \geq 0$, a capacity of a_i^h exists for P_i. Secondly, we consider a periodic task that had arrived before instance f. If the task belongs to P_j, where $j<i$, it will meet the deadline because the schedule for P_j is still unchanged after the *RP* operation. For a task that belongs to P_i, we have to show that there should be capacity a_i^h in any period $[PIST_i+e, PIST_i+e+h_i]$ where $0 < e < f-PIST_i$. We don't need to consider any periodic task that had arrived before $PIST_i$. Note that there was exactly a_i^h processor capacity in the period of $[PIST_i+e, PIST_i+e+h_i]$ before the operation, where $0 < \varepsilon < \phi$-$PIST_i$. After an exchange of the capacity y, which is less than a_i^h, in the period $[f, PL_i]$, there still exists the same amount of capacity a_i^h in the concerned interval $[PIST_i+e, PIST_i+e+h_i]$ since $PL_i \leq PIST_i+e+h_i$ and $[f,PL_i] \subseteq [PIST_i+e, PIST_i+e+h_i]$ for any $0 < e < f-PIST_i$. Therefore, the partition server P_i will be allocated a requested processor capacity in every invocation period. ■

3.3.3 Compacting

To reduce the problem of segmentation due to the *RP* operations, we propose a *Compacting* operation. The operation delays a MAPS entry and exchanges it with the entry of a future partition server. This contrasts with the *RP* operation, which first postpones a partition server and then brings and starts a MAPS immediately. A heuristic selection is performed in a *Compacting* operation such that the neighboring entries that belong to the same server can then be merged in the scheduling table. Since the *compacting* operation needs multiple *RP* operations, the complexity of *Compacting* is also $O(N^2)$. The operation can be applied periodically. For instance, this can be done every few major frames, or at the time of completion of a certain number of *RP* operations, or in the event that the number of context switches exceed a certain threshold.

Theorem 4. The modified cyclic schedule after the application of the *Compacting* operation is feasible, i.e., it guarantees to meet the hard deadlines of periodic tasks in all partition servers.

Proof: The proof is trivial. Since the *Compacting* operation uses a similar approach as the *RP* operation for exchanging scheduling entries, the proof is the same as for the *RP* operation. It also can preserve the distance constraints guaranteed in the original cyclic schedule, after the adjacent scheduling entries that belong to the same server in the same minor frame are merged. ■

3.3.4 DC^2 Scheduler

Based on the three operations *LS*, *RP*, and *Compacting*, we propose a DC^2 scheduler that accommodates aperiodic tasks within the distance-constrained cyclic scheduling of partition servers. When there is no pending aperiodic task, it behaves like a cyclic scheduler, except for the intervention of *LS* operations. Whenever a server becomes idle without consuming all its allocated capacity, the *LS* operation is invoked. Hoping to use saved slack times, the scheduler can apply a *RP* operation when an aperiodic task arrives. The scheduler invokes the *Compacting* operation when the number of context switches exceeds a certain threshold at the beginning of a major frame. Following Theorems 2, 3, and 4, we can conclude that the DC^2 scheduler guarantees the deadlines of hard periodic tasks in all partition servers while serving aperiodic tasks. In Figure 3-4, we present the abstract algorithm of the DC^2 scheduler.

The resulting scheduler may lead to an immediate execution of an arriving aperiodic task, and does not need to wait for the beginning of the next aperiodic server interval. Practically, the DC^2 scheduler should be carefully designed so that the scheduling overhead is minimized. For example, we can limit the number of burst arrivals of aperiodic tasks for a short interval time. The scheduling policy of MPAS will be discussed after discussion of hard aperiodic task scheduling.

```
loop {
   receive a Timer interrupt;
   S = getNextServer;     // S will be either a partition server or MPAS
   while (S doesn't have any pending task) {
         LS(|S|);
         S = getNextServer;
   }
   set Timer to current time + |S|;          // |S| is allocated capacity
   dispatch S;
   sleep;
}
signals:
   case (S finishes earlier by x units of time) do LS(x);
   case (aperiodic task arrives) do RP and dispatch MPAS if available;
```

Figure 3-4 DC^2 Scheduing Algorithm

3.4 Hard Aperiodic Task Scheduling

When a hard aperiodic task arrives, the scheduler must invoke an acceptance test algorithm. Basically, the algorithm checks the available slack time between the current time f and the task deadline $f+D_i$. An acceptance can only be made if the available slack time is bigger than the task's WCET. Thus, the task's timing requirement can be met and none of the hard aperiodic tasks admitted previously may miss their deadlines. With the help of the dynamic behavior of the DC^2 scheduler, we can take advantage of *future RP* operations in addition to the slack times of the current static schedule.

3.4.1 Low-Bound Slack Time

With the current snapshot of the cyclic schedule as of current time f, the scheduler can obtain a *low-bound slack time* between f and $f+D_i$ that exists regardless of any future *LS* and *RP* operations. Note that the further slack time can be added in due to the results of *LS* and *RP* operations before $f+D_i$.

To aid the computation of *low-bound slack time*, we use three parameters; *sf*, *sl* and *sr*. sf_i equals the sum of processing capacities allocated to all MAPS's in a minor frame *i*, and represents the available slack times in minor frame *i*. sl_k and sr_k store the available slack time before and after the schedule entry *k* in a minor frame. In Figure 3-5, we show an example that illustrates the values of these parameters.

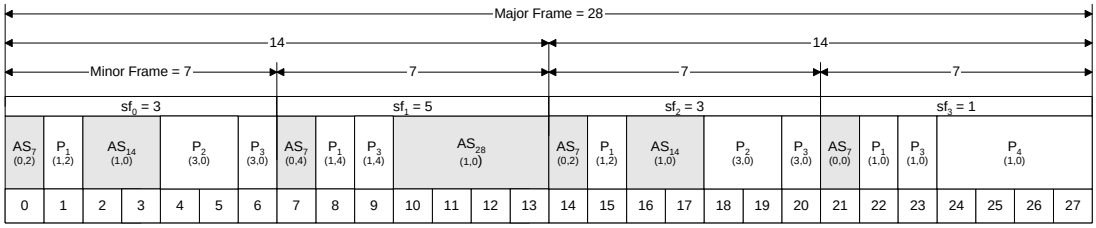


Figure 3-5 Example of the sf, sl, and sr Parameters

The value pair of *sl* and *sr* is shown in each scheduled interval. The scheduled interval for partition P_1 in minor frame 0 has a left slack time of 1 and a right slack time of 2 in the example. We maintain the *frame-to-frame slack time table* of accumulated slack time from one frame to the other in the cyclic schedule. For instance, the table corresponding to the schedule of Figure 3-5 is shown in Table 3-1. The accumulated slack time from frame 0 to frame 2 is computed as $sf_0+sf_1+sf_2=11$.

Table 3-1 Frame-to-Frame Slack Time Table

	f0	f1	f2	f3
f0	3	8	11	12
f1	12	5	8	9
f2	7	12	3	4
f3	4	9	12	1

Using this table and relevant parameters, we can efficiently calculate the *low-bound slack time* in interval $[f, f+D_i]$. For example, we assume that a hard aperiodic task that has a relative

deadline 18 arrives at time 5. By summing up 0[*sr* of P_2 at 5], 8[the accumulated slack time from frame f_1 to frame f_2] and 1[*sl* of P_3 at 23], we can obtain a *low-bound slack time* of 9 for the arriving task.

To guarantee the schedulability of an arriving hard aperiodic task, we only consider the *low-bound slack time* and *future RP* operations even if we know the slack time can be increased due to future *LS* operations. We cannot take advantage of future *LS* operations because they depend on the future usage of partition servers. However, at the arrival instance, we can exactly calculate the extra slack time obtainable from *future RP* operations which exchange the capacity of partition server P_j in $[f, f+D_i]$ with that of MPASs in $[f+D_i, PL_j]$, for every partition server P_j, where $f+D_i < PL_j$. The expected *future RP* operations must occur while there is at least one aperiodic task pending. The contribution from the *future RP* operations can be significant in slack time calculation, as it will be evaluated in later simulation studies.

3.4.2 Acceptance Test Considering Future RP Operations

In an acceptance test, the algorithm must guarantee that the existing hard aperiodic tasks, which have already been admitted, should meet their own deadlines. We assume that MPAS schedules hard aperiodic tasks with an EDF scheduling policy. In the decision process, we assign two parameters, remaining execution time (*rc*) and individual slack time(*s*), for each existing hard aperiodic task. Using these two parameters and the requirements of the arriving task, the *acceptance test* algorithm, as shown in Figure 3-6, first checks whether total slack time, *totalSlack*, as of the current time is sufficient to accommodate the existing hard aperiodic tasks and the newly arrived one, or not. If the total slack time is not sufficient, it rejects the new one. It then checks the schedulability of pre-admitted hard aperiodic tasks. For the hard aperiodic task whose deadline is earlier than the new one, its schedulability will not be affected by the admission of the new task due to the EDF scheduling policy. However, we should consider the

ones whose deadlines are later than the deadline of the arriving task, and check the total slack time, s_k, for each of them.

---

obtain totalSlack[ϕ, $\phi+D_i$]= low-bound slack time from ϕ to $\phi+D_i$ plus expected slack times from future RP operations;
if ((s_i = *totalSlack*$[f, f+D_i] - C_i - \sum_{k=1}^{m} rc_k$)< 0) **return** *reject*;
for all hard aperiodic tasks *k* whose deadline is later than $f+D_i$
if (($s_k = s_k - C_i$) < 0) **return** reject;
return *admit*;

Figure 3-6 Acceptance Test Algorithm

In order to prove that our *LS* and *RP* operations are also harmless to the admitted hard aperiodic tasks, we now show that the *currently calculated slack time* in the interval $[f, f+D_i]$ cannot be reduced by future *LS* and *RP* operations.

Lemma 1. Future *LS* and *RP* operations will not jeopardize the hard aperiodic tasks that had already been admitted before current time f.

Proof: Firstly, considering a future *LS* operation, as long as there is an existing hard aperiodic task, the *LS* operation cannot be applied to any MPAS. If the *LS* operation is applied to a partition server, the slack time will just be increased. Secondly, considering a future *RP* operation, there are only two cases. In the first case, if the *RP* operation exchanges the capacity of a partition server with the capacity of a MPAS that lies in $[f, f+D_i]$, it does not change the slack time in the interval $[f, f+D_i]$. In the second case, if the *RP* operation exchanges the capacity of a partition server with the capacity of a MPAS that lies beyond $f+D_i$, it increases the slack time in the interval $[f, f+D_i]$. Therefore, future *LS* and *RP* operations will not reduce the *slack time*. ■

3.4.3 Dynamic Slack Time Management

Since we apply three dynamic operations, *LS*, *RP*, and *Compacting*, to the distance-constrained cyclic schedule at run time, the slack times available at a certain instance may be

changed. To keep the slack time related parameters, i.e. *sf*, *sr*, *sl*, and the *frame-to-frame slack time table*, correct in an efficient way, an update must be done whenever the dynamic operations are applied. At initialization time, the scheduler calculates these parameters and generates the *frame-to-frame slack time table* based on the initial cyclic schedule. After completion of an *LS* operation, the scheduler does not need to change anything. This is because that *LS* operation does not affect the sequence or allocated capacity of the cyclic schedule. After completion of an *RP* operation, the scheduler must adjust the parameters and the table for affected slack times. In Figure 3-7, we depict six different cases of an *RP* operation and their corresponding slack time adjustment.

Figure 3-7 (a1, b1, and c1) shows the slack time adjustments when RP operations are performed within the same minor frame. Three cases (a1), (b1), and (c1) show three different situations, i.e., when the capacity of a partition server is equal to, greater than, or less than the capacity of a corresponding MPAS respectively. In (a2), (b2), and (c2), the only difference is that the RP operation brings a MPAS interval from a different minor frame in a late timeline. Since no partition or MPAS server crosses a minor frame boundary, there is no other case. In each case of the figure, we show the original cyclic schedule in the top diagram and the result after completion of an RP operation in the bottom. In the bottom diagram of each case, the shaded entry represents the scheduled interval whose parameter, either *sl* or *sr*, should be modified. For example, in the case of (a1), the swapped MPAS and the partition server P_i will have (*sl*, *sr*-x) and (*sl*'+x, *sr*') respectively as a new parameter pair. All other entries between them will have an x-time-unit increased *sl* and an x-time-unit decreased *sr*. In addition to modifications of *sl* and *sr* values, *sf* value should be modified properly in the cases; (a2), (b2), and (c2). This is because the exchange of capacity between two minor frames will influence the slack time in each minor frame. As an embedded function of an *RP* operation, this slack time adjustment algorithm does not influence the time complexity of *RP* operation, *O*(N).

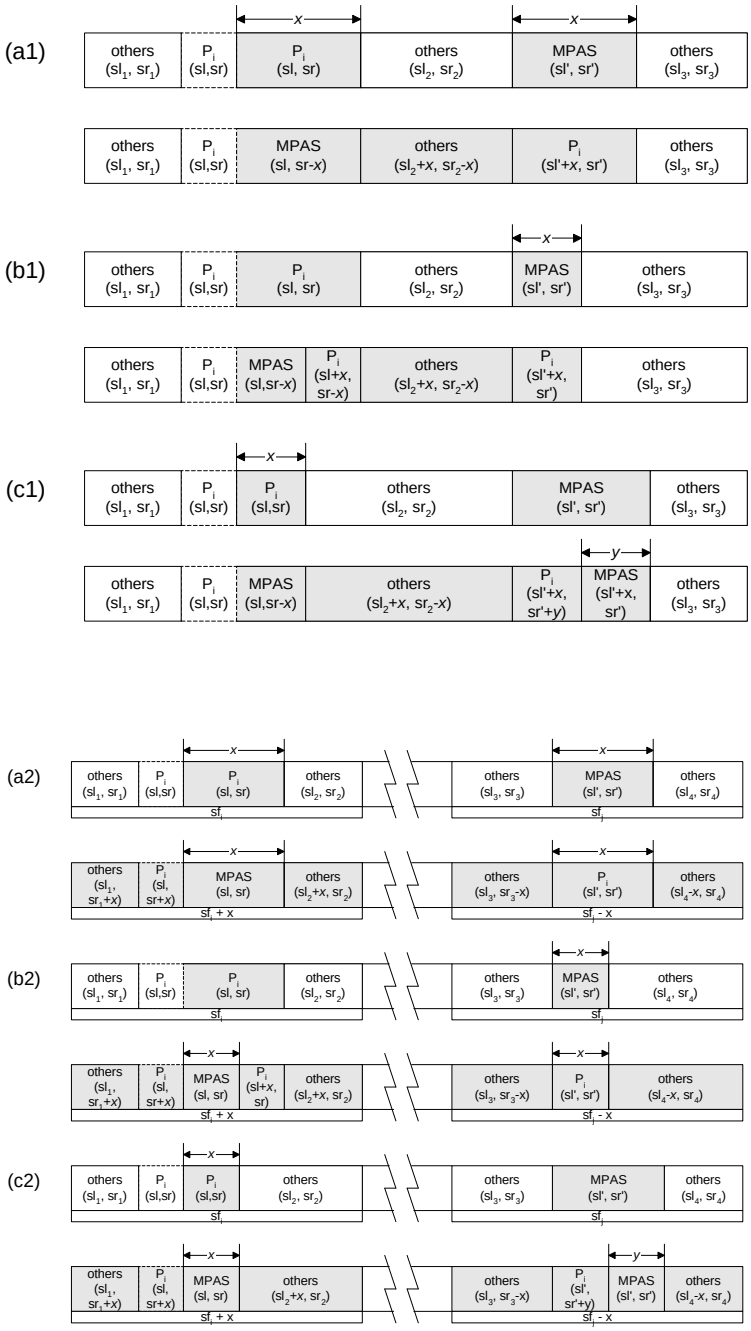


Figure 3-7 Six Possible Slack Time Adjustments after an RP Operation

3.4.4 Multi-Periods Aperiodic Server Scheduler

In scheduling partitions that consist of only periodic tasks, it is not necessary for the lower-level cyclic scheduler to keep track of execution activities of local tasks within a partition. This is possible due to the off-line scheduling analysis of partitions that have static requirements.

A sufficient static processor allocation, which is characterized by a pair of partition capacity and cycle, guarantees the real-time constraints of all periodic tasks of partitions. But this cannot be applied to our DC^2 scheduling approach, because the aperiodic task scheduler utilizes the unused capacity of other partitions to maximize the performance. So the minimal amount of aperiodic task information is also dynamically kept in the lower-level cyclic scheduler, especially via the MPAS scheduler, as shown in Figure 3-8.

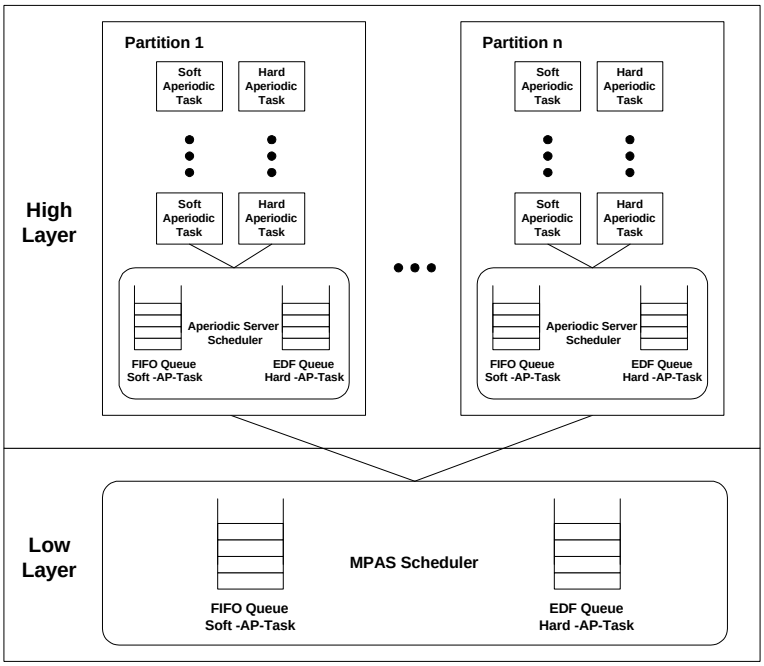


Figure 3-8 Two-Level Hierarchy of Aperiodic Task Schedulers

In the higher layer, an aperiodic server scheduler takes over processor control when the current time instance belongs to the MPAS server in the cyclic schedule and the MPAS server gives its capacity to the corresponding partition. Then the aperiodic server schedules either soft or hard aperiodic tasks based on the information given by the MPAS server.

In the lower layer, a MPAS server maintains a FIFO queue for partitions holding ready soft aperiodic tasks, and an EDF queue for partitions holding ready hard aperiodic tasks. In both queues, we do not store task-level information but corresponding partition-level information. The EDF queue also has a higher priority than the FIFO queue. The MPAS scheduler dispatches a

partition's aperiodic server according to the described policy, and then each dispatched aperiodic server schedules its own aperiodic tasks.

3.5 Simulation Studies of the DC^2 Scheduler

Simulation studies have been conducted to evaluate the DC^2 scheduler against the simple *polling server* and the *LS-only scheduler*. In a *polling server* method, aperiodic tasks are served only in fixed time intervals allocated to the *MPAS*. In order to find the scheduling characteristics of the DC^2 scheduler, we also introduce and make a comparison with the *LS-only scheduler* that uses the *LS* operation only. In the simulation work, we measured average response time of soft aperiodic tasks and average acceptance rate of hard aperiodic tasks. The aperiodic workload was modeled with the *Poisson* inter-arrival time and exponential distribution of execution time. We choose the hard deadline of a hard aperiodic task based on the inter-arrival time of the task, e.g. 0.5 or 1.0 times the inter-arrival time. For each combination of inter-arrival time and execution time, we randomly generated 25 different simulation sets. For each random set, we generated 5 partition servers for periodic tasks and simulated it for 2,000,000 units of time. For every random set, we randomly choose a minor frame size from 56 to 200, and a major frame size of 16 times the chosen minor frame size. The sum of workloads from all partition servers in the cyclic schedule is 69%.

3.5.1 Response Time

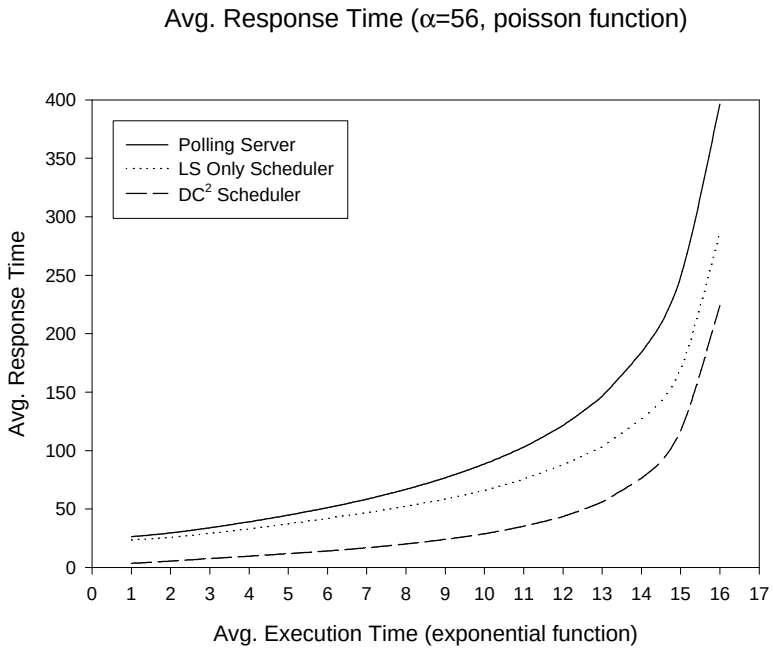
We present the measured average response time of soft aperiodic tasks in Figure 3-9. We have built two different simulation scenarios. Firstly, we fixed the *Poisson* inter-arrival time of aperiodic tasks with 56. Then, we varied the distribution of worst execution time of a task following an exponential distribution of μ between 1 (1.78% workload) and 16 (29% workload). The simulation result is shown in Figure 3-9-(a). Secondly, we fixed the distribution of worst case execution time with an exponential function of μ equal to 3. Then we varied the *Poisson* inter-arrival time of, α between 12 (25% workload) and 40 (7.5%). The result is shown in Figure 3-9-

(c). The normalized average response time of (a) and (c), based on that of the *polling server* method, is also shown in Figure 3-9-(b) and (d).

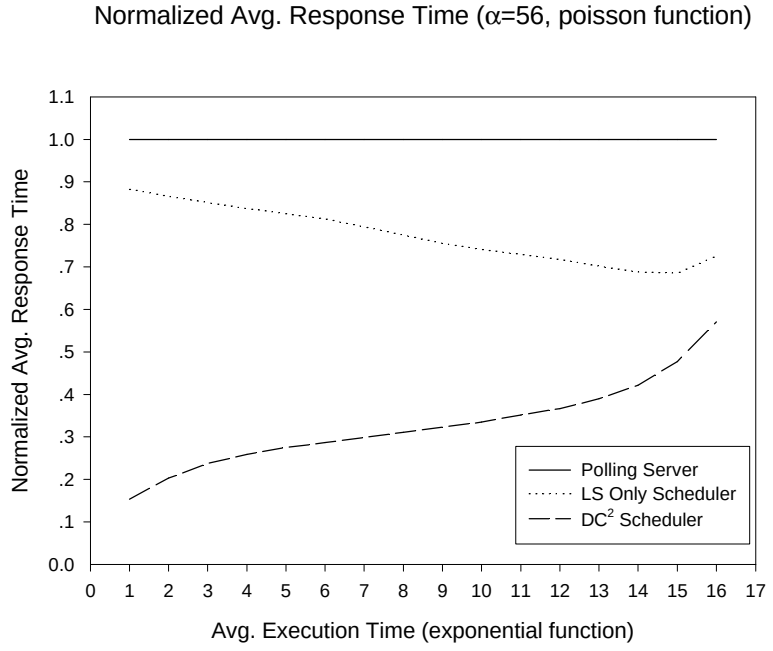
As shown in (a) and (c), the DC^2 scheduler significantly improves the average response time of soft aperiodic tasks compared to the simple *polling server* and *LS-only scheduler*. Since the DC^2 scheduler uses both *LS* and *RP* operations, it is worth investigating contributions from both operations. As we can observe in (b) and (d), an *RP* operation contributes more in a lighter aperiodic task workload (or lighter processor utilization), while *LS* contributes more in a heavier aperiodic task workload (or heavier processor utilization). This is due to the fact that when the processor utilization increases, there is less opportunity of success for the *RP* operation. However, still in this situation, the *LS* operation can flexibly and efficiently manage its MPAS server capacity. Whenever there is an idle server, the DC^2 scheduler can successfully invoke the *LS* operation.

3.5.2 Acceptance Rate

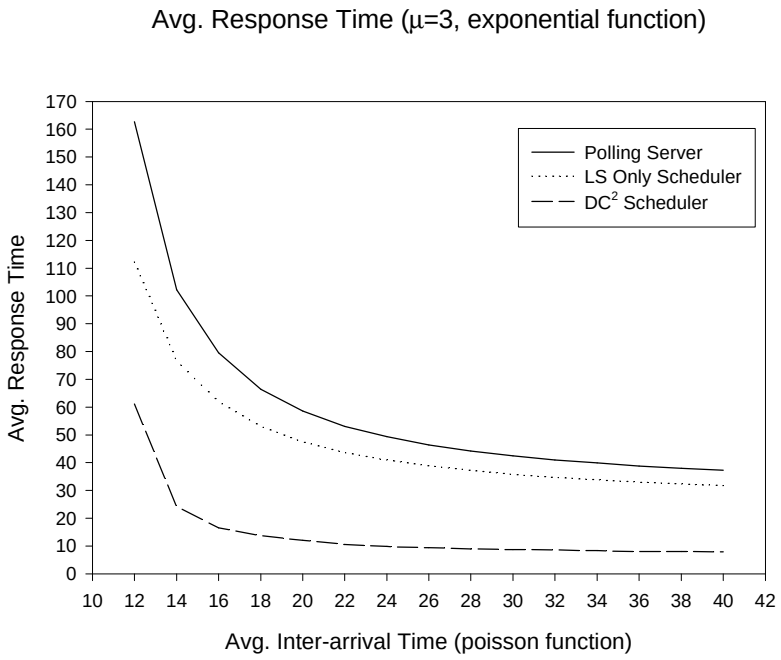
To evaluate the acceptance rate of hard aperiodic tasks, we compared the average acceptance rates of the *polling server* and the DC^2 scheduler. A simulation was performed with a sequence of hard aperiodic tasks that had a *Poisson* inter-arrival time, α equal to 50 and 200. The worst case execution time of each hard aperiodic task follows an exponential distribution. For each *Poisson* inter-arrival time of, 50 and 200, we varied workloads of hard aperiodic tasks from 10% to 60%. We also varied the deadline of each hard aperiodic task as 0.5 and 1.0 times the inter-arrival time of the task. We show the simulation result in Figure 3-10.


Avg. Response Time (α=56, poisson function)
Avg. Response Time
Avg. Execution Time (exponential function)
Polling Server
LS Only Scheduler
DC2 Scheduler
0
50
100
150
200
250
300
350
400
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

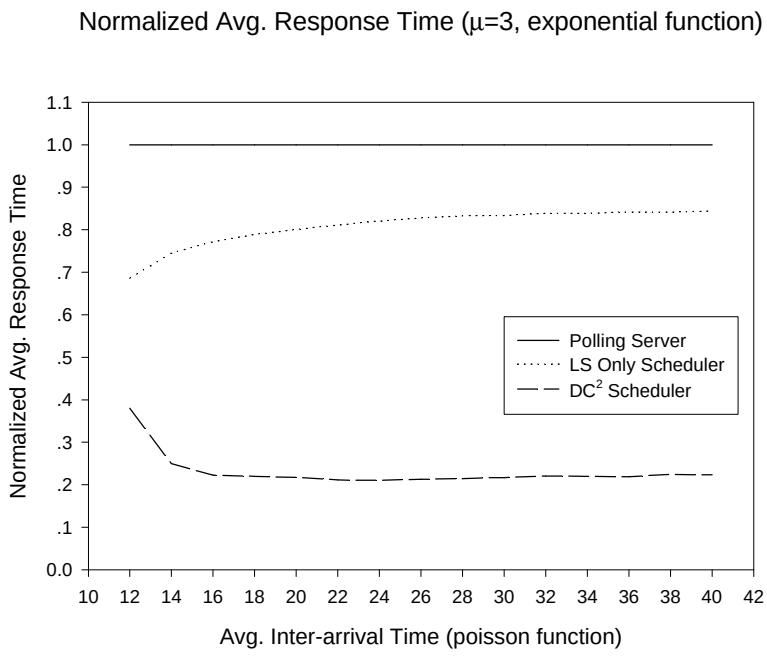
(a)


Normalized Avg. Response Time (α=56, poisson function)
Normalized Avg. Response Time
Avg. Execution Time (exponential function)
Polling Server
LS Only Scheduler
DC2 Scheduler
0.0
.1
.2
.3
.4
.5
.6
.7
.8
.9
1.0
1.1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

(b)

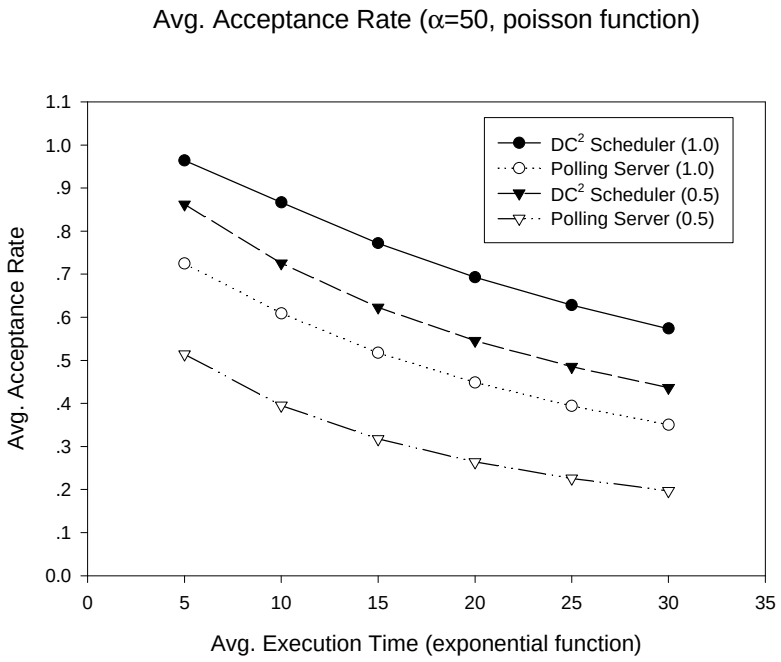


(c)



(d)

Figure 3-9 Average Response Time



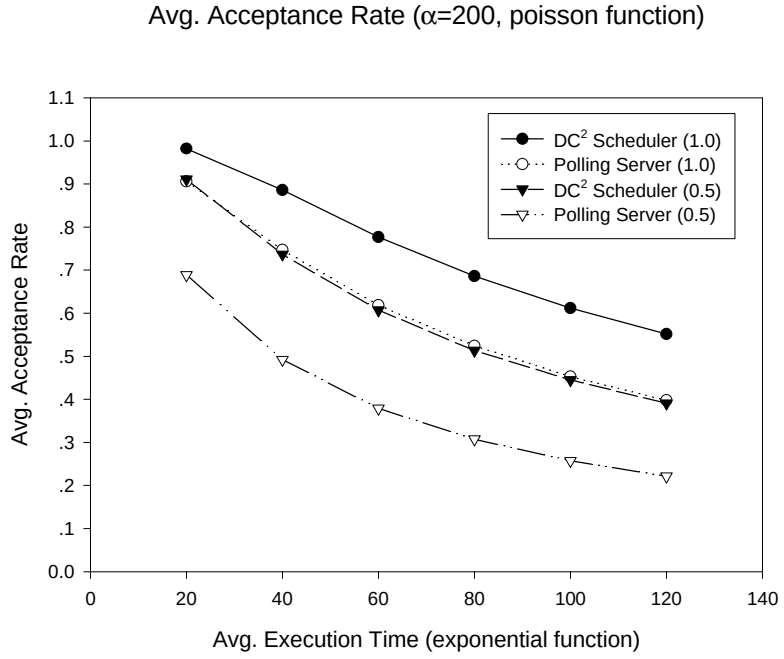


Figure 3-10 Average Acceptance Rate

As shown in the two simulation results, the DC^2 scheduler outperforms the *polling server* in acceptance rate. In the case of the *Poisson* inter-arrival time being equal to 50, the acceptance rates of the DC^2 scheduler are 29.5% and 24.6% higher than those of the polling server at the deadline factor of 0.5 and 1.0 respectively. When the *Poisson* inter-arrival time is 200, which is the maximum possible minor frame size in the simulation, the acceptance rates of the DC^2 scheduler are 21.2% and 17% higher. The smaller the inter-arrival time and the tighter the deadline, the DC^2 scheduler shows the advantage of a higher acceptance rate. This is because the benefit from future RP operations is limited by the period of each partition server. Therefore, we can expect a much higher acceptance rate advantage when hard aperiodic tasks come in bursts with tighter deadlines.

3.6 Conclusion

The proposed DC^2 scheduling algortihm guarantees the real-time constraints of periodic tasks of partitions while dynamically scheduling both soft and hard aperiodic tasks. By applying three essential operations, *Left Sliding*, *Right Putting*, and *Compacting*, in the DC^2 scheduler, we maximize the slack time to be consumed by an aperiodic task, while guaranteeing that all hard periodic tasks of each partition meet their own deadlines. We also developed hard aperiodic task scheduling and admission methods. With the help of the dynamic behavior of the DC^2 scheduler, we can take advantage of *future RP* operations in addition to the slack times of current static schedules in admission tests of hard aperiodic tasks. Through the simulation studies, we show that the DC^2 scheduler outperforms the simple *polling server* method and *LS-only scheduler* which have similar concept to other resource reclaming algorithms, both in average response time and in acceptance rate.

CHAPTER 4
REAL-TIME KERNEL FOR INTEGRATED REAL-TIME SYSTEMS

4.1 Introduction

To achieve reliability, reusability, and cost reduction, a significant trend in building large complex real-time systems is to integrate separate application modules. Such systems can be composed of real-time applications of different criticalities and even Commercial-Off-The-Shelf (COTS) software.

An essential requirement of integrated real-time systems is to guarantee strong partitioning among applications. The spatial and temporal partitioning features ensure an exclusive access of physical and temporal resources to the applications. The SPIRIT-μKernel has been designed and implemented based on a two-level hierarchical scheduling methodology such that the real-time constraints of each application can be guaranteed. It provides a minimal set of kernel functions such as address management, interrupt/exception dispatching, inter-application communication, and application scheduling.

In designing the SPIRT-μKernel, we followed the design concepts of the second-generation microkernel architecture, because it provides a good reference model to achieve both flexibility and efficiency. Compared to the first-generation micorkernel architecture, the second-generation microkernels such as MIT's Exokernel and GMD's L4 achieve better flexibility and performance by following two fundamental design concepts. Firstly, to achieve better flexibility, the microkernel is built from scratch to avoid any negative inheritance from monolithic kernels. Secondly, to achieve better performance, it implements a hardware dependent kernel, maximizing the support from the hardware.

The goals of the SPIRIT-μKernel are to provide dependable integration of real-time applications, flexibility in migrating operating system personalities from kernel to user applications, including transparent support of heterogeneous COTS RTOS on top of the kernel, high performance, and real-time feasibility. To support integration of real-time applications that have different criticalities, we have implemented a strong partitioning concept using a protected memory (resource) manager and a partition (application) scheduler. We also developed a generic RTOS Port Interface (RPI) for easy porting of heterogeneous COTS real-time operating systems on top of the kernel in user mode. A variety of operating system personalities, such as task scheduling policy, exception-handling policy and inter-task communication, can be implemented within the partition according to individual requirements of partition RTOS. To demonstrate this concept, we have ported two different application level RTOS, WindRiver's VxWorks 5.3 and Cygnus's eCos 1.2, on top of the SPIRIT-μKernel.

The scheduling policy of the SPIRIT-μKernel is based on the two-level hierarchical scheduling algorithm, which guarantees the individual timing constraints of all partitions in an integrated system. At the lower microkernel level, a distance-constrained cyclic partition scheduler arbitrates partitions according to an off-line scheduled timetable. On the other hand, at the higher COTS RTOS level, each local task scheduler of a partition schedules its own tasks based on fixed-priority driven scheduling.

We can find the application areas of the SPIRIT-μKernel in the following two cases. Firstly, it can be used in the integration of existing real-time applications that run on different COTS RTOS. Secondly, it can be used in the integration of real-time applications that require different operating system personalities.

The rest of this chapter is structured as follows. We discuss the system model and design concepts of the SPIRIT-μKernel in section 4.2. We describe the kernel architecture in section 4.3.

In section 4.4, we present generic a RTOS Port Interface. In section 4.5, we explain the prototype implementation and performance evaluation. A short conclusion then follows in section 4.6.

4.2 SPIRIT-μKernel Model and Design Concepts

In this section, we describe a software architecture for which we build the SPIRIT-μKernel, and then discuss the design concepts of the SPIRIT-μKernel.

4.2.1 Software Architecture model for Integrated Real-Time Systems

The SPIRIT is composed of multiple communicating partitions in which there are also multiple interacting tasks, as shown in Figure 4-1. The SPIRIT uses the two-level hierarchical scheduling policy in which partitions are scheduled by the SPIRIT-μKernel's cyclic partition scheduler, and the tasks of a partition are scheduled by the fixed-priority driven local task scheduler of each partition.

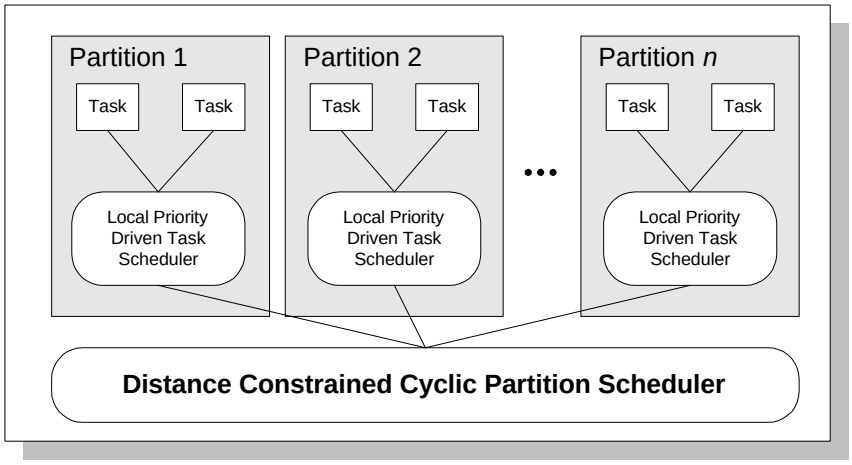


Figure 4-1 Strongly Partitioned Integrated Real-Time System Model

In order to meet the deadlines of periodic tasks within each partition, the processing capacity dedicated to a partition is then dependent on how frequently it is served in the cyclic schedule. We have investigated the constraints on the allocated capacity (a) and the invocation period (h) and have been able to construct a monotonic function between a and h for any set of periodic tasks, under either rate- or deadline-monotonic scheduling algorithms. Then, similar to pinwheel scheduling [42], we can compose a distance-constrained cyclic schedule for all

partitions in the SPIRIT system. Following this approach, for a system of n ordered partitions $P_{1, \ldots, n}$, each partition server, P_i, will be given a processor capacity of a_i in the period of h_i to process its periodic tasks, where $\sum_{i=1}^{n} a_i \leq 1$ and h_i | (divides) h_j for $i < j$. By the characteristics of distance-constrained cyclic scheduling, the independence between the lower-level partition scheduler and the higher-level local task scheduler is achieved. The details of the scheduling work can be found in Chapter 2.

4.2.2 Design Concepts of the SPIRIT-μKernel

As a key implementation vehicle of the strongly partitioned integrated real-time systems, the SPIRIT-μKernel has the following design concepts at its core:

Ensuring temporal partitioning. Enforcing temporal separation should be implemented efficiently by avoiding excessive overheads, and feasibly by meeting application's timing constraints. The key to achieving temporal partitioning is the scheduling strategy. One of the most difficult jobs is to build a feasible schedule for the SPIRIT. Since the SPIRIT is composed of multiple partitions that might have different local scheduling policies, the kernel must provide a feasible two-level hierarchical scheduling strategy. In this dissertation, we have devised a feasible two-level hierarchical scheduling algorithm that solves a pair of fixed-priority and cyclic scheduling. In the first prototype of the kernel, we adopt this pair of scheduling policies.

Ensuring spatial partitioning. Resources allocated to a partition must be protected from unauthorized access by other partitions. Also, the kernel's system resources must be protected from the partitions. In addition to protection, efficiency and deterministic applications execution are important features that should not be sacrificed for spatial partitioning. Usually, spatial protection is implemented using the memory management unit of a processor. The important factors to be concerned with in achieving efficiency and deterministic execution are a kernel-user mode switch, a partition address space switch, a Table Look-aside Buffer (TLB) overhead, etc.

Supporting applications based on heterogeneous COTS RTOS on top of the kernel. Our emphasis is also on the flexibility of accommodating COTS real-time operating systems on top of the kernel. For example, in Integrated Modular Avionics, there is a need to integrate multiple applications that are originally developed in different real-time operating systems. If the kernel and scheduling algorithms are certifiable, it can reduce integration efforts and costs by a tremendous amount.

Restricting COTS RTOS to user mode. In the SPIRIT, all codes including the COTS RTOS kernel of the partition must be run in user mode. This is a challenge because we need to develop a secure interaction method between a kernel and partitions and to modify COTS real-time operating systems in order to enforce such a policy. The problem becomes even harder when both COTS real-time operating systems and their applications run in supervisor mode for efficiency purposes, as is common in COTS RTOS.

Capturing second-generation microkernel architectural concepts. Since the SPIRIT supports different kinds of application environments in a shared computing platform, it is better to follow a microkernel architecture than a monolithic kernel. The second-generation microkernel architecture provides better flexibility and high performance.

4.3 SPIRIT-μKernel Architecture

The SPIRIT-μKernel provides a strongly partitioned operating environment to the partitions that can accommodate application specific operating system policies and specialties. A partition can have the flexibility of choosing its own operating system personalities, such as task scheduling policy, interrupt and exception handling policy, and inter-task communication, etc. The SPIRIT-μKernel provides only the minimal necessary functions, such as partition management and scheduling, memory management and protection, interrupt/exception dispatcher, timer/clock services, inter-partition communication, device driver support, and system configuration management, as shown in Figure 4-2. Following the design concepts of the second-

generation microkernel architecture, we have built the kernel from scratch and finely tuned it, utilizing the features of the underling processor architecture in order to achieve flexibility and efficiency. The current implementation of the SPIRIT-µKernel takes advantage of hardware support from the PowerPC processor architecture. We believe that the kernel can be ported to different hardware platforms easily, because the kernel requires only a MMU and timer interrupt support. In this section, we discuss only the fundamental features of the kernel. The subject related to the RTOS Port Interface will be discussed in the following section.

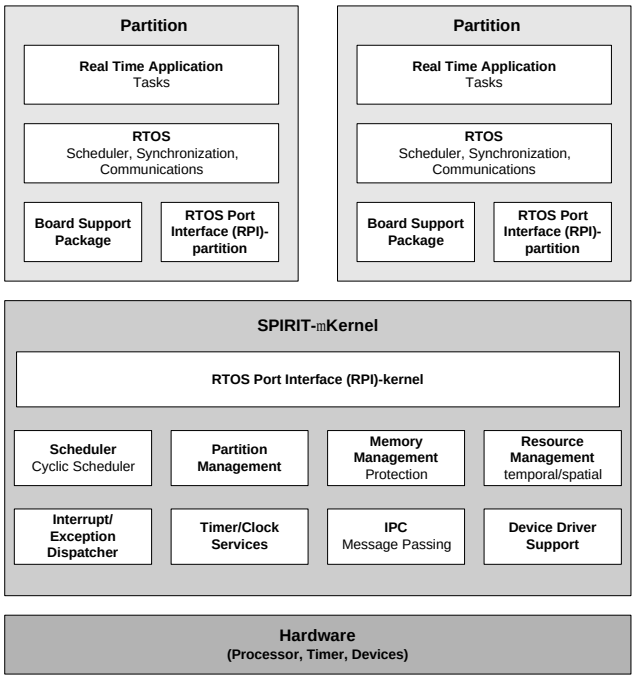


Figure 4-2 Architecture of the SPIRIT-µKernel

4.3.1 Memory Management

To guarantee strong spatial partitioning, we use a fixed two-level hierarchy of address spaces - kernel and partition address spaces. The kernel and partitions have their own protected address space in which their codes, data, and stacks are located. Protection of address space is implemented by a hardware protection mechanism like the memory management unit of a processor. We believe that an optimization of address space management to the underlying

processor architecture is the best solution to achieve both efficiency and deterministic access. A kernel address space is protected from illegal accesses of partitions, and a partition address space is also protected from potential interference from other partitions. Since the kernel is dependable and needs to be certified in safety-critical systems, we can allow the kernel to access the address spaces of all partitions.

In the SPIRIT-μKernel environment, address space allocation and access policy are determined at system integration time and saved in a secure kernel area. Like most real-time operating systems, we excluded the use of the virtual memory concept in which each partition is allocated independent virtual address space. While configuring an integrated system, we divide total physical memory space into non-overlapped regions that satisfy the memory requirement of each partition. Instead of using the full features of the memory management unit, we use an address translation from virtual (logical) address to physical address for protection purposes only. If we allocated a separate page table per partition for supporting a full virtual memory system, we would face poor deterministic behavior and performance, and increased page table size. In the SPIRIT-μKernel, since we use the same address for both virtual address and physical address, the page table size is only dependent on the size of physical memory.

One of the advantages of microkernel architecture is that the user application is allowed to have its own memory management scheme. To support this capability, the microkernel offers kernel primitives that can be used in dynamic configuration of the address space. The SPIRIT-μKernel also provides a limited capability of a dynamic memory management primitive for a partition. It provides a kernel primitive to allow a partition to change its page attributes dynamically. For example, using this primitive, a partition can change the access attribute of its code segment to read-only. When the primitive is issued by a partition, its validity is checked at kernel level before modifying the actual page attributes by the kernel. However, it is always

enforced that the ownership of a particular page, which is determined at system integration time, is not changed at run time.

To achieve high performance as well as protection, we optimized the memory management scheme of the SPIRIT-μKernel according to the PowerPC architecture. Both MMU BAT (Block Address Translation) and Segment/Page functions of the PowerPC are used to implement spatial partitioning. The BAT and Segment/Page table mechanisms are used to protect the kernel's address space and partition's address space respectively. Combined use of the two memory management techniques of the PowerPC enables us to achieve a very low overhead in kernel-partition switching and task switching within the partition address space. For example, there are expected to be many invocations of kernel primitives and interrupt/exception handlers. All these invocations potentially require context switching between the partition (user mode) and the kernel (supervisor mode). Using BAT and Segment/Page table methods for the kernel and partitions respectively, we can significantly reduce the kernel-partition switching overhead because the PowerPC can concurrently perform address translations and protection checks in both BAT and Segment/Page table mechanisms. In the case of partition switching, only one additional segment register loading would be sufficient.

4.3.2 Partition Management

An application partition is the basic scheduling entity of the SPIRIT-μKernel. A partition represents a well-defined and protected (both temporally and spatially) entity in which different kinds of real-time operating systems and their applications can be implemented. Different operating system personalities, such as local task scheduler, communication facilities, synchronization method, etc., can be implemented within the boundary of a partition. Due to the two-level static scheduling policy used in our SPIRIT, temporal and spatial resources are allocated to the partitions at system integration time. Therefore, all attributes of the partitions, including time slices, memory allocations, etc., are stored in the partition's configuration area in

the kernel space. Table 4-1 shows the important configuration parameters of a partition stored in the kernel.

Table 4-1 Partition Configuration Information

Class	Data	Description
Spatial Partitioning	Address range	Memory space allocated to a partition
Temporal Partitioning	Cyclic schedule table	Distance constraint guaranteed cyclic time allocation table
	Local time tick slice	Local time tick resolution of a partition
Miscellaneous	Event server	Address of event server of a partition
	Event Delivery Object	Address of event delivery object
	Initialization address	Start address of a partition
	Interrupt Flag	Address of interrupt flag to emulate user-mode interrupt enable/disable

Basically, we do not allow the shared libraries among the partitions because our goal is to protect against partition failure, even caused by the COTS kernel itself. So all partition code, including partition's local kernel, are executed in user mode. If we allow partitions to share the COTS real-time operating system kernel code and its libraries, a corruption or bug of the shared code would affect all the partitions that share the code. However, we allow shared libraries among the non-critical partitions to save memory space.

As described earlier, we use a distance-constrained cyclic partition scheduler to dispatch partitions. There are two possible methods in implementing a cyclic scheduler. One is to use a SPIRIT-μKernel time tick implemented by a periodic time tick interrupt with a fixed rate. In this case, the granularity of the execution time allocation to the partitions and local time tick slice of a partition are limited by the resolution of the kernel tick. Although it may decrease the schedulable utilization bound of the integrated system, it enables building a simpler and more predictable cyclic scheduler. The other approach is to use a reference clock and a timer of fine resolution. It has the advantages of accurate time allocation and avoiding a spurious kernel tick overhead,

however it increases the complexity of managing time both in the kernel and partitions. In the prototype implementation, we use the first approach, as used in MIT's Exokernel.

Since the SPIRIT-μKernel does not restrict the scheduling policies of the kernel scheduler and local task scheduler of a partition, it is easy to use other hierarchical scheduling policies in the SPIRIT-μKernel environment.

4.3.3 Timers/Clock Service

Timers/clock services provided by COTS RTOS generally rely on the time tick interrupt of the kernel, which has a relatively low resolution, such as 10ms, or 16ms. The sleep system call is a typical example of this service category. The other way of providing high-resolution timers/clock services is by implementing a special timer device and its device driver. In our environment, the latter can be implemented by the device driver model, which will be described later.

A partition maintains a local time tick that is synchronized to the global reference clock, the value of kernel tick. The local time tick interval of a partition must be multiples of the kernel tick interval. Since we use a cyclic scheduling approach, it is possible that the user timer service event could occur in the deactivation time of a partition. The solution to this is that every time a partition is resumed, it goes to a local event server first. The event server checks the events that have occurred during the deactivation time of the partition, and delivers events to the local kernel if needed. The event server will be discussed later in the paper

4.3.4 Exception Handling

We distinguish the interrupt and exception in designing the kernel. An interrupt is triggered from the external environment of the core processor, while an exception is triggered internally during the execution. For example, a time tick event is considered as an interrupt, while an illegal instruction violation event is an exception. The distinction is necessary as exceptions are usually processed in the context of their own partitions. For example, if an

exception occurs in the middle of a task, the exception handling routine of the corresponding partition should handle the exception as an event related to the task. We will discuss detailed exception handling using the event delivery object and the event server in the next section. In this sub-section we only describe, on a higher level, exception-handling issues related to the dependability of the integrated system.

Since the SPIRIT-μKernel has two-level hierarchical schedulers implementing strong partitioning concepts, a well-defined exception handling mechanism is essential to achieve the dependability of the integrated system. For example, when a fatal exception occurs while executing the partition's local kernel, it causes the crash of the partition. But, if an exception occurs while executing a task, the handling may be different. In the first case, we need to restart the crashed partition, or disable the partition and execute the idle partition server instead. In the latter case, we just suspend the failed task or restart the task.

4.3.5 Kernel Primitives Interface

To request a kernel service such as inter-partition communication, a partition calls kernel primitives. However, the kernel primitives cannot be directly called from a partition because a partition runs in a different privilege mode, and the object code format may be different. Therefore, we use a trap instruction of the PowerPC to allow a partition to call kernel primitives. Information exchanged between the kernel and partitions is stored in generic object code that is independent of the object file format.

4.3.6 Inter-Partition Communication

In complex and evolving real-time systems like SPIRIT, it is very important to provide scalable and robust communication facilities for communicating partitions. The legacy inter-process communication method used in traditional microkernel architecture is not sufficient to support all the requirements of SPIRIT, which needs special features like fault tolerance, one(many)-to-many partitions communications, supporting system evolution, and strong partitioning.

Therefore, we have selected a Publish-Subscribe model as the basic inter-partition communication architecture of SPIRIT-μKernel. The proposed Publish-Subscribe communication model was designed to achieve the following goals:

Supporting a variety of communication models – Due to the complexity of SPIRIT, it needs a variety of communication models, such as point-to-point, multicast and broadcast to serve various requirements of integrated partitions.

Fault tolerance – In SPIRIT, it is not uncommon to replicate applications for fault tolerance. With the enhanced Publish-Subscribe model we can implement a fault tolerant communication model such as a master/shadow model.

Evolvability/upgradability – Considering the relatively long life-time of complex real-time systems such as avionics, it is anticipated that the system will face modifications for system evolution and upgrade. A basic Publish-Subscribe model is a good candidate for this purpose.

Strong partitioning – Basically, all communication requirements are scheduled statically at system integration time. So the correctness and compliance of messages exchanged among partitions are to be checked at the kernel to protect other partitions from being interfered with by non-compliant messages.

In the inter-partition communication mechanism of the SPIRIT-μKernel, it is not recommended to use acknowledged message communication among partitions. This is because we use a cyclic scheduler for the partitions. When a sender sends a message and waits for the response from a receiver, it has to wait until the receiver resumes at the next assigned time interval, and the sender's own partition gets dispatched after that.

4.3.7 Device Driver Model

The SPIRIT-μKernel provides two device driver models for exclusive and shared devices. The exclusive device driver model can be used for the devices that are safety-critical or exclusively used by a partition. To allow a partition to access exclusive devices, the device I/O

address must be exclusively mapped to the address space of the corresponding partition. In safety-critical systems like avionics systems, it is preferred to use a polling method rather than an interrupt method, because a polling method is more predictable and reliable than an interrupt method. The kernel also provides a secure interrupt registration method for the exclusive device driver model as well as the polling method. The shared device driver model is used for devices that are potentially shared by multiple partitions using multiplexing/demultiplexing methods.

4.4 Generic RTOS Port Interface (RPI)

The SPIRIT-μKernel provides a generic RTOS Port Interface (RPI) that can be used to port different kinds of COTS real-time operating systems on top of the kernel. With the help of the RPI, the SPIRIT-μKernel itself can be built independently without considering the COTS RTOS. In this section, we describe the selected features of RPI, such as Event Delivery Object (EDO), Event Server, Kernel Context Switch Request (KCSR) primitive, user-mode Interrupt Enable/Disable emulation, and miscellaneous considerations. To give an overview of the RPI mechanism, we depict an example of partition switching using RPI in Figure 4-3.

In Figure 4-3, when a partition is to be preempted, the interrupt/exception dispatcher saves the context of the current partition and loads the context of the next partition using its control block (1). The dispatcher prepares the EDO, including the new loaded context, and delivers it to the event server of the next partition (2). The event server does housekeeping jobs for the next partition and also invokes the scheduler of the partition if needed (3). The updated partition context, to which CPU control goes, is delivered to the SPIRIT-μKernel via the Kernel Context Switch Request primitive (4). Finally, the kernel gives CPU control to the correct location given by the KCSR primitive (5). The detailed explanation will be given in subsequent sections.

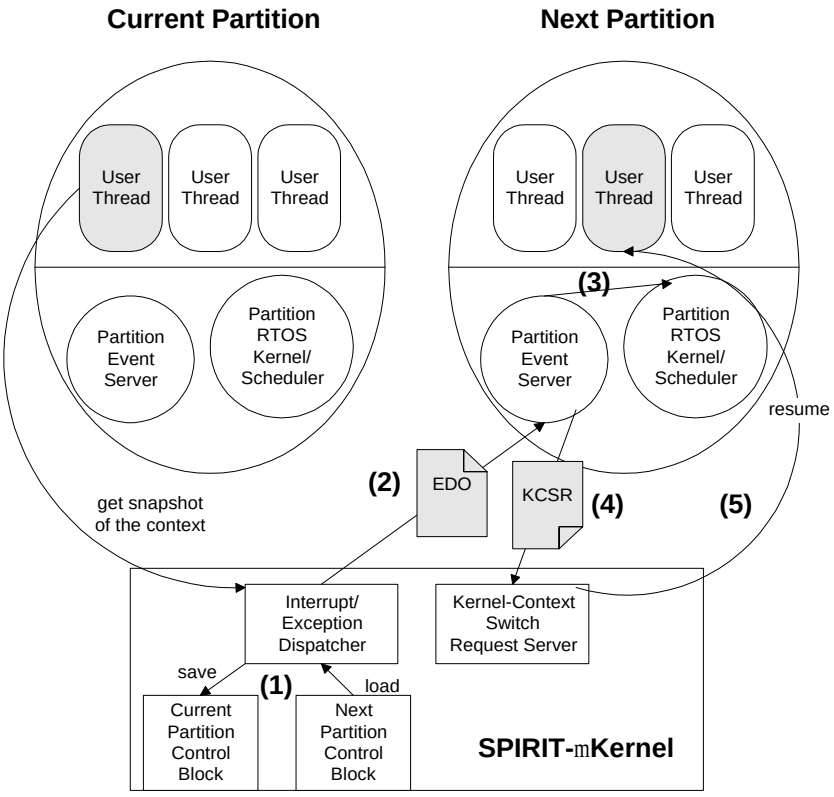


Figure 4-3 EDO, Event Server, and Kernel-Context Switch Request Primitive Interaction

4.4.1 Event Delivery Object (EDO)

A partition, which might be based on a COTS real-time operating system, needs to be informed of hardware events, such as time tick expiration, processor exceptions, and interrupts. Since these event classes are caught by the *SPIRIT-mKernel* in supervisor mode, secure and efficient delivery of an event from the kernel to a partition is necessary to be processed in the local event handling routine of a partition. In SPIRIT, it is not allowed for the kernel to directly execute RTOS-specific local event handling routines for two reasons. Firstly, if an event handler in RTOS's address space is called in the context of the kernel (supervisor mode), it may violate the strong partitioning requirements of the system due to the possible corruption or overrun of the event handler. Secondly, it increases the complexity of the kernel's interrupt/exception handling codes and data structures, because the kernel must have detailed knowledge of the RTOS design, which is vendor specific in COTS RTOS. So we have devised a generic event delivery method

that can be used to solve the problem. The event server and kernel context switching request primitive, introduced in the next two sub-sections, are used together with the event delivery method.

As mentioned above, to deliver an event securely and efficiently from the kernel to a partition, we have devised the Event Delivery Object (EDO). The EDO is physically located in the address space of a partition, so it can be accessed from both the kernel and its owner partition. We show the abstract structure of the EDO that is used in our PowerPC prototype in Figure 4-4.

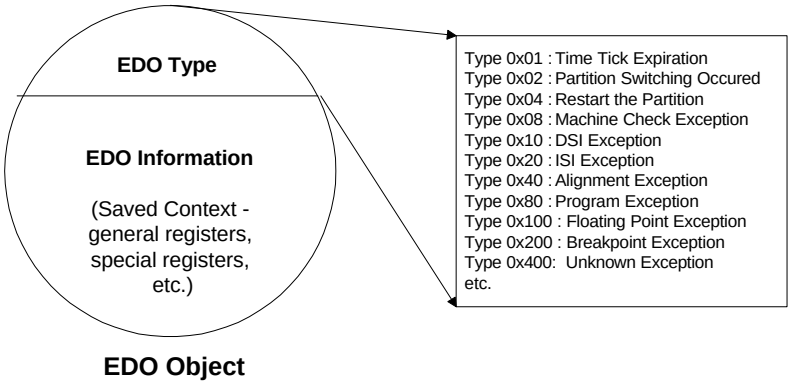


Figure 4-4 Structure of the Event Delivery Object

We describe the usage of the EDO with an example of task switching caused by partition time tick expiration delivery. There are two different task context-saving approaches in implementing RTOS. One is to save the task context that is caught at the instance of the time tick interrupt arrival. The other is to save the context in the middle of the time tick service routine of the RTOS. For instance, the first approach is used in VxWorks and the latter in eCos. In the SPIRIT-μKernel environment, the kernel must transfer the saved task context to a partition in the first case, but not in the second case. However, for the generality of the SPIRIT-μKernel, it also delivers the saved context in the second case. Based on the EDO mechanism, we have developed the generic interrupt and exception handling routine of the SPIRIT-μKernel shown in Figure 4-5.

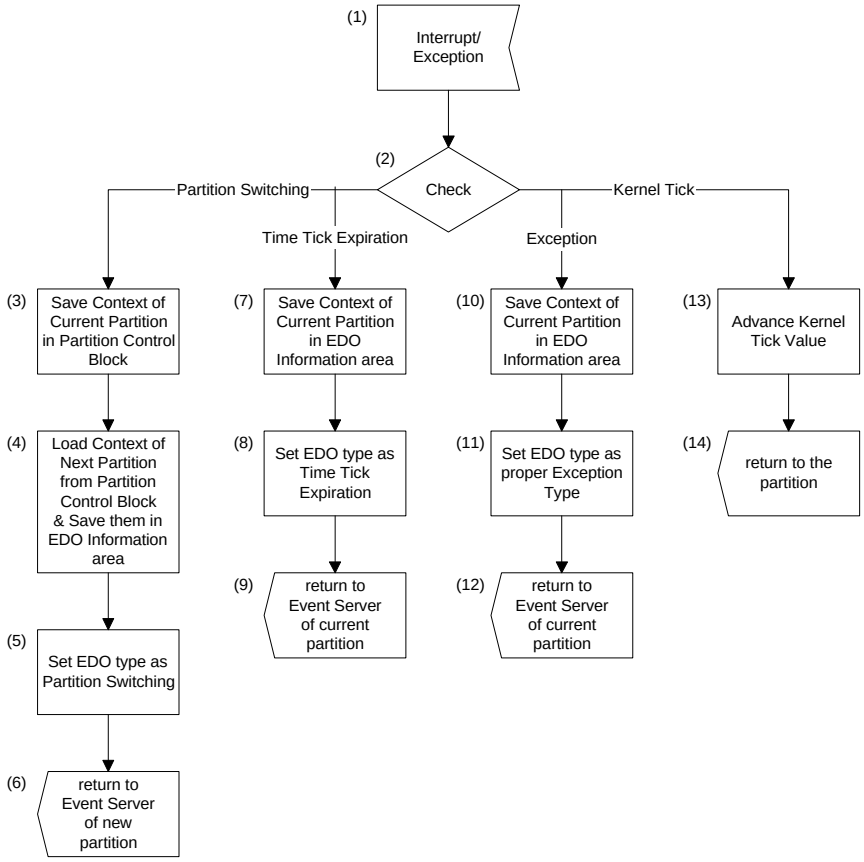


Figure 4-5 SPIRIT-μKernel's Generic Interrupt/Exception Handling Procedure

Responding to local time tick expiration, the local task scheduler may switch tasks based on its local scheduling policy and the status of tasks. In Figure 4-5, upon the arrival of the time tick expiration interrupt, the kernel saves the context in the EDO information area of the corresponding partition, and sets the EDO type as time tick expiration (7)(8). After preparing an EDO, the kernel sets the interrupt return address as the entry address of the partition's Event Server (9).

After return from the interrupt, now in user mode, the local event server checks the EDO and dispatches the proper local event handling routine. In the VxWorks case, receiving a time tick expiration, the local time tick interrupt handler checks the status of the VxWorks kernel to determine whether task switching is needed or not. When task switching is needed, it saves the context in the EDO information area in the task control block of the preempted task. Then the

event server issues a kernel context-switch request to the kernel providing the context of the newly dispatched task. While interrupt and exception handling routines are dispatched by the kernel in supervisor mode in the original COTS RTOS, the partition's local handling routines are dispatched by the event server in user mode in the SPIRIT-μKernel. Since all local event-handling routines are executed in user mode and in the local partition's address space, we can protect the kernel and other partitions from the faults of a partition. The overall procedure of the event server can be found in Figure 4-6.

4.4.2 Event Server

The event server of a partition is an essential part of the SPIRIT-μKernel. It has the role of a mediator between the kernel and partitions by executing partition's local interrupt and exception handling routines, and performing housekeeping jobs such as delivering local kernel events that occurred during the deactivation time of a partition. Figure 4-6 shows the generic event server procedure of a partition. The implementation of an event server depends on the corresponding RTOS. We explain the handling of the partition switching event as an example.

In Figure 4-6, when an event server is user-mode-called from the SPIRIT-μKernel, it checks the type of EDO and process the EDO properly. If the event is a partition switching, it executes the housekeeping jobs of a partition (3). Since there may be asynchronous local kernel events that arrived in the deactivation time of the newly dispatched partition, they must be delivered to the local RTOS kernel before dispatching the task of the partition. If rescheduling is needed in the local RTOS kernel and the current resume pointer is within the application task, not within a local kernel, the old task context that is in the EDO will be saved in the task control block, and the new context will be transferred to the kernel using the KCSR primitive (4) (5). It should be noted that the SPIRIT-μKernel would not allow any potentially harmful modifications to privileged registers based on security rules.

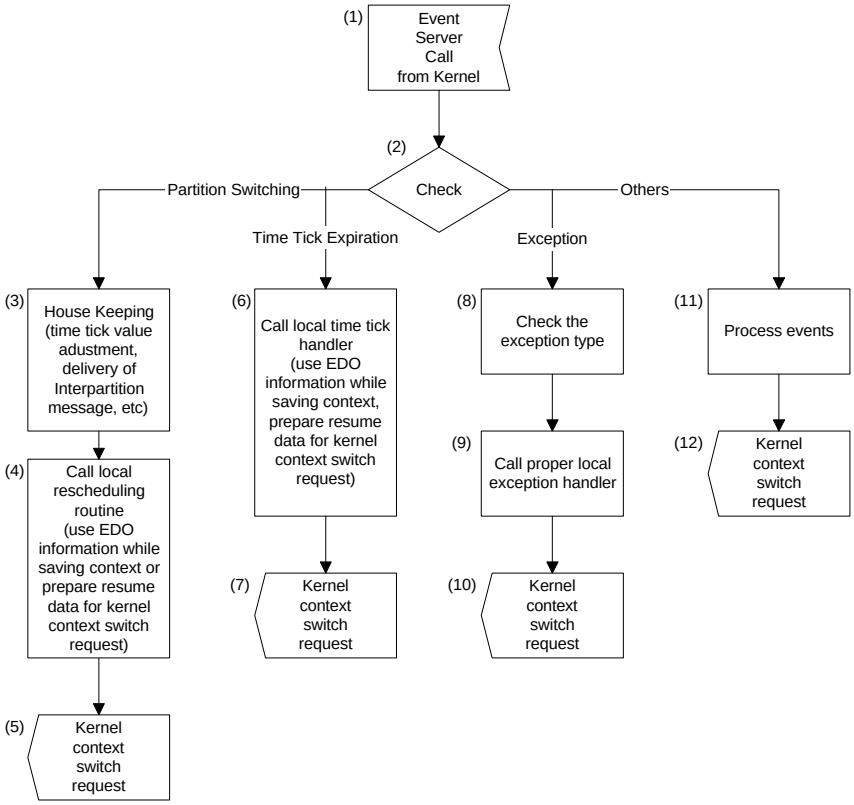


Figure 4-6 SPIRIT-μKernel's Generic Event Server Procedure

4.4.3 Kernel Context Switch Request Primitive

In most processor architectures, the context of a task is composed of both user-mode and supervisor-mode accessible registers. When a partition's local task scheduler tries to restore the context of the next dispatched task in the SPIRIT-μKernel system, a privilege violation exception may occur, because the context loader runs in user mode. To solve this problem, we have devised a kernel context switch request primitive, which delegates the kernel to perform a context-loading job. The kernel is provided with the restoring context information with which the kernel loads and resumes a new task. The primitive has two types; one is for loading privilege registers only, and the other for loading all registers. Since a large number of registers are user-mode accessible, it is better to use the first primitive in most cases for better performance. However, in some cases, these primitives are not needed because the context of a task is solely composed of user-mode accessible registers.

4.4.4 Interrupt Enable/Disable Emulation

Most COTS real-time operating systems use interrupt- enable and disable functions to guard critical sections in implementing system calls and the core of their kernels. But, in the SPIRIT-μKernel environment, we cannot allow a partition to execute interrupt control functions, because these functions must be run in supervisor mode. Disabling interrupts may block the SPIRIT-μKernel for a significant amount of time. To solve this problem, it is required to re-write the original interrupt- enable and disable functions, while guaranteeing its original intention of protecting critical sections from interrupts.

We use atomic set and reset instructions to implement replacements of the original interrupt enable and disable functions. In the PowerPC architecture, atomic set and reset logical functions are implemented by a reservation-based instruction set. In a similar way to implementing a binary semaphore, disabling interrupts is emulated by an atomic set instruction for a variable, interrupt flag, which is located in the partition address space and shared by the kernel and the owner partition. While the interrupt flag is set, the kernel delays the delivery of an interrupt until the interrupt is enabled. The interrupt enable function is implemented by the atomic reset instruction, which clears the flag.

4.4.5 Miscellaneous Interface

There are other interfaces to be considered while porting new COTS real-time operating systems on top of the SPIRIT-μkernel. An example of these miscellaneous interfaces is related to hardware initialization. Most real-time operating systems provide a board support package that is used to initialize and configure the hardware. The board support package includes hardware dependent activities such as cache enabling and disabling, MMU manipulation, etc. Since only the SPIRIT-μKernel initializes and configures shareable hardware resources to ensure dependable operation, hardware dependent functions in a board support package must be removed.

4.5 Performance Evaluation

In order to prove and demonstrate the feasibility of the proposed SPIRIT-μKernel, we have implemented a prototype on a DY4-SVME171 commercial-off-the-shelf embedded CPU board. The board houses an 80MHz Motorola PowerPC 603e and 32MB of main memory. The PowerPC 603e provides a memory management unit that is essential for implementing spatial partitioning. The board provides a programmable periodic timer interrupt for kernel tick implementation. These two hardware components are the minimum requirements of the SPIRIT-μKernel. On top of the hardware platform, we have implemented the SPIRIT-μKernel that cyclically schedules heterogeneous COTS RTOS partitions, Windriver's VxWorks and Cygnus's eCos. The reason we chose VxWorks and eCos for COTS RTOS partitions is that they are extreme cases to be considered in designing the kernel. While the VxWorks provides binary kernel and modifiable board support packages written in C, eCos provides full C++ kernel sources. The size of the SPIRIT-μKernel code is 36 Kbytes.

In evaluating the prototype we integrate and run four different partitions, two VxWorks-based and two eCos-based applications. We measure the execution time using the PowerPC time base register, which has a 120 ns resolution. To evaluate the performance of the kernel, we have measured and analyzed the overheads for kernel tick, partition switching, kernel-user switch, and TLB miss handling.

4.5.1 Kernel Tick Overhead

Since the kernel and partition's local RTOS are synchronized to and scheduled based on the kernel tick, the resolution of the kernel tick is an important factor that affects the system's performance and schedulability. To evaluate the overhead of the kernel tick, we obtained two basic components, *kt_overhead1* and *kt_overhead2*. *kt_overhead1* is measured when a kernel tick is used for neither partition switching nor local partition time tick expiration. *kt_overhead2* is the

time taken to deliver the EDO type of Time_Tick_Expiration to the event server of the local partition. We show the result in Table 4-2.

Table 4-2 Measured Kernel Overheads

kt_overhead1			kt_overhead2		
Avg	Min	Max	Avg	Min	Max
0.84 μs	0.75 μs	3.88 μs	9.38 μs	9.00 μs	18.38 μs

To help find the feasible kernel tick resolution, we calculate *r_kt_overhead* and *p_kt_overhead* that are the percentages of the redundant kernel tick overhead and total kernel tick overhead for a partition's local task scheduling, respectively. *kt_period* and *pt_period* represent the length of kernel time tick and partition time tick respectively.

$$r\_kt\_overhead = \frac{kt\_overhead1}{kt\_period}$$

$$p\_kt\_overhead = \frac{kt\_overhead1 * (\frac{pt\_period}{kt\_period} - 1) + kt\_overhead2}{pt\_period}$$

In Figure 4-7, we depict average and worst case kernel tick overheads that are obtained by varying the kernel tick period, 100us, 500us, 1ms, 2ms, 5ms, with a fixed partition local tick period of 10ms.

As illustrated in Figure 4-7, while the kernel tick period is greater than 500μs, each of all four overheads is less than 1% of the total CPU capacity. These practically acceptable overheads were possible due to the efficient design of the kernel tick interrupt handler and low latency interrupt handling support from the PowerPC processor. In the instance of the redundant kernel tick event, the kernel tick interrupt handler increments the kernel tick value and compares it with

the pre-computed kernel tick value of the next events of either partition switching or local time tick expiration, and returns from the interrupt.

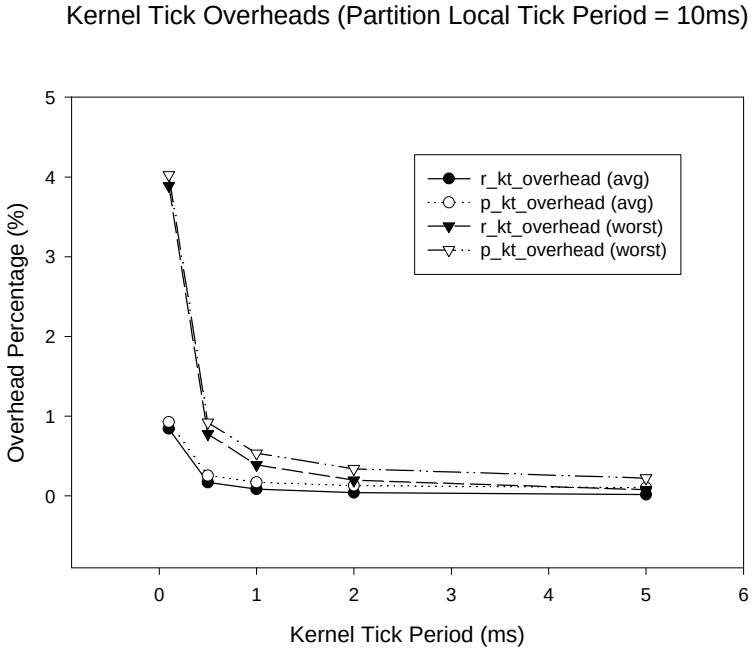


Figure 4-7 Kernel Tick Overheads

4.5.2 Partition Switch Overhead

Based on the off-line scheduled timetable, the kernel scheduler dispatches partitions. When a partition-switching event occurs, the kernel scheduler saves the context of the current partition and reloads the saved context of the next partition. Then CPU control goes to the event server of the newly dispatched partition. Since the kernel has a generic partition-switching handler regardless of the types of RTOS that are saved or restored, we expect deterministic partition switch overheads. Table 4-3 shows the partition switch overheads.

Table 4-3 Partition Switch Overheads

Avg	Min	Max
10.50 μs	10.00 μs	21.50 μs

4.5.3 Kernel-User Switch Overhead

As in a conventional operating system, the kernel-user mode switch overhead is an important factor for evaluating the performance of the system. To reduce the overhead, we use the BAT and segment/page table schemes for the kernel and a partition respectively. Since both schemes are executed concurrently during the address translation and a result is chosen according to the current processor mode, the only overhead for the kernel-user switch is the time needed to set or reset the processor mode bit in the machine state register. So we can claim that the pure overhead due to the kernel-user switch, while distinguishing the processor modes, supervisor and user mode, can be ignored.

Instead of measuring the pure kernel-user switch overhead, we measured the kernel-context switch request primitive, which is an essential method of CPU control transfer, which requires supervisor-mode privileges, in user mode. The execution time of the primitive is measured and listed in Table 4-3.

4.5.4 TLB Miss Handling Overheads

In many real-time applications, virtual memory is not supported and supervisor and user modes of execution are not distinguished. However, it is an essential principle to distinguish supervisor and user modes to guarantee the spatial partitioning concept. The SPIRIT-μKernel uses the PowerPC MMU's segment/page scheme to protect the address space of partitions. Since this scheme relies on virtual to physical address translation, the performance of the TLB (Translation Look-aside Buffer) scheme of the PowerPC is very important. Considering the fairly uniform distribution of memory access in the SP-RTS environment due to partitioning, we should pay much attention to the performance of the TLB. In the prototype, it requires 8K page table entries for 32Mbytes of physical memory. However, the PowerPC provides only 64 page table entries for ITLB and DTLB respectively. We measured the TLB miss handling overhead and showed in Table 4-4.

Table 4-4 TLB Miss Handling Overheads

Instruction TLB			Data TLB Load			Data TLB Store		
Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
1.41 μs	0.88 μs	3.50 μs	2.43 μs	1.63 μs	4.25 μs	2.22 μs	1.13 μs	4.88 μs

4.6 Conclusion

The SPIRIT-μKernel is designed to provide a software platform for Strongly Partitioned Integrated Real-Time Systems, such as Integrated Modular Avionics Systems. Using the kernel, we can achieve better reliability, reusability, COTS benefits, and cost reduction in building complex integrated real-time systems. The kernel is assumed to use scheduling works presented in Chapter 2.

For further study, we are planning to enhance the kernel in three directions. Firstly, we will build our own local kernel for a partition instead of using COTS RTOS. Secondly, we will extend the current uni-processor architecture to a multiprocessor and distributed real-time computing environment. Thirdly, we will develop other two-layer scheduling strategies, instead of cyclic/priority driven scheduler pair, while guaranteeing the strong partitioning requirement.

CHAPTER 5
REAL-TIME COMMUNICATION NETWORKS FOR INTEGRATED REAL-TIME SYSTEMS

5.1 Introduction

Given that safety-critical real-time systems require deterministic behaviors in their communication operations, expensive and specialized communication networks have been used in the areas of avionics, defense, and space applications. For example, the MIL-STD and Aeronautical Radio Inc. (ARINC) standard communication networks are popular for building avionics systems [3]. One of the problems of using these networks is that it is difficult to find engineers who are experienced with them. Also, this type of communication networks is expensive and has a limited bandwidth. An alternative way is to make use of popular and COTS network technology wherever the communication delays can be made predicable.

In our integrated real-time systems model, we assume that real-time communication networks should be compatible with the proposed scheduling theory. Basically, it must conform to cyclic scheduling. The most popular protocol is a kind of Time Division Multiplexing Protocol (TDMA), which is used in IMA standards [3] and also assumed in the previous scheduling work presented in Chapter 2 and Chapter 3. In this chapter, we propose to adopt Ethernet technology for real-time communication networks for integrated real-time systems.

Currently, Ethernet is the dominant network technology because of its cost-effectiveness, high bandwidth, and availability. With a Carrier Sense Multiple Access/Collision Detect (CSMA/CD) MAC protocol, Ethernet is inherently incapable of guaranteeing deterministic accesses due to possible packet collision and random back-off periods. However, in order to take advantage of COTS Ethernet in real-time systems, it is preferred not to change the standard network hardware components and protocol so that most advanced and inexpensive Ethernet

products can be employed without any modifications. The issue is then how to design an easily accessible, reliable, deterministic, and affordable real-time Ethernet for safety-critical real-time systems using COTS Ethernet hardware and software methods. In this chapter, we propose a Table-Driven Proportional Access (TDPA) protocol in the standard Ethernet MAC layer to achieve guaranteed real-time communication. The TDPA protocol is a good candidate to implement cyclic scheduling, which is practically used in safety-critical real-time systems. The protocol uses a master to signal the starting instance of slots that are allocated to channels of fixed capacity. There are four types of channel, for control, single message, multiple messages, and Ethernet traffic. A complete scheduling approach for partition-based systems can be applied to establish capacity requirement for each channel.

The rest of the chapter is organized as follows. We introduce the proposed Strongly Partitioned Real-time Ethernet (SPRETHER) architecture and system model in Section 5.2. The details of the protocol are presented in Section 5.3 and its scheduling approach is shown in Section 5.4. The prototype platform and performance measurement are discussed in section 5.5. A short conclusion follows in section 5.6.

5.2 SPRETHER Architecture and Model

The SPRETHER has been developed for the Strongly Partitioned Integrated Real-Time System (SPIRIT), which emphasizes the integration of real-time applications in a common multiprocessor platform. The SPIRIT utilizes multiple standardized processor modules in building functional components of comprehensive real-time systems. While permitting resource sharing among applications, the integration approach employs temporal and spatial partitioning to set up the application boundaries needed to maintain system predictability, real-time response, and dependability. Each processor can host multiple partitions in which applications can be executed using the assigned resources. Spatial partitioning implies that a partition cannot access other partitions' resources, like memory, buffers, and registers. On the other hand, temporal partitioning guarantees a partition's monopoly over the use of a pre-allocated processing time

without any intervention from other partitions. As a result, the applications running in different partitions cannot interfere with each other. To facilitate communications between applications, each partition can be assigned with one or more communication channels. An application can transmit messages during the time slots assigned to its channel and access exclusively the channel buffers.

In Figure 5-1, we simplify and generalize the SPIRIT model to focus on communication aspects of the SPRETHER. A channel is a basic scheduling entity of the SPRETHER. Each channel server can support single or multiple message streams. To support non-real-time applications, a dedicated non-real-time channel server is provided. A two-level hierarchical scheduling method is applied to schedule channels and messages. A channel server, which supports multiple real-time message streams, provides a fixed-priority preemptive scheduling. Then, a cyclic schedule assigns Ethernet bandwidth to the channel servers of multiple partitions. Note that the preemption unit of fixed-priority scheduling is a fixed size packet and a message is segmented into multiple packets for transmission.

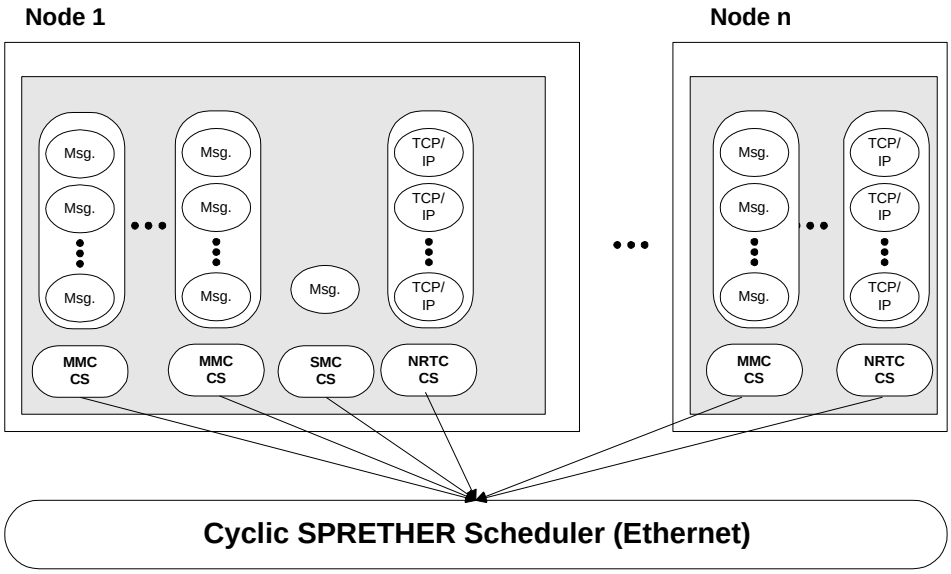


Figure 5-1 SPRETHER Real-Time Communication Model

We show the message stream model in Figure 5-2. A message stream is specified by several parameters, including message period or the minimum inter-arrival time (T_i), message size (M_i), and deadline (D_i).

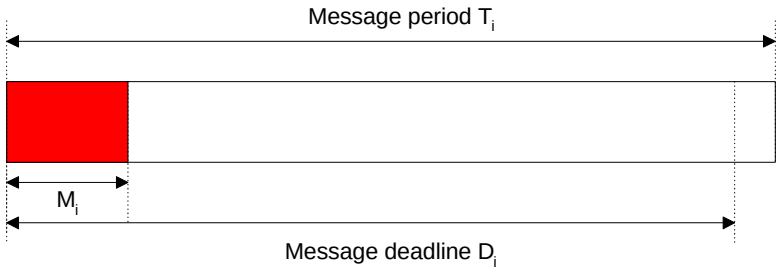


Figure 5-2 Message Stream Model in the SPRETHER

We assume that the scheduling of all real-time message streams is done off-line, and a cyclic table for message transmission and reception can be constructed. To utilize the shared media of an Ethernet, we adapt a Table-Driven Proportional Access (TDPA) based protocol on top of the Ethernet MAC protocol. The TDPA protocol is implemented using a synchronized RX and TX table in each node. All transmissions and receptions are invoked based on timing, and spatial information stored in the table. The proposed network model is shown in Figure 5-3. It has the following characteristics:

Software Based Frame Synchronization. Synchronization of message transmission and reception based on the cyclic table is essential to guarantee real-time constraints in the TDPA protocol. Since our approach employs COTS Ethernet hardware, it is required to develop software based frame (cyclic table) synchronization.

Publish/Subscribe Communication Model. The communication mechanism we adopted in the SPRETHER has the feature of supporting a publish/subscribe communication model, which has proved to be useful for the upgrade and evolution of real-time systems and is a fundamental communication model of the IMA system.

CSMA/CD Support for Non-Real-Time Applications. One of the advantages of the SPIRIT model is to support various applications of different criticalities. So non-real-time applications can be run on the same processor with hard real-time applications. The proposed real-time Ethernet is required to support the CSMA/CD Ethernet protocol, which can be used to transmit non-real-time TCP/UDP/IP packets.

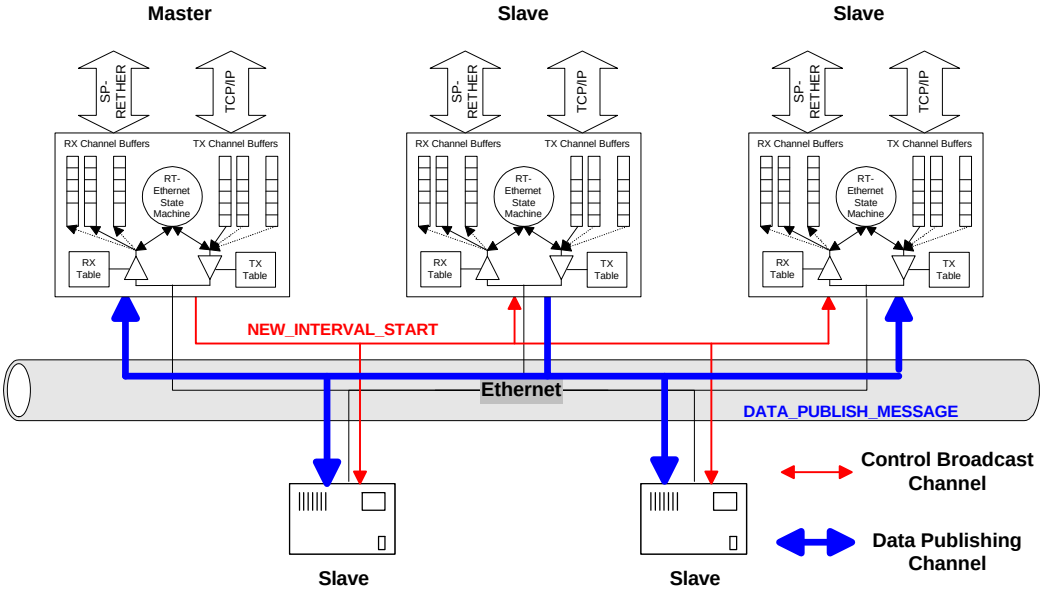


Figure 5-3 Strongly Partitioned Real-Time Ethernet Architecture

5.3 SPRETHER Protocols

5.3.1 Table-Driven Proportional Access Protocol

SPRETHER supports four different channels; single message stream channel (SMC), multiple message stream channel (MMC), non-real-time channel (NRTC), and control channel (CCH). The channels are configured following a time-multiplexing approach that is defined in a cyclic scheduling table as illustrated in Figure 5-4.

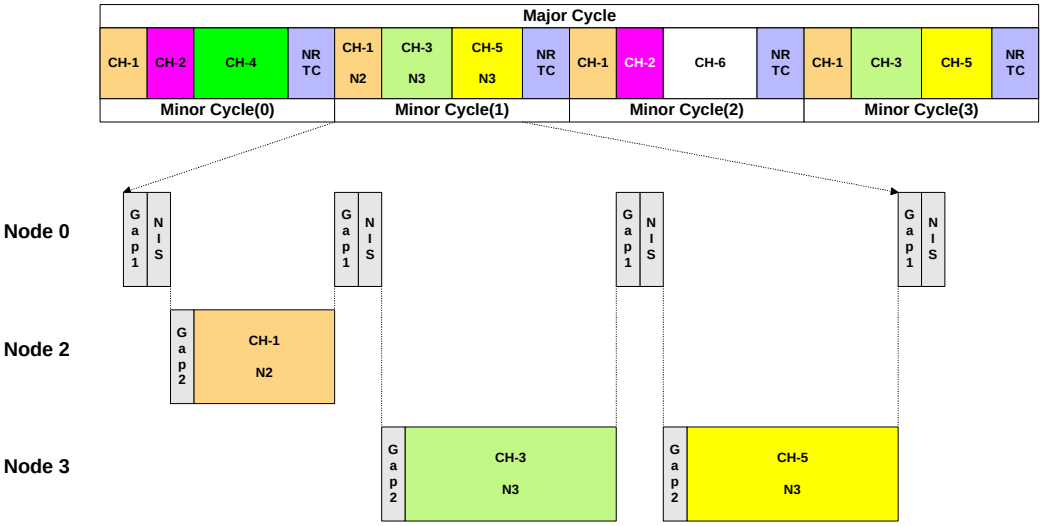


Figure 5-4 Channels and Gaps Configuration According to a Cyclic Scheduling Table

The control channel carries SPRETHER control messages and is a broadcast Ethernet channel (with an Ethernet broadcast MAC address) from the master node to all slave nodes. The most important control message is the New Interval Start (NIS) packet that is broadcast to all slave nodes by the master. The packet is used to establish a global frame synchronization according to the cyclic scheduling table, as well as to indicate the beginning of the next transmission interval that is assigned as a SMC, MMC, or NRTC within a frame.

Single or multiple message stream channels carry all real-time message and are configured as another broadcast channel in Ethernet (with a fixed multicast MAC address). According to the cyclic scheduling table, the transmission intervals are assignment to SMCs MMCs, and NRTC. There is one sender per SMC and MMC, which is responsible for publishing its message to all possible subscribers. Using an expiration timer and segmenting messages, a sender must ensure that its transmission doesn’t overrun the allocated intervals. Thus, there will be no collision in SMCs and MMCs.

Non-real-time messages can only be transmitted via a non-real-time channel that is allocated with intervals according to the scheduling table. Once a node detects the beginning of non-real-time channel based on NIS packets, it can send out any waiting non-real-time messages

following the conventional Ethernet CSMA/CD mode and with the constraint that the transmission should not overrun the allocated interval. A detailed discussion of this feature will be given in Section 5.3.4.

As the illustration of a cyclic schedule for the TDPA protocol (Figure 5-4) shows, all channel slots are contention free except those of NRTC, which allows all nodes to participate in CSMA/CD transmission. The gaps between two consecutive transmission intervals can be one of two types; Gap1 and Gap2. Gap1 tolerates possible jitter caused by a sender at the end of the current slot. This jitter could be a result of clock drifting or asynchronous services in the node. Gap2 compensates for the propagation delay and processing time of a NIS packet. The worst-case delay of gaps should be calculated and included in the channel scheduling and capacity assignment. The cyclic scheduling table is constructed off-line and put into the SPRETHER kernel space. The data structures such as the cyclic table, and the transmission and reception buffers of the channel, are initialized and allocated at system boot time. An operational example of the TDPA based SPRETHER protocol is given in Figure 5-5.

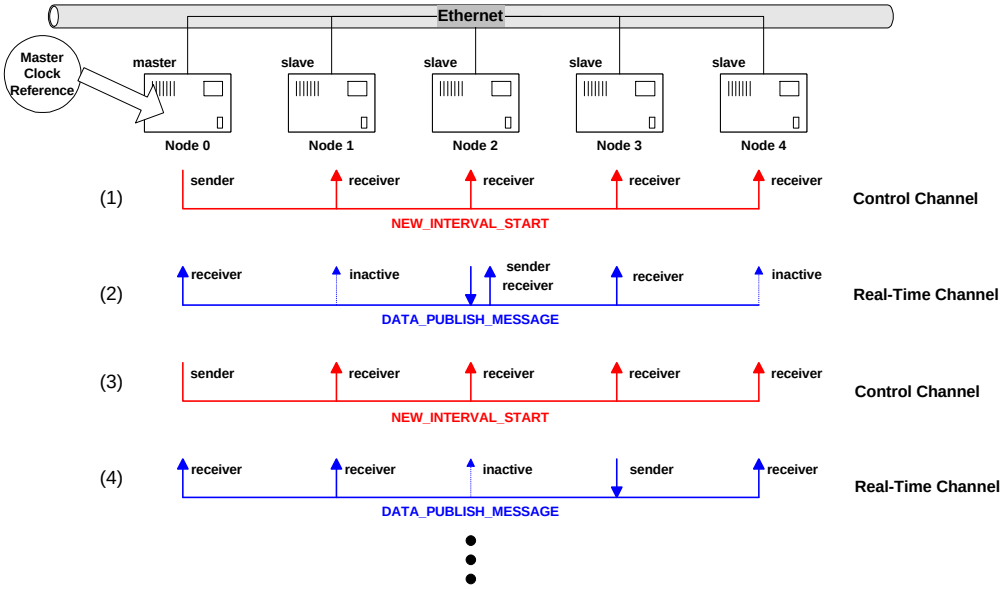


Figure 5-5 Example of TDPA the Real-Time Ethernet Protocol

5.3.2 Single Message Stream Channel (SMC)

A SMC channel can be easily supported by the SPRETHER because it transmits only one message arrival stream. The capacity allocated to a SMC of message stream *i* must be larger than $M_i/min(T_i, D_i)$ and must compensate for the drift of synchronization due to the propagation delays and processing times of NIS and data packets. We illustrate the protocol activities of the SMC channel in Figure 5-6.

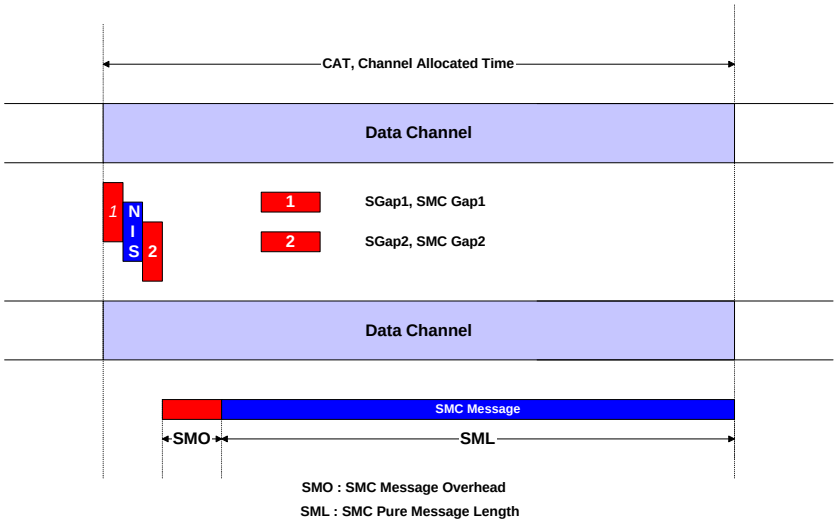


Figure 5-6 Analysis of SMC Channel Activity

In the SMC protocol, there are two overhead gaps; *SGap1* and *SGap2*. Table 5-1 shows the parameters that constitute the overheads of the SMC channel.

Table 5-1 Parameters Used in the SMC Protocol

Notation	Description
MIG	Minimum Interframe Gap of Ethernet (9.6μs)
SGap1	Max {MIG, Preparation time of NIS} = 99.2 μs (refer Table 5-4)
S(NIS)	NIS packet transmission time = 86 μs (refer Table 5-4)
SGap2	Max {MIG, Processing Time of NIS} = 33.84 μs (refer Table 5-4)
SMO	Ethernet Frame Overhead (26 Bytes = 26 x 0.8μs = 20.8μs) + SPRETHER Packet Overhead (12 Bytes = 12 x 0.8us = 9.6μs) = 30.4μs
CAT	Channel Allocation Time

Then, the utilization of the SMC channel can be obtained with the following equation:

$$Util_{SMC} = 1 - \frac{SGap1 + S(NIS) + SGap2 + SMO}{CAT}$$

We show the utilization graph of the SMC protocol in Figure 5-7. According to the figure, we can obtain 50% and 70% utilization when CAT is 499 μs and 832 μs respectively.

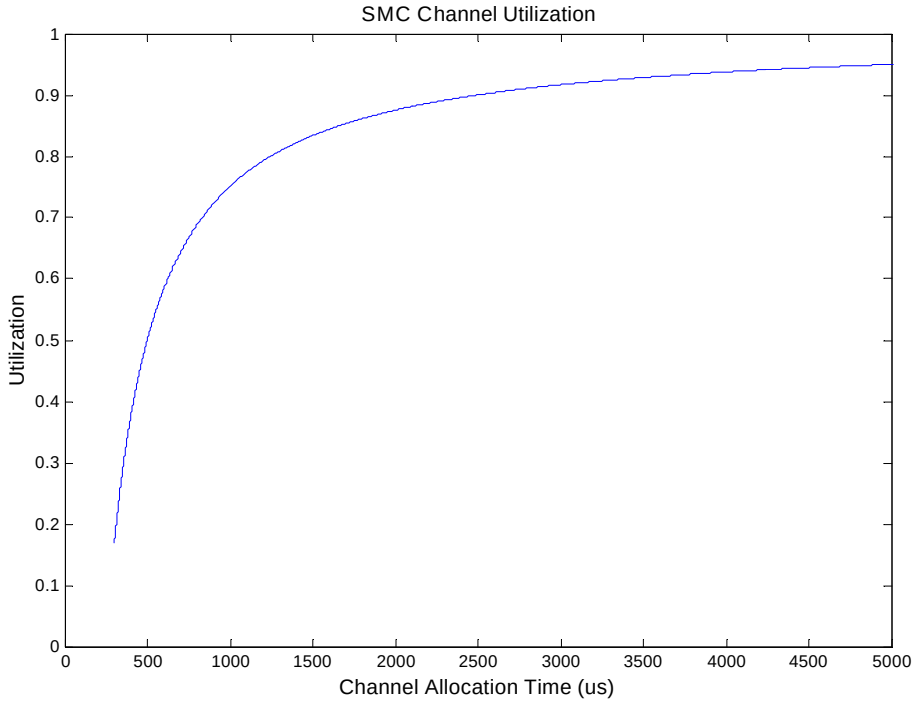


Figure 5-7 Utilization of SMC Channel

5.3.3 Multiple Message Streams Channel (MMC)

A multiple message channel can be used to serve several message streams simultaneously. We assume that messages are with fixed priorities that follow rate-monotonic or deadline-monotonic ordering. Since a transmitting packet should not be stopped arbitrarily, a higher-priority message that just arrived could be blocked. To reduce the blocking factor caused by non-preemption of packet transmission, we segment a message into packets of fixed size. The bigger the packet size, the more the blocking delay. This blocking factor is unavoidable in Ethernet because when the packet is sent to the Ethernet hardware, it cannot be cancelled by the

arrival of a higher-priority packet (or message). The segmentation involves disassembly and reassembly procedures. We illustrate the protocol activities of the MMC channel in Figure 5-8.

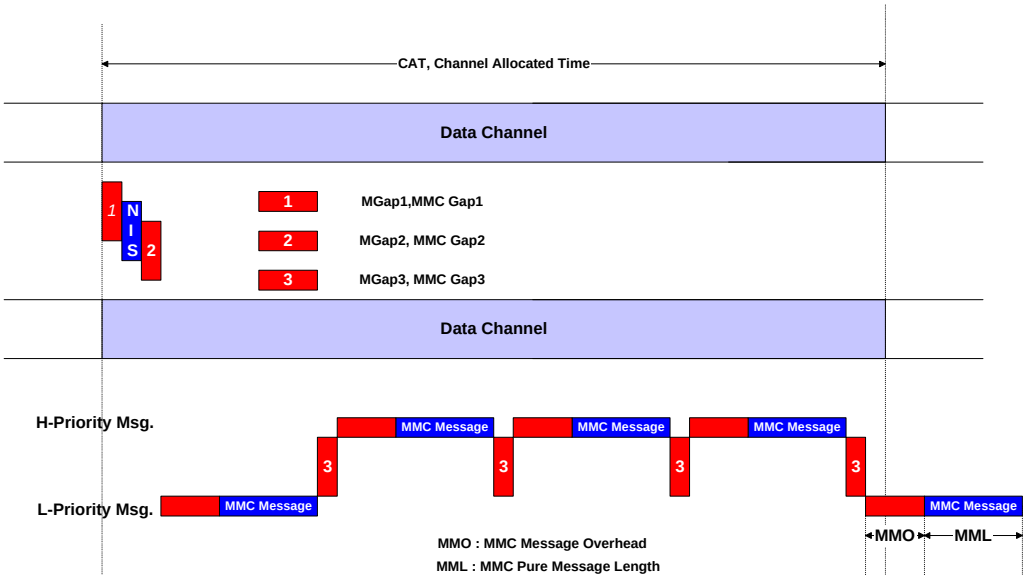


Figure 5-8 Analysis of MMC Channel Activity

If the last packet is sent just before the time of expiration of the current channel interval, the transmission will cause an overrun. To avoid an overrun, a MMC channel must monitor the amount of time left before the expiration of the channel interval, and determine a packet of suitable size for transmission. There is a trade-off choosing the size of a fixed packet. If we choose a small packet size, a message will be segmented into many packets, which causes an increased number of inter-packet gaps (*MGap3*) and MMO overheads. On the other hand, if we select a large packet size, the blocking delay may result in an increase of the bandwidth requirement for the channel. So it is required to select the optimal packet size based on the characteristics of arrival message streams. In the MMC protocol, there are three types of timing gaps; *MGap1*, *MGap2*, and *MGap3*. Table 5-2 shows the parameters that are used in obtaining utilization of a MMC channel.

Table 5-2 Parameters Used in the MMC Protocol

Notation	Description
MGap1	Max {MIG, Preparation time of NIS = 99.2 μs, MMO+MML} (refer Table 5-4)
MGap2	Max {MIG, Processing Time of NIS} = 33.84 μs (refer Table 5-4)
MGap3	Minimum Interframe Gap (MIG) of Ethernet (9.6 μs)
S(NIS)	NIS packet transmission time = 86 μs (refer Table 5-4)
MMO	Ethernet Frame Overhead (26 Bytes = 26 x 0.8μs = 20.8μs) + SPRETHER Packet Overhead (12 Bytes = 12 x 0.8us = 9.6μs) = 30.4 μs
MML	Pure Message Size in Bytes x 0.8 μs

Then, the utilization of the MMC channel can be obtained with the following equation:

$$Util_{MMC} = 1 - \frac{MGap1 + S(NIS) + MGap2 + \left((MMO + MGap3) * \left\lceil \frac{CAT - (MGap1 + S(NIS) + MGap2)}{(MMO + MML + MGap3)} \right\rceil \right)}{CAT}$$

The fully analyzed utilization of the MMC channel is illustrated in Figure 5-9. A utilization of below zero means that it is not schedulable.

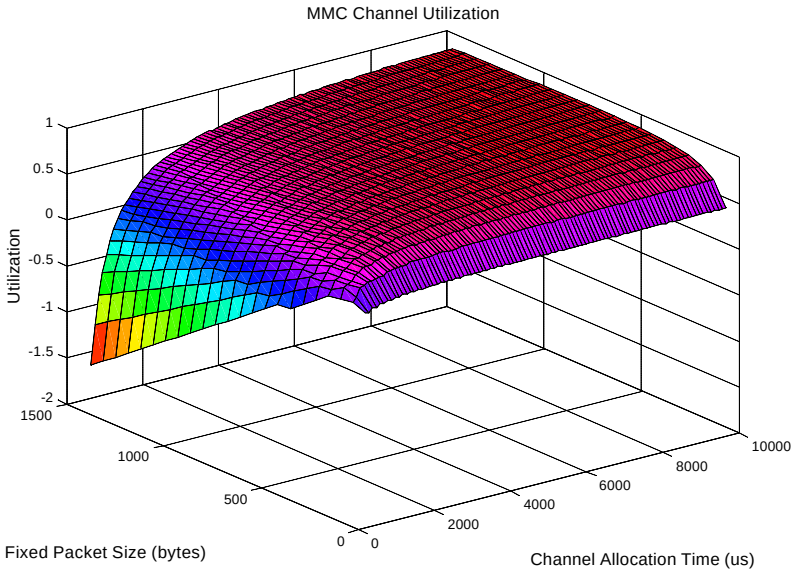
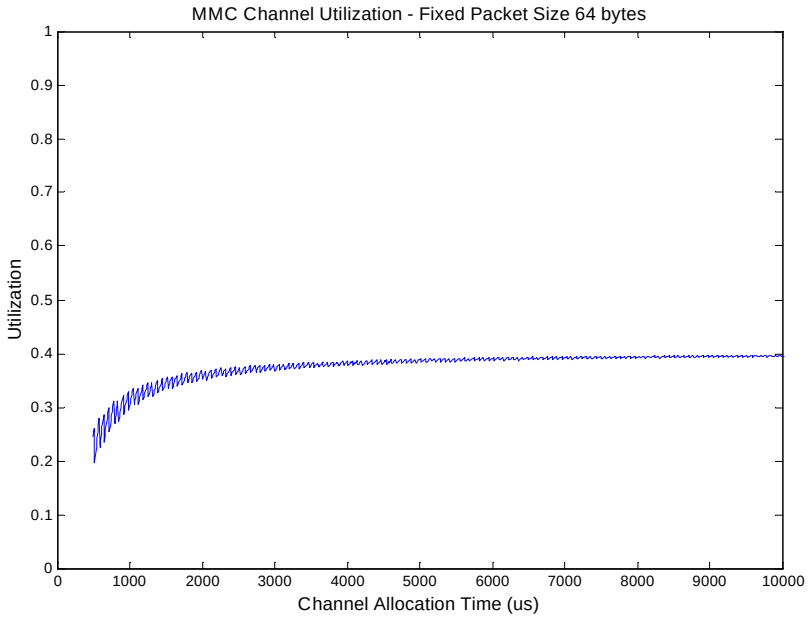
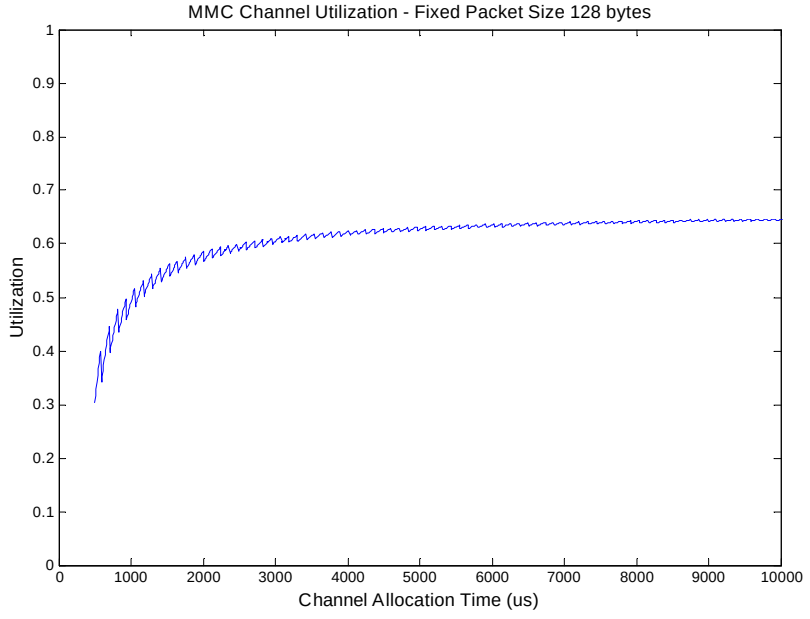


Figure 5-9 Utilization of MMC Channel

We also show the utilization of the MMC channel when the size of fixed packets varies from 64, 128, 256, and 512 bytes, in Figure 5-10.





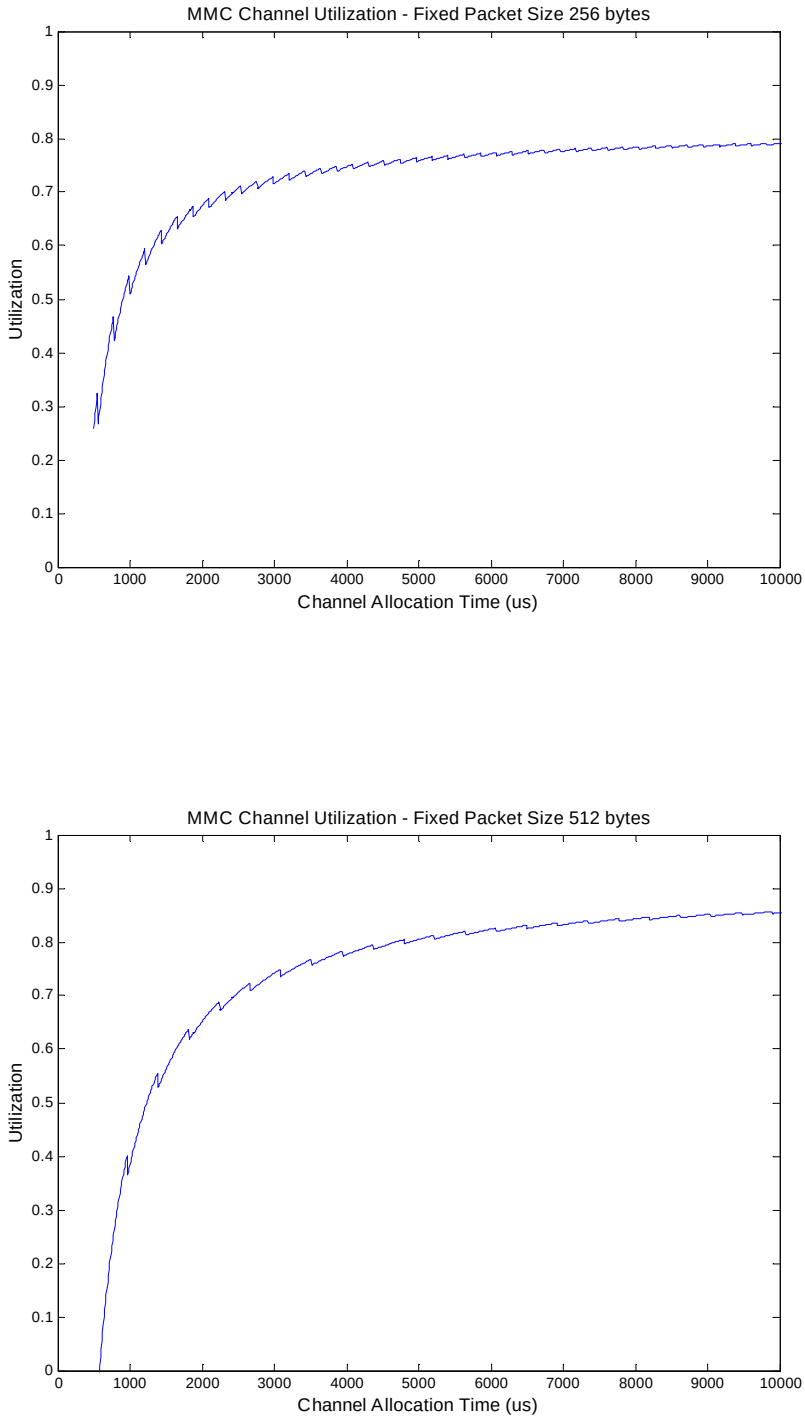


Figure 5-10 Utilization of MMC Channel when Packet Size is 64, 128, 256, and 512 Bytes

5.3.4 Non-Real-Time Channel (NRTC)

We allow a node to use CSMA/CD mode operation when the current time interval falls on the non-real-time channel (NRTC). We illustrate an NRTC in the cyclic time frame and show NRTC packet transmission behaviors in Figure 5-11. A node, which participates in NRTC mode operation, must not overrun the scheduled NRTC time interval. Thus, a message may be segmented for a possible transmission before the end of the expiration of the channel interval. However, a standard CSMA/CD protocol does not guarantee it because of the retransmission with random back-off algorithm. As the number of participating nodes increases, we can see that the statistical upper bound of expected completion time of the packet transmission becomes unacceptable in real-time communication. So we use an enforcement technique with which we can stop a packet retransmission if there is a collision. Even though we cancel the on-going packet transmission, we aim to achieve the maximum utilization of NRTC.

We call the enforcement method the Stop Gracefully (SG) technique. Whenever a sender requests a transmission to Ethernet hardware, it sets a local timer to the value of SGT (Stop Gracefully Timer value). If the transmission succeeds before timer expiration, it cancels the timer. Otherwise it issues a Stop Gracefully command to the Ethernet hardware upon the timer expiration. Then the Ethernet hardware either succeeds in transmitting the packet by the end of the time interval, or cancels and postpones the failed transmission due to a collision. The value of the SGT for packet j should be set to $D - S(M_j)$ where $S(M_j)$ is the expected time being taken to transmit packet M_j and D is the end of the current channel interval.

The implementation of the SG technique varies with the Ethernet chipset being used. For instance, in the case of the Motorola MPC860, its CPM (Communication Processor Module) supports the Stop Gracefully command. The 80C03 of LSI logic provides a method that stops the transmission upon the first collision.

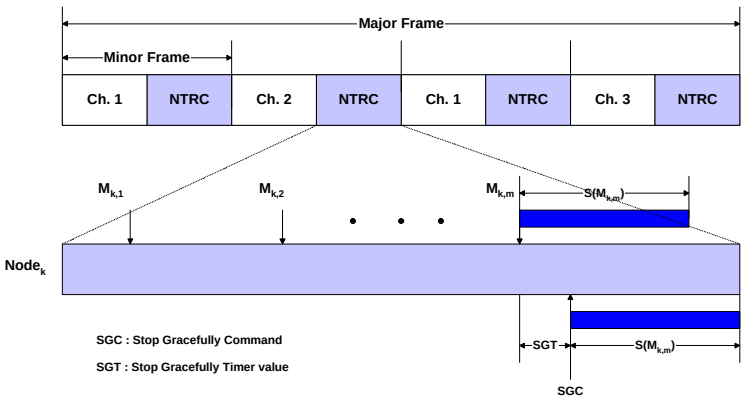


Figure 5-11 Protection Mechanism of the Real-Time Channel

5.3.5 Packet and Frame Format and Management

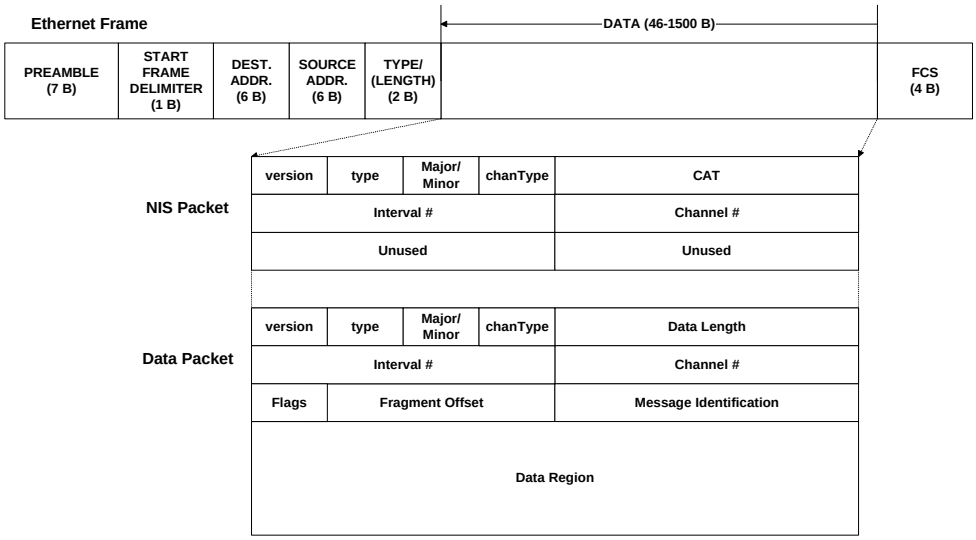


Figure 5-12 SPRETHER Packet Format

In addition to non-real-time messages, which are transmitted with Ethernet packets, SPRETHER adds two types of packets; control and real-time data packets. A control packet is used to synchronize the transmission and reception, and to implement other control functions. A real-time data packet is used to carry either SMC or MMC real-time messages. We show the format of a packet in Figure 5-12. A SPRETHER packet is encapsulated in the data area of an Ethernet frame, and distinguished by the TYPE word of the Ethernet frame header from other packets such as IP. The description of fields is shown in Table 5-3.

Table 5-3 The SPRETHER Packet Field Description

Field type	Description	Field type	Description
version	version number of a SPRETHER packet.	*Interval#*	serial number of current time interval in a cyclic schedule table.
type	classification of SPRETHER packet.	*Channel#*	identification of channel.
CAT	channel allocation time in microseconds.	*Flags*	flag which identifies the final fragmented packet.
Data Length	total length of real-time data.	*Fragment Offset*	offset field for fragmentation.
Major/Minor	flags for indication of Major and Minor frame.	*Message Identification*	identifier of the message stream.

5.3.6 SPRETHER API

We provide a simple and easy-to-use API for real-time messages in SPRETHER. For non-real-time messages, socket interfaces can be applied.

int sp_open (channelNumber, mode) : This creates a SPRETHER service point and registers the service with the SPRETHER module. The first parameter, channelNumber, designates the pre-scheduled channel number. The second parameter, mode, can be used to inform the SPRETHER module of the mode of operation, RD_ONLY, WR_ONLY, and RD_WR. It returns a corresponding identifier, *spid*, with which a task can send and receive real-time messages.

int sp_close (spid) : This removes the real-time session itself from the SPRETHER module.

int sp_send (spid, flag, smsg, length): This requests a transmission of the message stored in the buffer **smsg** of specified **length**. According to the value of the flag, the sender can either be blocked until the message is transmitted, or continued.

int sp_receive (spid, flag, rmsg, length): This requests to receive the next available message in the buffer **rmsg**. If the flag is block_mode_call, it waits until the next message arrives.

5.4 Scheduling Real-Time Messages

In this section, we present scheduling algorithms for real-time messages of SPRETHER. The scheduling approach first finds the necessary capacity for each real-time channel, and then obtains a distance-constrained cyclic schedule for all channels. The distance-constrained cyclic schedule is used to build RX and TX tables of each node for the TDPA protocol.

5.4.1 Scheduling Requirement of the SMC and MMC Channels

We first find the necessary capacity for the SMC channel server. Since the SMC channel server serves only one message stream, it is quite straightforward to find the necessary capacity. A message stream of the SMC is associated with three parameters; message size, $M_{k,}$ period, $T_{k,,}$ and deadline D_k . Then the capacity assignment, which will be used in cyclic scheduling in the following subsection, requires finding the channel cycle, b_k , and the channel capacity, m_k . The cyclic scheduler allocates b_k percentage of total SPRETHER bandwidth at every m_k period. The following formula is used to obtain the channel cycle and capacity for each SMC channel server.

$$b_k = \frac{M_k + SMO}{Min(T_k, D_k)}, m_k = Min(T_k, D_k)$$

The overheads, such as *SGap1*, *S(NIS)*, and *SGap2* are excluded from the above formula because they are dependent on cyclic scheduling. So they are included in the schedulability criteria for cyclic scheduling.

Then we find the necessary capacity for the MMC channel server. Since the MMC channel server schedules local messages in a fixed-priority driven method, the logical architecture of a channel server can be depicted as in Figure 5-13. The channel server schedules the current set of arrived messages and puts the highest priority message into the top of the buffer. One of the important design criteria is to select the fixed buffer size, MSIZE in Figure 5-13. In the case of ARINC 659, the resolution of MSIZE supported by the underlying hardware is a word (32bits). However, it is not practical to choose a word for MSIZE in SPRETHER, because it increases overheads. So, with the scheduling analysis of message (channel) requirements, we must decide

on a practically feasible MSIZE. It is an important job of the system integrator to find the optimal MSIZE.

Suppose that there are p message streams, MS_i, in a MMC channel server, S_k listed in priority order $MS_1 < MS_2 < ... < MS_p$ where MS_1 has the highest priority and MS_p has the lowest. In order to evaluate the schedulability of the MMC channel server S_k, let's consider the case that S_k is executed at a dedicated bandwidth of b_k, normalized with respect to the bandwidth of S_k.

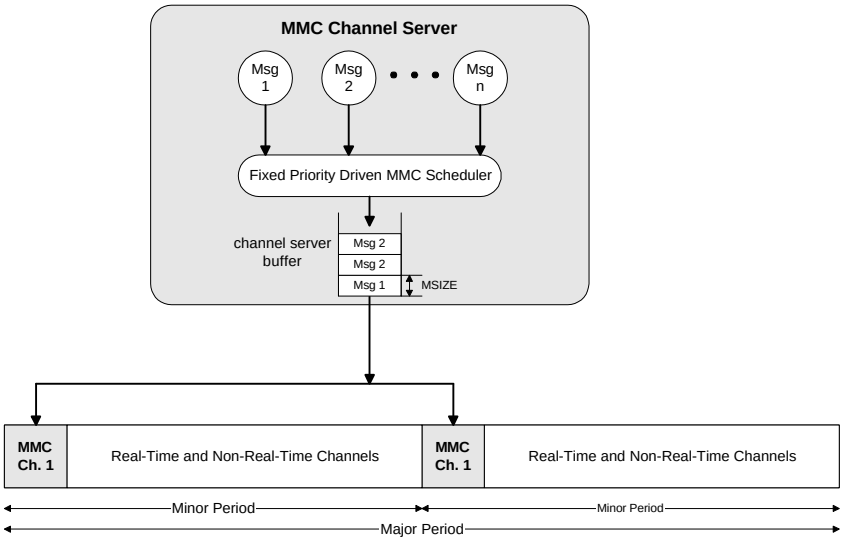


Figure 5-13 Architecture of MMC Channel Server

Based on the necessary and sufficient condition of schedulability in [40][41], message stream MS_i is schedulable if there exists a $t \in H_i = \{ lT_j \mid j = 1,2,...i; l = 1,2,..., \lfloor D_i / T_j \rfloor \} \cup \{D_i\}$, such that

$$W_i(b_k, t) = \sum_{j=1}^{i-1} \left(\frac{\left(MGap3 + MMO + M_j\right)}{b_k} \cdot \left\lceil \frac{t}{T_j} \right\rceil \right) + \frac{\left(MGap3 + MMO + M_i\right)}{b_k} + MSIZE \le t \ .$$

The expression $W_i(b_k, t)$ indicates the worst cumulative message demand, including overheads on the SPRETHER made by the message streams with a priority higher than or equal to MS_i during the interval [*0*,*t*]. We now define $B_i(b_k) = \max_{t \in H_i} \{t - W_i(b_k, t)\}$ and $B_0(b_k) = \min_{i=1,2,..,p} B_i(b_k)$. Note that, when MS_i is schedulable, $B_i(b_k)$ represent the total

period in the interval [0, D_i] that the SPRETHER is without any message streams with a priority higher than or equal to that of MS_i. It is equivalent to the level-*i* inactivity period in the interval [0, D_i] [8]. By comparing the message transmission at server S_k and at a dedicated bandwidth of b_k, we can obtain the following theorem:

Lemma 2. The set of message streams, A_k , is schedulable at server S_k that is with a channel cycle m_k and a channel capacity b_k, if

a) A_k is schedulable at a dedicated bandwidth utilization of b_k, and

b) $m_k \le B_0(b_k)/(1-b_k)$

Proof: The message transmission at server S_k can be modeled by message streams MS_1, MS_2,..., MS_p of A_k and an extra message stream MS_0 that is invoked every period h_k and has message size $M_0=(1-b_k)m_k$. The extra message stream MS_0 is assigned with the highest priority and can preempt other message streams. We need to show that, given the two conditions, any message stream MS_i of A_k can meet its deadline even if there are preemptions caused by the invocations of message stream MS_0.

To simplify the proof, we introduce a new term, $U_i = M_i + MGap3 + MMO$. According to the schedulability analysis in [40,41], message stream MS_i is schedulable at server S_k if there is a $t \in H_i \cup G_i$, such that

$$\sum_{j=1}^{i} U_j \left\lceil \frac{t}{T_j} \right\rceil + U_0 \left\lceil \frac{t}{h_k} \right\rceil \le t .$$

$$\text{where } G_i = \{ lh_k \mid l = 1,2,\cdots,\lfloor D_i/h_k \rfloor \} .$$

If MS_i is schedulable on a bandwidth capacity of b_k, there exists a $t_i^* \in H_i$ such that $B_i(b_k) = t_i^* - W_i(b_k, t_i^*) \ge B_0(b_k) \ge 0$ for all *i=1,2,...,p*. Note that $W_i(b_k, t_i)$ is a non-decreasing function of t_i. Assume that $t_i^*=mm_k+d$, where $d < m_k$. If $d \ge B_0(b_k)$,

$$\begin{aligned}\sum_{j=1}^{i} U_j \left\lceil \frac{t_i^*}{T_j} \right\rceil + U_0 \left\lceil \frac{t_i^*}{m_k} \right\rceil &= b_k W_i(b_k, t_i^*) + (m+1)U_0 \\ &\le b_k \left(t_i^* - B_0(b_k)\right) + (m+1)U_0 \\ &= b_k \left(t_i^* - B_0(b_k)\right) + (m+1)(1-b_k)m_k \\ &\le b_k \left(t_i^* - B_0(b_k)\right) + (1-b_k)(t_i^* - d) + B_0(b_k) \\ &= t_i^* + (1-b_k)(B_0(b_k) - d) \\ &\le t_i^*\end{aligned}$$

The above inequality implies that all message streams MS_i are schedulable at server S_k. On the other hand, if $d < B_0(b_k)$, then, at $t_i' = mm_k < t_i^*$, we have

$$\begin{aligned}\sum_{j=1}^{i} U_j \left\lceil \frac{t_i'}{T_j} \right\rceil + U_0 \left\lceil \frac{t_i'}{m_k} \right\rceil &\le b_k \left(t_i^* - B_0(b_k)\right) + mU_0 \\ &\le b_k \left(t_i^* - d\right) + m(1-b_k)m_k \\ &= b_k t_i' + (1-b_k)t_i' \\ &= t_i'\end{aligned}$$

Since $t_i' \in G_i$, the set of message streams, A_k is schedulable at server S_k.

When we compare the transmission sequences at server S_k and at the dedicated bandwidth, we can observe that, at the end of each channel cycle, S_k has put the same amount of bandwidth capacity to transmit the message streams as the dedicated network. However, if the message streams are transmitted at the dedicated network, they are not blocked and can be completed earlier within each channel cycle. Thus, we need an additional constraint to bound the delay of message transmission at server S_k. This bound is set by the second condition of the Theorem and is equal to the minimum inactivity period before each message's deadline. ■

5.4.2 Distance-Constrained Cyclic Scheduling

By the schedulability requirement analysis presented in the previous subsection, we can obtain a feasible set of n channel servers, which have a pair of channel capacity and cycle $(b_1, m_1), (b_2, m_2), \ldots, (b_n, m_n)$ and the set be sorted in the non-decreasing order of m_k. The set cannot be

directly used in a cyclic schedule that guarantees the distance constraint of assigning b_k bandwidth capacity for every m_k period in a channel. To satisfy the distance constraint between any two consecutive transmissions, we can adopt the pinwheel scheduling approach and transfer $\{m_k\}$ into a harmonic set through a specialization operation [42]. Note that, in [42], a fixed message size is allocated to each channel and would not be reduced even if we transmit message more frequently. This can lead to a lower utilization after the specialization operations. In our channel-scheduling solution, we allocate a certain percentage of bandwidth capacity to each channel. When the set of channel cycles $\{m_k\}$ is transformed into a harmonic set $\{m_k\}$, this percentage doesn't change. Thus, we can schedule any feasible sets of (b_k, m_k) as long as the total sum of b_k is less than 1.

Given a base channel cycle m, the algorithm finds a m_i for each m_i that satisfies:

$$m_i = \mu * 2^j \le \mu_i < \mu * 2^{j+1} = 2*m_i,$$

To find the optimal base m in the sense of bandwidth utilization, we can test all candidates m in the range of $(m_1/2, m_1]$ and compute the total capacity $\sum_k b_k^h$. To obtain the total capacity, the set of m_k is transferred to the set of m_k based on corresponding m and then the least capacity requirement, b_k^h, for channel cycle m_k is obtained from Lemma 2 for the MMC channel server. In the case of the SMC channel server, b_k^h is the same as b_k. With the set of capacity assignments we have found $\{(b_1^h, m_1),(b_2^h, m_2),...,(b_n^h, m_n)\}$, we can build a distance-constrained cyclic schedule. In Figure 5-14, we show an example of a cyclic schedule that guarantees distance constraints for the set of channel capacity requirements, *A(0.1,12), B(0.2,14), C(0.1,21), D(0.2,25), E(0.1,48),* and *F(0.3,50)*. We use the optimal base of 10 to convert the partition cycles to *10, 10, 20, 20, 40,* and *40*, respectively.

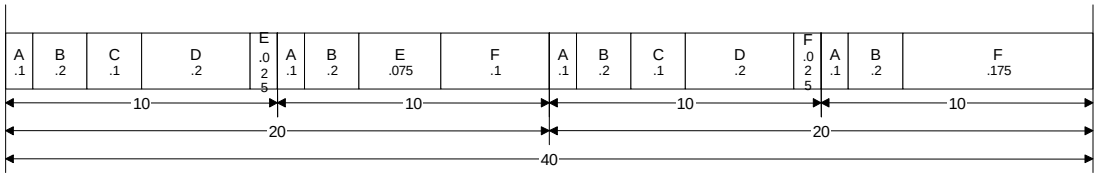


Figure 5-14 Example of a Communication Frame Cyclic Schedule

The optimal m is selected in order to minimize the total capacity, as shown in the left part of the equation below. The exact schedulability criterion is as follows.

$$Total\ Capacity = \sum_{i=1}^{n} \frac{n_i * (Gap1 + S(NIS) + Gap2) + \left| b_i^h * m_i \right|}{m_i} \leq 1$$

n_i = the number of fragmentations in the channel cycle,

Gap1 = *SGap1* (SMC) or *MGap1* (MMC)

Gap2 = SGap2 (SMC) or *MGap2* (MMC)

In the distance-constrained cyclic schedule, the channel capacity can be allocated in multiple time intervals within a channel cycle. For an example from Figure 5-14, channel F is allocated two time intervals, with 0.1 and 0.175 of the full bandwidth respectively. So the n_i, which corresponds to channel F, is 2.

5.5 Prototype and Performance Evaluation

We have prototyped SPRETHER on a network of Motorola MBX860 PowerPC-based embedded controllers with the HardHat real-time Linux operating system, as shown in Figure 5-15. The MBX860 equips a 40MHz PowerPC core and 36 MB DRAM memory. 20MB of DRAM is allocated for the ram-disk based root file system. We use Red Hat Linux as a cross-development environment. Real-time Linux on the MBX860 is booted with TFTP. After booting, the cross-development environment is disconnected from the Ethernet network, except the passive Ethereal network protocol analyzer. The experimental applications on real-time Linux are initiated and monitored by a serial console attached to the target MBX860 board.

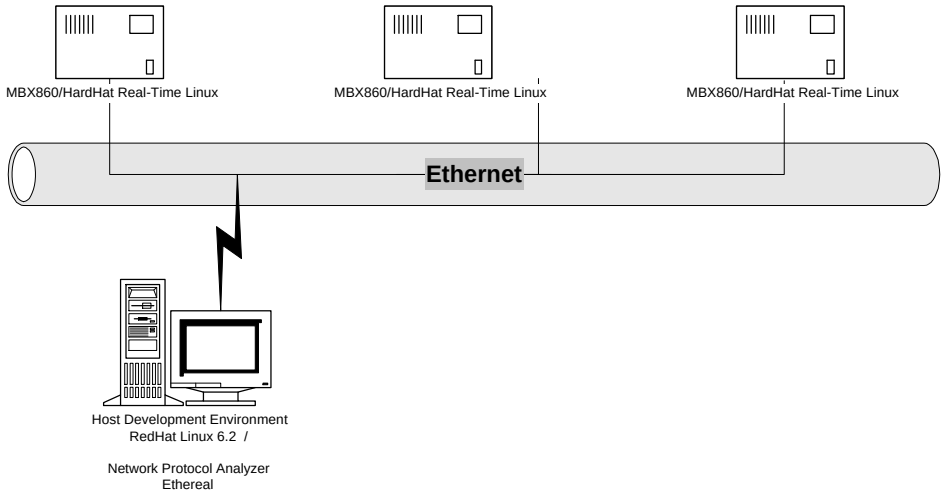


Figure 5-15 SPRETHER Prototype Platform

We show the software architecture of SPRETHER based on a Linux platform in Figure 5-16.

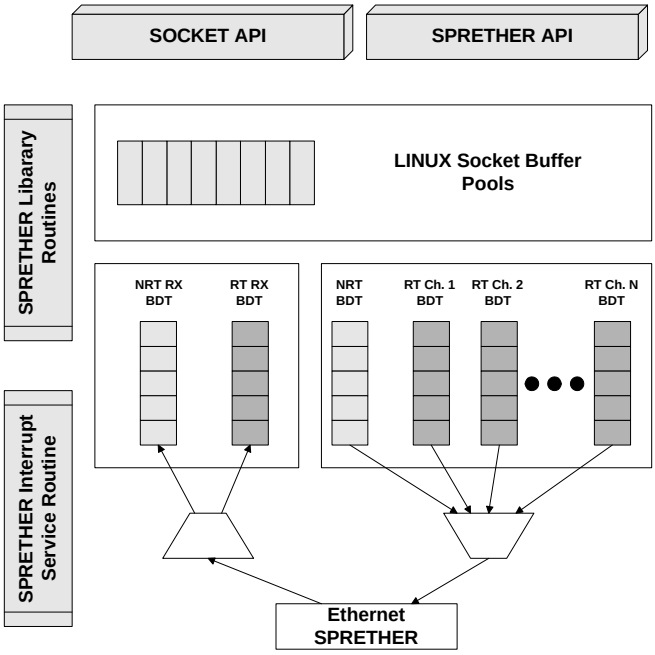


Figure 5-16 Software Architecture of SPRETHER

The transmitting and receiving buffers are allocated in the Linux socket buffer pool in the MBX860 memory. Only the buffer descriptors are stored in 5K DPRAM of the MBX860. Since a

buffer descriptor needs 8 bytes, there can be 640 buffer descriptors in 5K DPRAM. It is necessary to allocate a feasible number of buffer descriptors for real-time and non-real-time channels. The SPRETHER module supports both socket API for TCP/IP applications and SPRETHER API for real-time applications.

5.5.1 TDPA Protocol Overhead

Table 5-4 presents time overheads taken to prepare, send, and process a NIS packet, which is essential to implement synchronization for a software based TDPA protocol. The measured data were used in Section 3 to calculate the utilizations of SMC and MMC. All synchronization procedures have been implemented within the SPRETHER Ethernet driver of Hard-Hat Real-Time Linux. The synchronization overhead is composed of the NIS packet preparation time by the master node, packet transmission delay, and NIS packet processing time at the slave nodes. The overall overhead to process a NIS packet is approximately 219.04 μs. Both NIS packet preparation and processing time were decomposed into the Linux overhead and the SPRETHER overhead in the table.

Table 5-4 Synchronization (NIS Packet Processing) Overhead

Measurement		Time
NIS packet preparation time (after timer interrupt)		99.2 μs
	HardHat Linux interrupt latency for Timer	17.6 μs
	SPRETHER NIS packet preparation time	81.6 μs
NIS packet transmission time		86 μs
NIS packet processing time		33.84 μs
	HardHat Linux interrupt latency for Ethernet	16.8 μs
	SPRETHER NIS packet handling time	17.04 μs
Total (Sum)		219.04 μs

5.5.2 Response Time of Non-Real-Time ICMP Message

The response time of non-real-time ICMP messages is tightly related to the physical allocation scheme of the cyclic schedule and the reserved capacity of non-real-time communications. In this experiment, we varied the reserved capacity of non-real-time communications from 10% to 90% and measured the response times of ICMP messages. The reserved capacity of non-real-time communications was evenly distributed in the distance-constrained cyclic schedule. Figure 5-17 presents the experimental result of response times of non-real-time ICMP packets (echo request and echo reply) that were composed of 64 Bytes, including the header and frame check sequence. In this experiment, there is no collision between packets. The best response time of a packet is the same for all the reserved capacities as 1.4ms. The reason is that the best response time is obtained when the ping program sends out an echo request message at the time instance that falls on the NRTC interval. This situation can occur for all cases, 10% to 90% NTRC reserved capacities. The graph shows that the average and worst response times decrease linearly as the reserved capacity increases.

5.5.3 Throughput of Non-Real-Time Connection

To evaluate the utilization of the NRTC channel, we measured throughput of the non-real-time connection. Throughput is defined as the total number of bytes transmitted within a second. We use the TCP/IP ftp application, which sends a 350 KBytes sample binary file, to measure the throughput. In this experiment, we also varied the reserved capacity of non-real-time communications from 10% to 90%. The reserved capacity of non-real-time communications was evenly distributed in the distance-constrained cyclic schedule. There is no collision for a packet. Note that the files sent from the sender and received by the receiver are stored and being stored in local DRAM memory, because the root file system is implemented in the ram disk of Linux. Figure 5-18 shows the experimental result of throughput of ftp transmission. The graph shows that throughput increases in proportion to the reserved capacity.

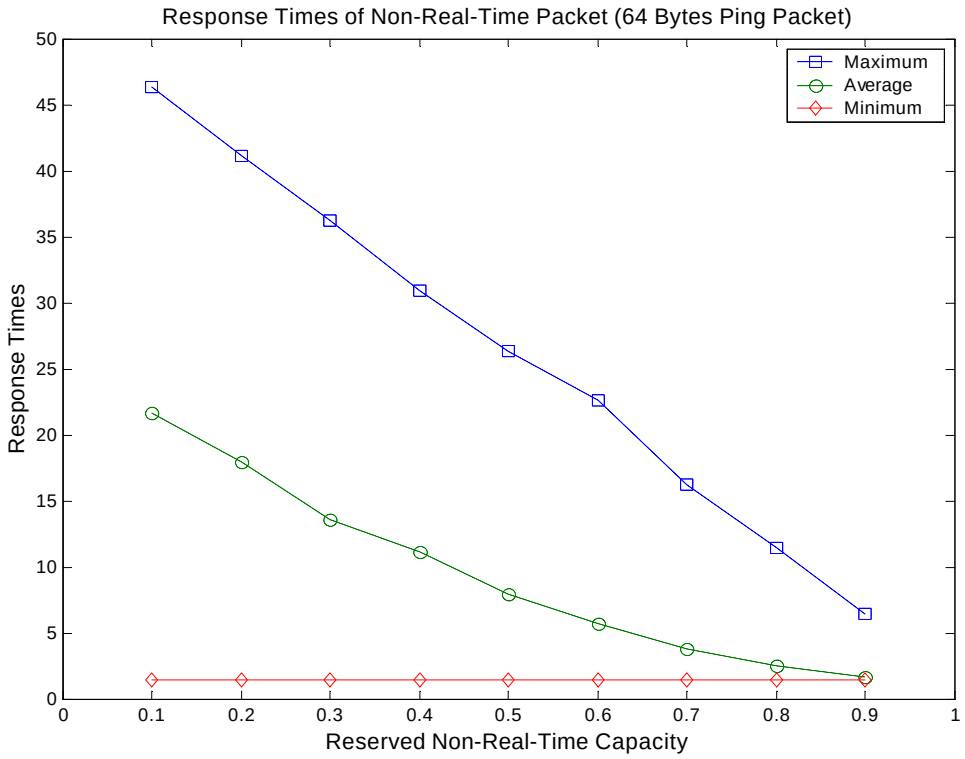


Figure 5-17 Response Time of Non-Real-Time Packet (64Bytes ICMP Packet)

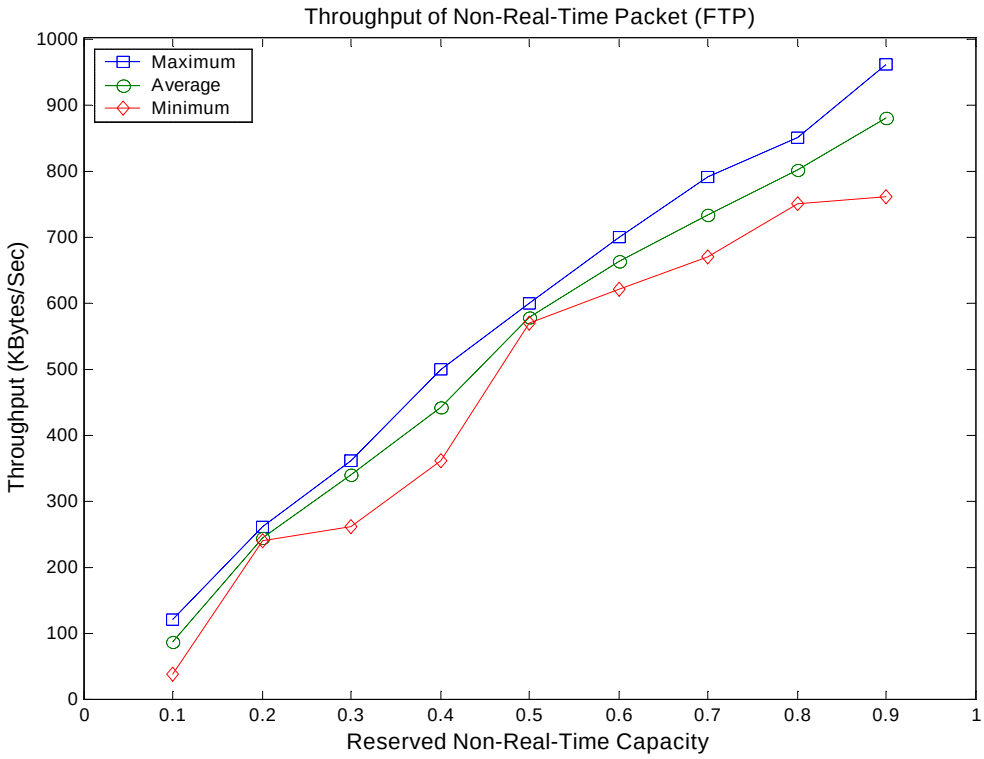


Figure 5-18 Throughput of Non-Real-Time Packet (FTP)

5.6 Conclusion

We proposed a real-time Ethernet, SPRETHER, for safety-critical real-time systems. Since the SPRETHER is implemented with a TDPA-based cyclic scheduling protocol, it can be used in hard real-time systems such as integrated modular avionics. Non-real-time communications can also be integrated as SPRETHER allocates a CSMA/CD Ethernet channel embedded in the cyclic schedule. For further study, we are investigating a multi-segments Ethernet configuration to solve scalability issue and dynamic real-time connection management. In addition to replicating SPRETHERs for fault-tolerance, we plan to look into the possible integration of redundancy management and channel scheduling to achieve the maximum performance.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Contributions

The objective of the dissertation is to design integrated real-time systems supporting a strong partitioning scheme. The main contributions of the dissertation are as follows:

A strongly partitioned integrated real-time systems model. We propose a system model for integrated real-time systems, which can be used for safety-critical real-time systems such as integrated modular avionics systems. The model is concerned with (1) integration of real-time applications with different criticalities, (2) integrated scheduling of processor and message communication, and (3) guaranteeing a strong partitioning concept.

Comprehensive scheduling approaches to integrated real-time systems. We provide comprehensive scheduling algorithms for integrated real-time systems. They include (1) a fundamental two-level scheduling theory, (2) integrated scheduling of partitions and channels, (3) soft and hard aperiodic task scheduling, (4) practical constraints solving algorithms, and (5) a full scheduling tool suite, which provides an automated scheduling analysis.

A real-time kernel for integrated real-time systems. We develop a software platform for integrated real-time systems by means of the SPIRIT-μKernel. The kernel implements (1) strong partitioning concepts with a cyclic scheduler and a strong resource protection mechanism, and (2) a generic real-time operating system port interface. (3) The kernel makes it possible to run real-time applications developed in different real-time operating systems on the same processor, while guaranteeing strong partitioning.

A real-time Ethernet for integrated real-time systems. We provide a real-time Ethernet for integrated real-time systems by means of SPRETHER. The SPRETHER includes (1)

a table driven proportional access protocol for implementing distance-constrained cyclic scheduling, (2) support for real-time messages, (3) a scheduling algorithm to meet real-time requirements, and (4) to support non-real-time messages by integrating the original CSMA/CD MAC operation into the TDPA protocol.

6.2 Future Research Directions

Since many integrated real-time systems are safety-critical and comprehensive, the design of such systems usually involves many aspects of computer and software engineering fields. Considering these aspects, we describe future research directions, which should be extended or developed with the dissertation as a starting point, as follows:

Enhancement of the SPIRIT-mKernel. We are planning to enhance the kernel in three directions. First, we will build our own local kernel for a partition instead of using COTS RTOS. It will make a full suite of two-level operating systems platforms. Second, we will extend the current uni-processor architecture to the multiprocessor and distributed real-time computing environment. Third, we will develop additional two-layer scheduling strategies instead of the cyclic/priority driven scheduler pair, while guaranteeing the strong partitioning requirement for different application environment.

Scalability and fault tolerance issues of SPRETHER. We are investigating multi-segments Ethernet configuration to solve scalability issues and dynamic real-time connection management. In addition to replicating SPRETHERs for fault-tolerance, we plan to look into the possible integration of redundancy management and channel scheduling to achieve maximum performance.

Real-time software testing and verification. Testing and verification of real-time software is a challenging research topic, considering the correctness of real-time software is not only dependent on functional correctness but also timing correctness. A test coverage analysis is a software testing method with which testers can measure how thoroughly a set of tests has exercised a program. During a test coverage analysis, the program is instrumented so that it will

produce an execution trace when it is run. Then, a testing tool analyzes the traces to determine which program components the tests have covered. Usually, a test coverage analysis produces measures such as basic-block, decision, and dataflow coverages.

A test coverage analysis of real-time software is a difficult task, because the probe effect caused by program instrumentation may introduce a different execution behavior of real-time software from its original one. In real-time software testing, this non-deterministic behavior is not tolerable because the correctness of real-time systems is dependent not only on functional behavior but also timing behavior. However, heavy program instrumentation for test coverage analysis cannot be avoided in obtaining the necessary information for correct coverage analysis. In the further study, we will investigate feasible solutions to perform a test coverage analysis for real-time applications in integrated real-time systems.

LIST OF REFERENCES

[1] "Design Guide for Integrated Modular Avionics," ARINC Report 651, Aeronautical Radio Inc., Annapolis, MD, Nov. 1991.

[2] "Avionics Application Software Standard Interface," ARINC Report 653, Aeronautical Radio Inc., Annapolis, MD, Jan. 1997.

[3] "Backplane Data Bus," ARINC Specification 659, Aeronautical Radio Inc., Annapolis, MD, Dec. 1993.

[4] O. Serlin. "Scheduling of Time Critical Processes," *Proc. of the Spring Joint Computer Conference*, pp. 925-932, May, 1972.

[5] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, no. 1, pp.46-61, 1973.

[6] B. Sprunt, L. Sha and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems", *Journal of Real-Time Systems* , vol. 1, pp. 27-60, 1989.

[7] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The Deferrable Server Algorithms for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Trans. on Computers*, vol. 44, no.1, pp. 73-91, Jan. 1995.

[8] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," *Proc. of IEEE Real-Time Systems Symposium*, pp. 110-123, Dec. 1992.

[9] M. Spuri, G. Buttazzo, "Efficient Aperiodic Service Under Earliest Deadline Scheduling," *Proc. of IEEE Real-Time Systems Symposium*, pp. 2-11, Dec. 1994.

[10] C. Shen, K. Ramamritham, and J. A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no.4, pp. 382-397, Apr. 1993.

[11] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems," *Proc. of IEEE Real-Time Systems Symposium*, pp. 152-161, Dec. 1995.

[12] J. Liu, "Real-Time Systems," Book Manuscript, Chap. 5, 1998.

[13] Z. Deng and J. W. S. Liu, "Scheduling real-time applications in an open environment," *Proc. of IEEE Real-Time Systems Symposium*, pp. 308-319, Dec. 1997.

[14] N. Audsley, and A. Wellings, "Analyzing APEX applications," *Proc. of IEEE Real-Time Systems Symposium*, pp. 39-44, Dec. 1996.

[15] A. J. Mullender, The Amoeba distributed operating system: Selected papers 1984-1987. Tech. Rep. Tract. 41, CWI, Amsterdam, 1987.

[16] R. Campbell, N. Islam, P. Madany, and D. Raila, "Designing and implementing Choices: An object-oriented system in C++," *Communications of ACM*, vol. 36, no. 9, pp. 117-126, Sep. 1993.

[17] J. M. Bernabeu-Auban, P.W. Hutto, and Y.A. Khalidi, "The architecture of the Ra kernel," *Tech. Rep. GIT-ICS-87/35*, Georgia Institute of Technology, Atlanta, 1988.

[18] D. R. Cheriton, G. R. Whitehead, and E. W. Sznyter, "Binary emulation of Unix using the V kernel, " *Proc. of the Usenix Summer Conference* , pp. 73-86, Jun. 1990.

[19] M. Guillemont, "The Chorus distributed operating system: Design and implementation," *Proc. of the ACM International Symposium on Local Computer Networks*, pp. 207-223, Apr. 1982.

[20] J. Liedtke, "A persistent system in real use-experiences of the first 13 years," *Proc. of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pp. 2-11, Dec. 1993.

[21] D. Golub, R Dean, A. Forin, and R. Rashid, "Unix as an application program," *Proc. of the Usenix Summer Conference*, pp. 87-96, Jun. 1990.

[22] D. Engler, M. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. of ACM Special Interest Group on Operating Systems*, pp. 251-266, Dec. 1995.

[23] J. Liedtke, "On microkernel construction," *Proc. of the 15th ACM Symposium on Operating System Principles*, pp. 237-250, Dec. 1995.

[24] D. Mosse, A. Agrawala, and S. Tripathi, "Maruti a hard real-time operating system," *IEEE Workshop on Experimental Distributed Systems*, pp. 29-34, 1990.

[25] H. Kopetz, and et. al., "Distributed fault-tolerant real-time systems: The MARS Approach," *IEEE Micro*, vol. 9, no. 1, pp. 25-40, 1989.

[26] M. Johnson, "Boeing 777 airplane information management system – philosophy and displays," *Proc. of the Royal Aeronautical Society's Advanced Avionics Conference on Aq330/A340 and the Boeing 777 aircraft*, 1993.

[27] IBM, "IBM Virtual Machine Facility /370: Release 2 Planning Guide," *Technical Report GC20-1814-0*, IBM Corporation, 1973.

[28] E. P. Goldberg, "Survey of virtual machine research," *IEEE Computer*, pp. 34-45, 1974.

[29] T. C. Bressoud, F. P. Schneider, "Hypervisor-based fault-tolerance," *Proc. of the ACM Symposium on Operating Systems Principles*, pp. 1-11, 1995.

[30] J. Gosling, H. McGilton, "The Java language environment: A White paper," *Technical Report*, Sun Microsystems Computer Company, 1996.

[31] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety, and performance in the Spin operating system," *Proc. of the 15th ACM Symposium on Operating System Principles*, pp. 267-284, 1995.

[32] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels Meet Recursive Virtual Machines," *Proc. of USENIX Symposium on Operating Systems Design and Implementation*," Oct. 1996.

[33] C. Small, M. Seltzer, "Structuring the kernel as a toolkit of extensible, reusable components," *Proc. of the International Workshop on Object Orientation in Operating Systems*, pp. 134-137, 1995.

[34] J. Liedtke, "Toward Real Microkernels," *Communications of the ACM*, vol. 39, no. 9, Sep. 1996.

[35] C. Mercer, R. Rajkumar, J. Zelenka, "Temporal protection in real-time operating systems," *Proc. of the IEEE Workshop on Real-Time Operating Systems and Software*, pp.79-83, 1994.

[36] TI, "Architecture Document for the Combat Vehicle Operating Environment," *Technical report*, Texas Instruments, Defense Systems & Electronics Group, 1997.

[37] A. R. Albrecht and P. A. Thaler., "Introduction to 100VG-AnyLAN and the IEEE 802.12 local area network standard, " *Hewlett-Packard Journal*, vol. 46, no. 4, Aug. 1995.

[38] X. Ran and W. R. Friedrich. , "Isochronous LAN based full-motion video and image server-client system with constant distortion adaptive DCT coding," *Proc. of the SPIE – The International Society for Optical*, 2094:1030:41, 1993.

[39] The 3COM Technical Journal. 3Com's New PACE Technology. http://www.3com.com/_les/mktg/pubs/3tech/195pace.html, 1996.

[40] J. Lehoczky, "Fixed-priority scheduling for periodic task sets with arbitrary deadlines," *Proc. of IEEE Real-time Systems Symposium*, pp. 201-209, Dec. 1990.

[41] J. Lehoczky, L. Sha, and Y.Ding, "The rate-monotonic scheduling algorithm: exact characteristics and average case behavior," *Proc. of IEEE Real-Time Systems Symposium*, pp. 166-171, Dec. 1989.

[42] C.-C. Han, K.-J. Lin, and C.-J. Hou, "Distance-constrained scheduling and its applications to real-time systems," *IEEE Trans. on Computers*, vol. 45, no. 7, pp. 814-826, Jul. 1996.

[43] M. Y. Chan and F. Y. L. Chin, "General schedulers for the pinwheel problem based on double-integer reduction," *IEEE Trans. on Computers*, vol. 41, pp. 755-768, Jun. 1992.

[44] K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133-151, 1994.

[45] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, "A new look at microkernel-based Unix operating systems," *Tech. Rep. CS/TR-91-7*, Chorus systèmes, Paris, France, 1991.

[46] A. Lindstroem, J. Rosenberg, and A. Dearle, "The grand unified theory of address spaces, " *Proc. of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[47] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang. , "The Tenet Real-Time Protocol Suite: Design, Implementation and Experiences, " *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pp. 1-10, Feb. 1996.

[48] C. C. Chou and K. G. Shin., "Statistical real-time channels on multiaccess networks, " *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 769-780, Aug. 1997.

[49] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis kernel, " *Comput. Syst.* vol. 1, no. 1, pp. 11-32, Jan. 1988.

[50] C. Topolcic. Experimental internet stream protocol, Version 2 (ST-II). Internet RFC:1190, Oct. 1990.

[51] C. Venkatramani and T. Chiueh., "Design, implementation and evaluation of a software-based real-time Ethernet protocol," *Proc. of ACM SIGCOMM*, pp. 27-37, Aug. 1995.

[52] C. Venkatramani and T. Chiueh., "Supporting real-time traffic on Ethernet," *Proc. of Real-Time Systems Symposium*, pp. 282-286, Dec. 1994.

[53] C. H. Lee, M. C. Chen, and R. C. Chang, "HiPEC: High performance external virtual memory caching," *Proc. of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 153-164, Nov. 1994.

[54] C. Venkatramani. The Design, Implementation and Evaluation of RETHER : A Real-Time Ethernet Protocol. PhD thesis, State University of New York at Stony Brook, 1996.

[55] D. Ferrari and D. Verma. , "A scheme for real-time channel establishment in wide area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368-379, Apr. 1990.

[56] D. W. Pritty, J. R. Malone, S. K. Banerjee, and N. L. Lawrie., "A real-time upgrade for Ethernet based factory networking," *Proc. of IECON*, pp. 1631-1637, 1995.

[57] D. Kim and Y.-H. Lee, "DC^2 Scheduling of Aperiodic Tasks for Strongly Partitioned Real Time Systems," *IEEE Real Time Computing Systems and Applications*, pp. 368-375, Dec. 2000.

[58] D. Kim, Y.-H. Lee, and M. Younis, "SPIRIT-μKernel for Strongly Partitioned Real Time Systems," *IEEE Real Time Computing Systems and Applications*, pp. 73-80, Dec. 2000.

[59] D. Kim, Y. Doh, and Y.-H. Lee, "Java Real-Time Publish-Subscribe Middleware for Distributed Embedded Systems," Book Chapter, Architecture and Design of Distributed Embedded Systems, pp. 193-203, Oct. 2001.

[60] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, J. Wolter, "The Performance of μ-Kernel-Based Systems," *Proc. of ACM Symposium on Operating Systems Principles*, pp. 66-77, Oct. 1997.

[61] H. Zhang and E. Knightly., "Providing end-to-end statistical guarantees using bounding interval dependent stochastic models," *Proc. of ACM SIGMETRICS*, pp. 211-220, 1994.

[62] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay, "Two years of experience with a microkernel based operating system, " Oper. Syst. Rev. vol. 25, no. 2, pp. 51-62, Apr. 1991.

[63] J. B. Chen, and B. N. Bershad, "The impact of operating system structure on memory system performance," *Proc. of the ACM Symposium on Operating System Principles*, pp. 120-133, Dec. 1993.

[64] L. Delgrossi, R. G. Herrtwich, and F. O. Homann., "An implementation of ST-II for the Heidelberg Transport System," *Internetworking: Research and Experience*, vol. 5, no. 2, pp. 43-69, Jun.1994.

[65] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala., "RSVP : A New Resource Reservation Protocol," *IEEE Network Magazine*, Sep. 1993.

[66] M. Condict, D. Bolinger, E. McManus, D. Mitchell, and S. Lewontin, "Microkernel modularity with integrated kernel performance," Tech. Rep., OSF Research Institute, Cambridge, Mass, 1994.

[67] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The duality of memory and communication in the implementation of a multiprocessor operating system," *Proc. of the ACM Symposium on Operating System Principles*, pp. 63-76, Nov. 1987.

[68] M. Younis, M. Aboutabl, and D. Kim, "An Approach for Supporting Software Partitioning and Reuse in Integrated Modular Avionics," *Proc. of IEEE Real-time Technology and Applications Symposium*, pp. 56-66, May 2000.

[69] M. Vernick., The design, Implementation and Evaluation of the Stony Brook Video Server. PhD thesis, State University of New York at Stony Brook, 1996.

[70] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard real-time scheduling: the deadline-monotonic approach," *IEEE Workshop on Real-time Operating Systems and Software*, pp. 133-137, 1991.

[71] P. Cao, E. W. Felten, and K. Li, "Implementation and performance of application-controlled file caching," *Proc. of the 1st Usenix Symposium on Operating Systems Design and Implementation*, pp. 165-178, Nov. 1994.

[72] P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. of USENIX Symposium on Operating Systems Design and Implementation*," Oct. 1996.

[73] R. Court. , “Real-time Ethernet,” *Computer Communications*, vol. 15, no. 3, pp. 198-201, Apr. 1992.

[74] R. Herrtwich and L. Delgrossi., “Beyond ST-II: Fulfilling the requirements of multimedia communication, “ *Network and Operating System support for digital audio and video*, Nov. 1992.

[75] R. Kessler, and M.D. Hill, “Page placement algorithms for large real-indexed caches,” ACM Trans. on Computer Systems, vol. 10, no. 4, pp. 11-22, Nov. 1992.

[76] R. L. Cruz., “A calculus for network delay, part I: network elements in isolation,” *IEEE Trans. on Information Theory*, vol. 37, no. 1, pp. 114-131, Jan. 1991.

[77] S. Kweon and K. G. Shin. Statistical real-time communication over ethernet/fast ethernet. Technical report, Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, 1999.

[78] S. L. Beuerman and E. Coyle, “The delay characteristics of CSMA/CD networks,” *IEEE Trans. on Communications*, vol. 36, no. 5, pp. 553-563, May 1988.

[79] T. Carpenter, “Avionics Integration for CNS/ATM,” *IEEE Computer*, pp. 124-126, Dec. 1998.

[80] T. Chiueh and C. Venkatramani. , ”Supporting Real-time Traffic on Ethernet,” *Proc. of IEEE Real-time Systems Symposium*, pp. 282-286, Dec. 1994.

[81] T. Chiueh, C. Venkatramani, and M. Vernick. ,”Design of the Stony Brook Video Server,” *SPIE First International Symposium on Technologies and Systems for Voice, Video and Data Communications*, Oct 1995.

[82] T. Vo-Dai. ,”Steady-state analysis of CSMA-CD,” *Performance*, pp. 243-265, 1984.

[83] T.H. Romer, D.L. Lee, B.N. Bershad, and B. Chen, “Dynamic page mapping policies for cache conflict resolution on standard hardware, “ *Proc. of the Usenix Symposium on Operating Systems Design and Implementation*, pp. 255-266, Nov. 1994.

[84] Y.-H. Lee, D. Kim, M. Younis, and J. Zhou, “Partition scheduling in APEX runtime environment for embedded avionics software,” *Proc. of Real-Time Computing Systems and Applications*, pp. 103-109, Oct. 1998.

[85] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy, “Resource Scheduling in Dependable Integrated Modular Avionics,” *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (FTCS-30/DCCA-8)*, pp. 14-23, Jun. 2000.

[86] Y. Shimokawa and Y. Shiobara., “Real-time Ethernet for industrial applications,” *Proc. of IECON*, pp. 829-834, 1985.

[87] Y.A. Khalidi, and M.N. Nelson, “Extensible file systems in Spring,” *Proc. of the ACM Symposium on Operating System Principles*, pp. 1-14, Dec. 1993.

[88] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, “Scheduling Tool and Algorithm for Integrated Modular Avionics Systems,” *IEEE/AIAA Digital Avionics System Conference,* Oct. 2000.

[89] Y.-H. Lee and D. Kim, “Partition and Message Scheduling in GPACS architecture,” *Final Research Report to Allied Signal Inc.*, Mar. 1999.

[90] Y.-H. Lee, D. Kim, “Development of Application Execution Environment in Partitioned Systems, “ *Final Research Report to Honeywell International*, Dec. 1999.

BIOGRAPHICAL SKETCH

Daeyoung Kim was born on January 27th, 1968, in Pusan, Republic of Korea. He received his Bachelor of Science degree in computer science and statistics from Pusan National University, Pusan, Republic of Korea, in February 1990. He then received his Master of Science degree in computer science from Pusan National University, Pusan, Republic of Korea, in February 1992. He has been a member of the research staff at the Electronics and Telecommunications Research Institute (ETRI), Republic of Korea, since 1992. He participated in the development of the ATM LAN/WAN switching system and led the embedded S/W and H/W group at ETRI. He is also a recipient of the ETRI scholarship for his doctoral study. He joined the Computer and Information Science and Engineering Department at the University of Florida, Gainesville, FL, in January 1998, to pursue a doctoral degree. His research interests include real-time systems (scheduling, communication, operating systems, and software testing), wireless and mobile networks, JAVA technology for real time applications, embedded system architecture, distributed and fault tolerant system, and QOS and fast routing in high-speed networks. During his doctoral study, he joined the Department of Computer Engineering at Arizona State University as a visiting research scholar in January 2001. He is a student member of the IEEE, the IEEE Computer Society, the ACM, and the AIAA.