SCHEDULABILITY ANALYSIS IN STATIC REAL-TIME SYSTEMS: PRIORITY MAPPING AND DASPCP FOR REAL-TIME CORBA

BY

RAMACHANDRA BETHMANGALKAR

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1999

MASTER OF SCIENCE THESIS

OF

RAMACHANDRA BETHMANGALKAR

APPROVED:

Thesis Committee

Major Professor ______________________________

____

____

____

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1999

Abstract

The goal of this work is to consider the effects of limited number of available priorities on schedulability analysis, implement an algorithm that maps the global priority of tasks to their run-time priorities, in PERTS, a real-time analysis tool from Tri-Pacific Software Inc.

PERTS assumes unlimited priorities and assigns unique priorities. But in reality, the number of priorities available on a system, operating system, network or backplane hardware is limited.

Since there are fewer priority levels supported by the operating system than the number of unique priorities, several of the tasks have to be mapped to the same local priority, introducing the possiblity of *priority inversion*: a higher priority task being blocked by a lower priority task. The tool should not only take this into effect but also do the mapping of unique task priorities possibly from an unlimited range to the limited range supported by that system.

We will also consider enhancement to PERTS to support RT CORBA, by implementing the *Distributed Affected Set Priority Ceiling Protocol (DASPCP)* as a control access protocol for resources in the system. DASPCP developed at University of Rhode Island, exploits the semantics of object oriented paradigm. Since DASPCP provides higher potential concurrency for distributed object-oriented systems than the existing Priority Ceiling protocols, it is an ideal resource control protocol for CORBA systems.

The proposed OMG Real-Time CORBA standard uses the notion of global, uniform priority assignment to threads of clients and servants in the CORBA system. RT CORBA also specifies a

Scheduling Service that uses RT CORBA primitives to facilitate enforcing various static priority real-time scheduling policies across the RT CORBA system. Hence it is highly desirable to have an augmented PERTS that not only analyzes RT CORBA systems but also generates output that the scheduling service can use in making scheduling decisions. We will also look into issues to make PERTS more suitable for analyzing Real-Time CORBA systems with emphasis on interfacing PERTS and RT CORBA.

Acknowledgments

First and foremost, I would like to thank my advisor Dr.Vic Fay-Wolfe for providing me an oppurtunity to work under his guidance. Vic has helped me in more than one way throughout my graduate study. The free access to excellent facilities in the real-time laboratory he extended has helped me gain considerable practical knowledge.

My sincere thanks to Dr.Ravi for his constant support and guidance. His enthusiasm and intelligence along with the overwhelming depth and breadth of his knowledge has been an inspiration to me. I hope to follow his systematic way of applying knowledge in my future endeavors.

I am grateful to Dr.Sury and Dr.Yang for consenting to serve on my committee.

I am indebted to Levon Esibov, Lisa DiPippo, Greg Cooper, Sean White and rest of the real-time research group for providing considerable inputs. I appreciate Levon's patience in answering my innumerable queries.

I am grateful to Mr.Ben Watson and Mr.Peter Kortmann of Tri-Pacific Software Inc. for giving me an oppurtunity to work with them. My sincere thanks to Ben for his constant advice, critique and encouragement.

I would like to thank Lorraine and Marge for making this department a wonderful place to study and work.

Thanks to my friends for their support and help, especially, Mithuna, Kartik, Ravi, Kanta, Sesha, Arjun, Soyal and Tina.

Finally, I would like to thank my family for their advice, support and patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last several years there has been a tremendous increase in the number of applications that can be classified under Real-time computing. In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

Prototyping Environment for Real-Time Systems (PERTS), developed at the University of Illinois (Urbana, IL) [3] and currently supported by Tri-pacific Software Inc., (Alameda, CA) [1] is an automated tool used to analyze static real-time systems. In static real-time systems all of the information about the system is known *a priori*. Chapter 2 briefly discusses scheduling theory and some of the features of PERTS.

Common Object Request Broker Architecture (CORBA) [2] is a widely accepted standard for distributed computing developed by the Object Management Group (OMG). There has been a great demand for Real-Time extensions to CORBA. It stimulated significant interest in industry and academia leading to the birth of Real-Time CORBA in November 1998. Real-Time CORBA is a recently developed standard by OMG that addresses real-time issues in distributed computing. RT CORBA is a product of decades of intense research in real-time systems and distributed systems, hitherto independent areas in Computer Science research. RT CORBA is described in greater detail in Chapter 3.

The possibility of preliminary schedulability analysis of the target system is of great value to the designer of the system. Unfortunately, PERTS does not fully support the analysis of RT CORBA systems. After a detailed analysis of the capabilities of PERTS, we have concluded that the present PERTS version may be modified to assist the designer better, in building real-time systems. These are discussed in Chapters 4 and 5.

In Chapter 6 we present the necessary modifications to PERTS and describe the Implementation.

The test cases that demonstrate the correctness of the modified PERTS components are presented in Chapter 7.

Closing remarks and future work are presented in Chapter 8.

Chapter 2

PERTS

Real-time computing systems play a vital role in our society, and they cover a spectrum from very simple to very complex applications. Examples of current real-time computing systems include control of laboratory experiments, control of automobile engines, command-and-control systems, nuclear power plants, process control plants, flight control systems, space shuttle and aircraft avionics and robotics. The more complicated real-time systems are expensive to build and their timing constraints are verified with ad hoc techniques. Minor changes in system result in another round of extensive testing. Different system components are extremely difficult to integrate and consequently add to overall system cost. Millions (even billions) of dollars are being spent (or even wasted) by industry and governments to build today's real-time systems. Hence availability of automated tools like PERTS is a great boon to the real-time systems designer.

In this chapter we review PERTS 3.2 regarding its abilities to describe and analyze real-time systems. For a complete description we refer the reader to the PERTS manual available on-line [4]. Here we concentrate only on those features that are relevant to this project.

PERTS (Prototyping Environment for Real-Time Systems) is a state-of-the-art modeling and prototyping tool for analyzing real-time architectures for scheduling and timing bottlenecks. It is a product of research done at the University of Illinois at Urbana-Champaign and Software

Engineering Institute (SEI), of the Carnegie Mellon University, Pittsburgh. PERTS uses Rate Monotonic Analysis (RMA) and other scheduling theories. It includes an extensible library of scheduling algorithms and resource access protocols.

In order to validate a real-time system's timing constraints and to evaluate its performance, the system parameters must be described. The input to a tool like PERTS, is a complete description of the system. A system is comprised of tasks, resources they use, their workloads, their timing contraints expressed as deadlines, dependencies if any among these tasks. The output is a *Yes* or *No* schedulability result for each task (and the whole system), priorities to be assigned to these tasks, priority ceilings for resources and suggestions, to make the system schedulable if it is found to be not schedulable. PERTS is organized into three different modules, the Task Graph Editor, the Resource Graph Editor and the Schedulability Analyzer. Each of these modules is described in the following sections.

2.1 Task Graph Editor

A Task Graph describes the application system, called the *task system.* It includes the set of tasks in the system that is being modeled. A task could be *periodic*, when it is scheduled to run after every T (a fixed interval of time) time units, or *aperiodic*, otherwise. Since aperiodic tasks is beyond the scope of our study, we will exclude their consideration in this project. A task in the system may be *dependent* on other tasks in the same system. Two tasks are said to be dependent when the completion of one is dependent on a result of the execution of the other task.

The collection of all tasks and their description is called a *Task Graph.* Every task and dependency is characterized by a set of parameters. To describe a task graph, the user must provide a complete set of parameters for every task and dependency in the task graph. PERTS Task Graph Editor is a user-friendly interface to describe the tasks in the system.

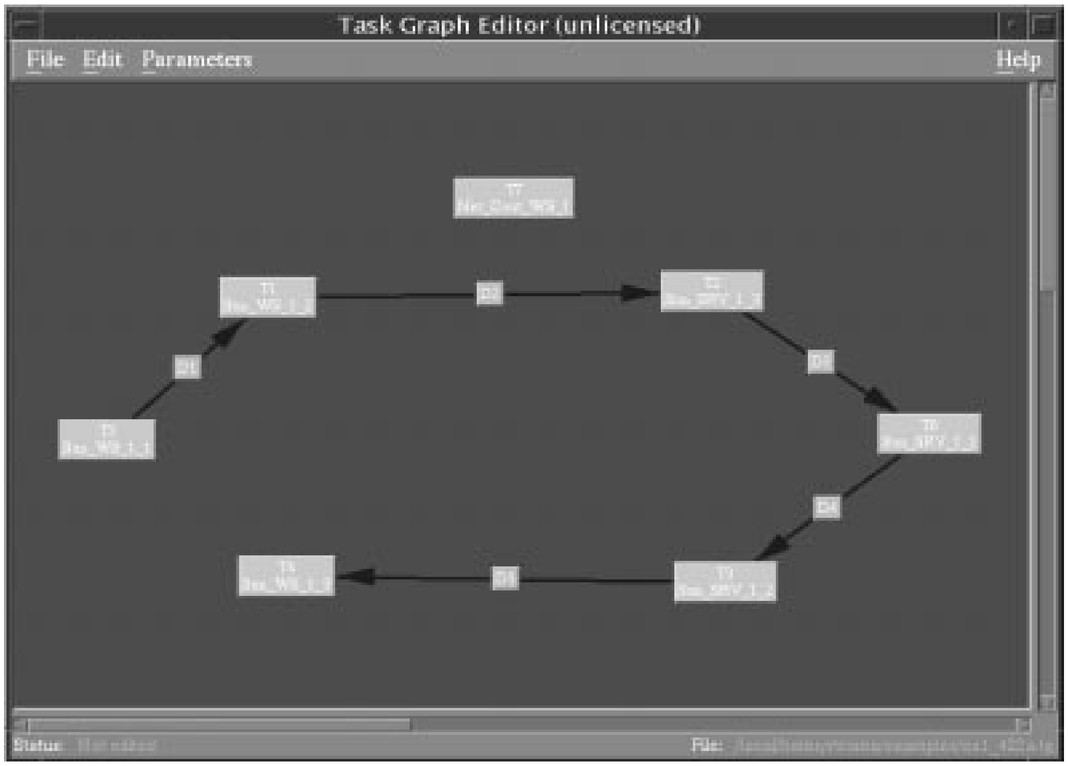


Figure 2.1: Snapshot of PERTS Task Graph Editor

The Task Graph Editor, shown in Figure 2.1, enables the user to create and update a task graph. It provides a graphical representation of the tasks in the system. All the tasks of a task graph are represented by rectangular nodes, all dependencies are presented by directed edges connecting the appropriate nodes. A task graph is described by choosing an appropriate operation in a menu bar and clicking on the appropriate node or edge. To describe the set of available in Editor operations/commands, we present them in groups, as they are arranged in menu bar.

File Commands enable the user to create a new task graph (*New*), open an existing task graph (*Open*), re-initialize already open task graph (*Reopen*), save current task graph (*Save*), save a new copy of current task graph (*Save As*), print current window (*Print Entire Window*), create a report of task graph information (*Generate Report*), launch any of the other PERTS tools or exit the task graph editor (*Quit*).

Edit Commands enable the user to manipulate task nodes and task dependencies. A user can add task node (*Add Task*), add dependency edge (*Add Dependency*), copy task characteristics (*Copy Task Parameters*), move task nodes to the new position on a screen (*Move Task*), delete task nodes or dependencies (*Delete*) or undo an unintentional edit command (*Undo (Add or Delete)*).

Parameter Commands enable the user to enter and change task parameters for each task in the task graph. Since we are interested in the periodic tasks only, we describe here only menu (and operations) for the periodic task parameters. It includes the options to enter and edit the *General Task Data, Optional Intervals, Non-Preemptable Sections, Resource Requirements* and *User Specified Priorities.*

When the user clicks on the General Task Data menu bar, an edit dialog window pops up, which enables input and edit of general task information for any task in the open task graph. General Task Data include the following parameters (we omit here some of the parameters irrelevant to our study):

- Task Name,
- Ready Time - the earliest time instant at which the task may begin execution,
- Relative Deadline - time frame after ready time within which the task must finish execution (reader can find in the literature a term absolute deadline, which is a sum of ready time and relative deadline),
- Period - constant length of time between two consecutive ready times of the task,
- Phase - the time at which the task starts its first period,
- Active Resource - the CPU the task should run on,
- Amount of Work - the execution time for the task.

The General Task Data Edit Dialog allows a user to enter the appropriate task data in the window, update the General Task Data (*OK*), print the screen to a file or to a printer (*Print*), cancel an operation (*Cancel*) and obtain help (*Help*).

In addition to the described General Task Data, every task is characterized by a list of *Optional Intervals, Non-Preemptable Sections* and *Resource Requirements.* Normally, a task, once scheduled, executes entirely. However, some tasks contain optional parts, which are specified by means of Optional Intervals. They are characterized by a *Start* and *End* Time. The task may contain more than one Optional Interval.

In a preemptive environment a task may be preempted by another task of higher priority. Sometimes a task should not be preempted during some certain sections of its execution called *Non-Preemptable Sections.* Similar to Optional Intervals, they are characterized by a Start and End Time. A task may contain more than one Non-Preemptable Section.

A task may require use of one or more resources during its execution. The resource requirements are described by *Resource Name, Start Time* and *End Time.*

To edit any of the already described parameters (Optional Intervals, Non-Preemptable Sections or Resource Requirements), the user must click on appropriate menu bar to pop up a corresponding Edit Dialog Window. Each window contains the appropriate fields for editing the chosen parameter, including a summary of all attributes. A user can enter the values of parameters in the window dialog, add new instances of a parameter (*Insert*), remove an instance (*Delete*), modify parameters of an existing instance (*Modify*), update the data (*OK*), cancel an operation (*Cancel*) and seek help(*Help*).

2.2 Resource Graph Editor

The Resource Graph describes the physical and logical resources available to the task system. It includes all the resources of the system and their *relationships.* A relationship means that the resources may be included (*a-part-of* type) or accessed (*accessible-from* type) by another resources. A database residing at a node is an example of a-part-of relationship (where the database is a part of the node). A database accessible from another node is an example of accessible-from relationship.

To describe a Resource Graph, the user must provide a complete description of every resource and its relationship with other resources. Resource Graph Editor provides a user-friendly interface to create and edit information about resources.

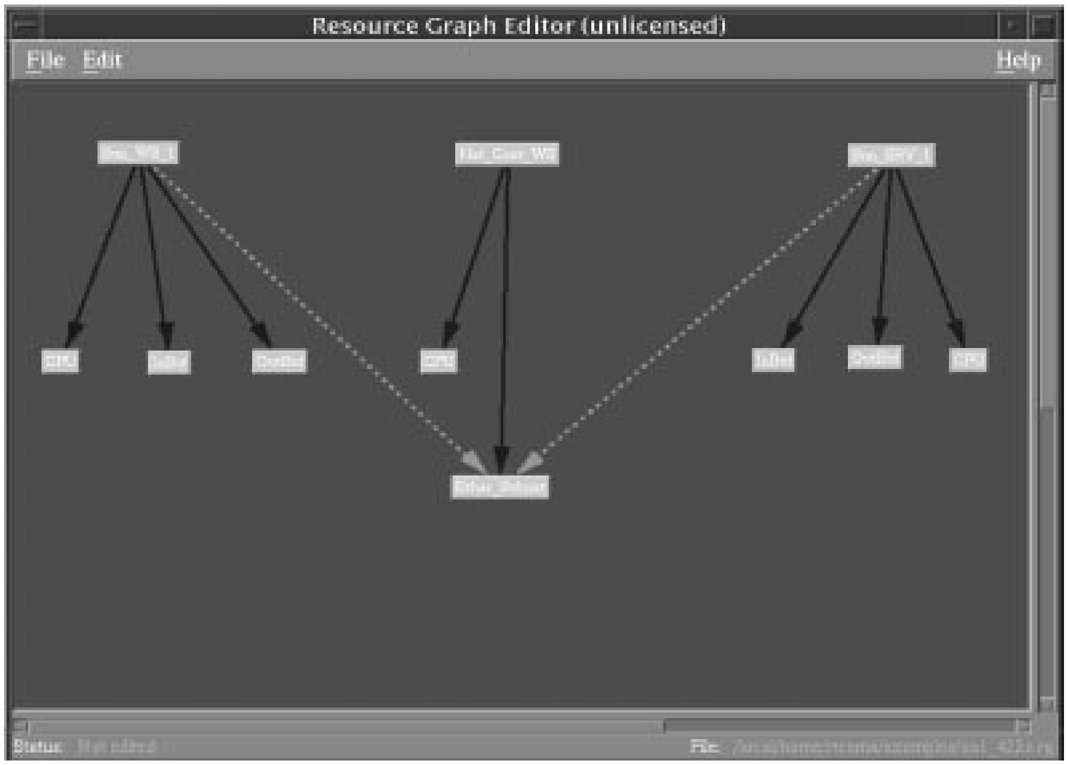


Figure 2.2: Snapshot of PERTS Resource Graph Editor

The Resource Graph Editor, shown in Figure 2.2, enables the user to create and update resources. All resources of a resource graph are represented by rectangular nodes, all relationships

- by directed edges connecting the appropriate nodes (solid red for *a-part-of* and dashed green for *accessible-from*). Similar to the task graph editor, a resource graph is described by choosing an appropriate operation in a menu bar and clicking on the appropriate node or edge.

2.3 Schedulability Analyzer

There are two complementary techniques in evaluating the timing behavior of a real-time system:

1. Schedulability analysis based on theoretical calculations and
2. Simulation.

Schedulability analysis rigorously checks whether timing constraints can be met, but requires an analyzable model of the system under consideration. On the other hand, the simulator provides no guarantees; it determines whether timing constraints are violated, relying on user's specification of the worst-case configuration. However, a simulator can deal with a more complex system than the schedulability analyzer. This project focuses on the schedulability analyzer, which can guarantee system schedulability. We would like to emphasize that since the schedulability analysis is based on sufficient (not necessary) criteria, it cannot guarantee non-schedulability of a task system. If a task system does not satisfy the schedulability criteria, it does not mean that it is not schedulable, but instead it means that theory is unable to guarantee its schedulability.

The Schedulability Analyzer is the last of the three key PERTS components. It performs the schedulability analysis for the systems that have been described using the task graph and resource graph. A system is analyzed by choosing a priority assignment mechanism and a resource access protocol.

2.3.1 Priority Assignment Mechanisms and Resource Access Protocols

The list of priority assignment mechanisms in the Schedulability Analyzer includes:

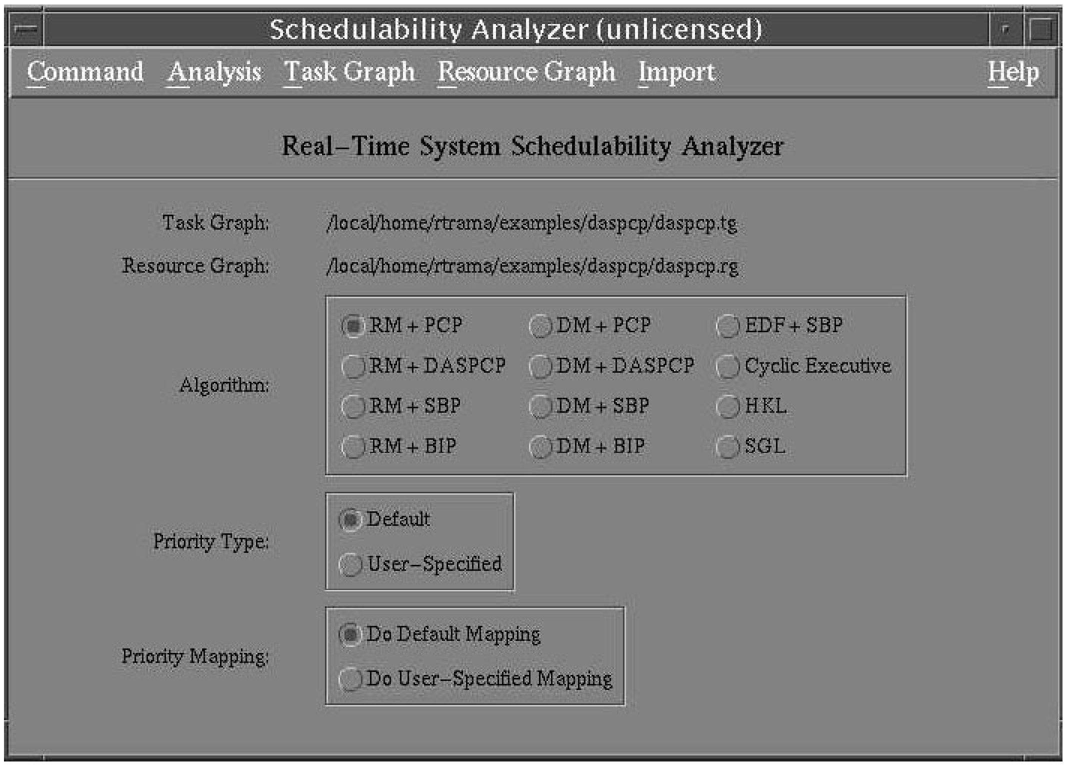
- Rate Monotonic (*RM*) [5] - which assigns higher priority to a task executing at higher rate,
- Deadline Monotonic (*DM*) [6] - which assigns higher priority to a task with shorter relative deadline,
- Earliest Deadline First (*EDF*) [5] - which assigns a higher priority to a task with faster approaching deadline.

There are three other Priority Assignment mechanisms currently supported by PERTS, Cyclic Executive (*CE*), Harbour-Klien-Lehoczky (*HKL*) and Sun-Gardner-Liu (*SGL*), not applicable for our study because of their limitations. CE is applicable for a system containing harmonic tasks only, while HKL and SGL prohibit resource accesses.

The list of available resource access protocols includes:

- Priority Ceiling Protocol (*PCP*) [7] - which avoids deadlocks, limits blocking time and guarantees that the blocking time is a function of duration of critical sections only;
- Basic Inheritance Protocol (*BIP*) - which is similar to PCP. It is easier in implementation than the latter, but does not limit the number of times a task may be blocked and does not prevent deadlocks;
- Stack Based Protocol (SBP) [8] - which assigns a fixed preemption level to every task inversely proportional to its relative deadline. It avoids deadlocks and multiple blocking, but applicable only to Single-Node systems.

After the user specifies a priority assignment mechanism and a resource access protocol and chooses the appropriate text files with that describe the tasks and resources, the system is completely described.



Real–Time System Schedulability Analyzer

Task Graph: /local/home/rtrama/examples/daspcp/daspcp.tg

Resource Graph: /local/home/rtrama/examples/daspcp/daspcp.rg

Algorithm: [✓ RM + PCP] DM + PCP EDF + SBP RM + DASPCP DM + DASPCP Cyclic Executive RM + SBP DM + SBP HKL RM + BIP DM + BIP SGL

Priority Type: [✓ Default] User–Specified

Priority Mapping: [✓ Do Default Mapping] Do User–Specified Mapping

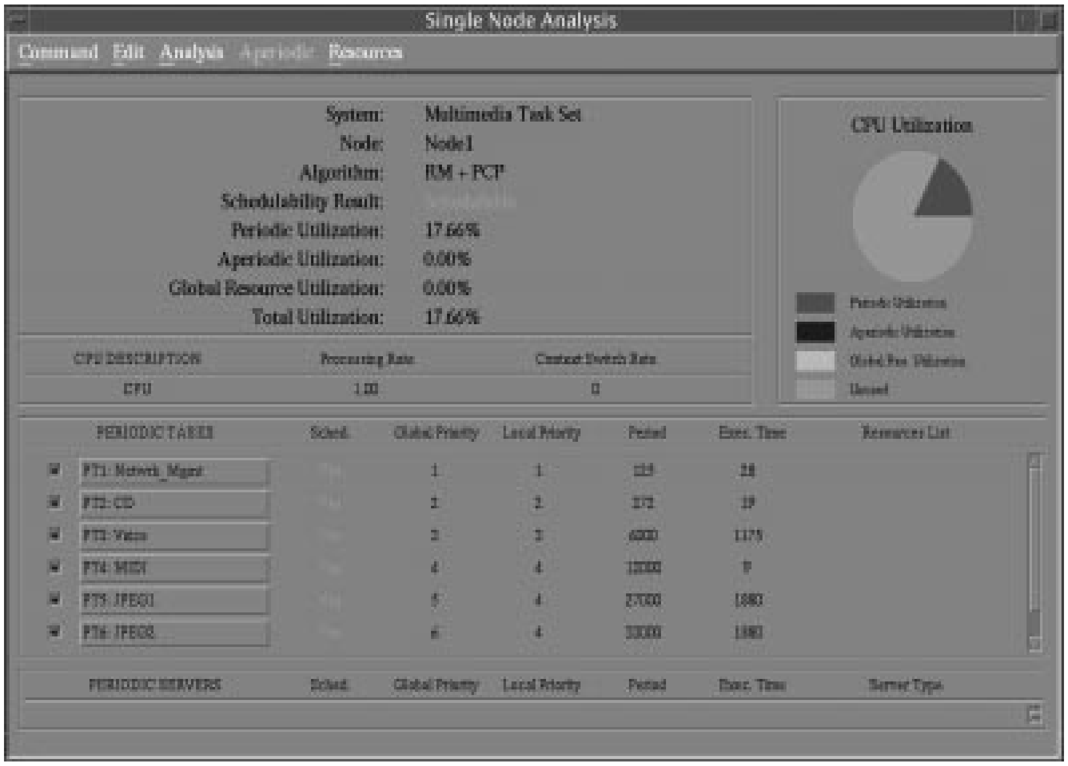
Figure 2.3: Snapshot of PERTS Schedulability Analyzer

2.3.2 Schedulability Analysis Regimes

There are three different regimes of analysis provided by PERTS: *Single-Node, Multi-Node* and *End-to-End.*

Single-Node Analysis, shown in Figure 2.4, determines whether the node is schedulable. A task is *schedulable* if it always completes its execution before its deadline; a node is *schedulable* if all the tasks assigned to that node are schedulable. In addition to the report on schedulability of the node, the Single-Node analysis reports the CPU utilization, and it provides the list of all tasks indicating their schedulability. The user can modify the system parameters to allow *what if?* modeling.

Single-Node analysis may be used for systems consisting of a single node, as well as for multiple node systems. In the latter case, the user should choose for analysis either Multi-Node or End-to-End Analysis, described below. However, to obtain details on the particular node of the distributed



System:	Multimedia Task Set
Node:	Node1
Algorithm:	RM + PCP
Schedulability Result:	[illegible]
Periodic Utilization:	17.66%
Aperiodic Utilization:	0.00%
Global Resource Utilization:	0.00%
Total Utilization:	17.66%

CPU DESCRIPTION	Processing Rate	Context Switch Rate
CPU	1.00	0

PERIODIC TASKS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Resources List
PT1: Netwrk_Mgmt	[illegible]	1	1	125	28	
PT2: CD	[illegible]	2	2	272	19	
PT3: Video	[illegible]	3	3	6000	1175	
PT4: MIDI	[illegible]	4	4	12000	9	
PT5: JPEG1	[illegible]	5	4	27000	1880	
PT6: JPEG2	[illegible]	6	4	33000	1880	

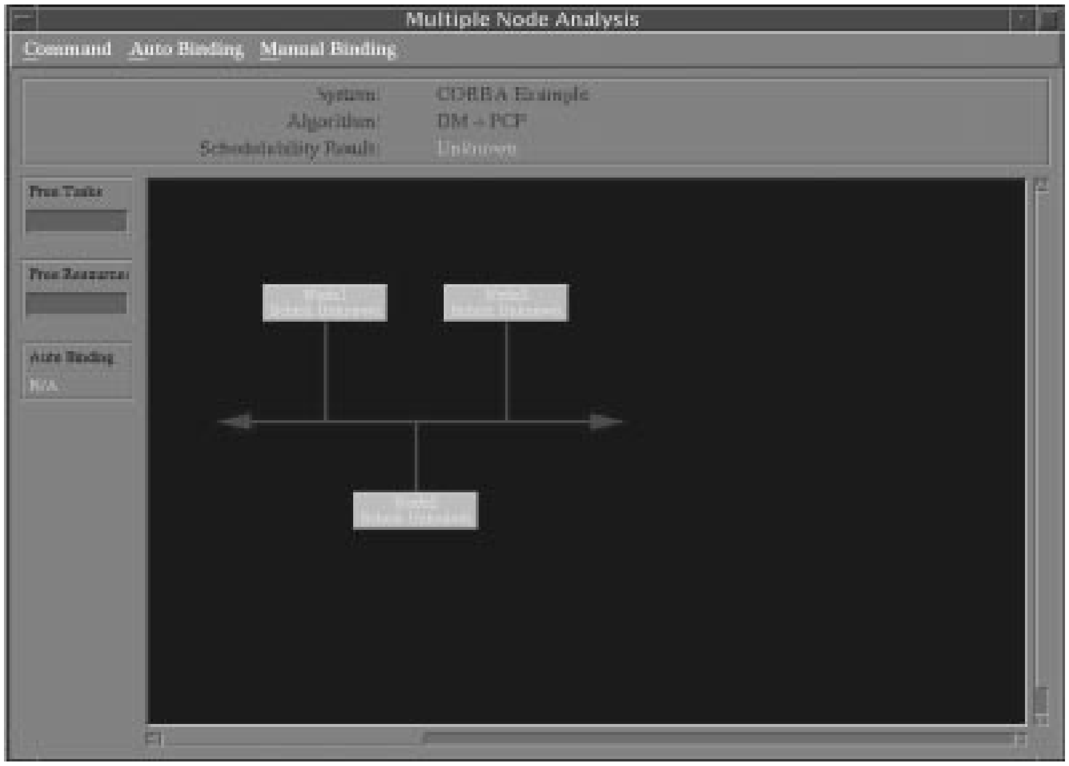
PERIODIC SERVERS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Server Type

Figure 2.4: Snapshot of PERTS Single-Node Analysis

system the user may use the Single Node analysis.

Multi-Node Analysis, shown in Figure 2.5, examines the schedulability of multiple-node real-time systems. To allow *what if?* modeling, the Multi-Node analysis interface enables modification of the binding of tasks and resources to different system nodes. The binding may be either manual or automatic, using such algorithms as best fit, first fit, next fit and worst fit.

The ability to analyze system architectures that have more than one node and share resources can be critical for distributed real-time system developers. PERTS can help point out potential overhead problems and blocking problems that may be introduced by sharing resources across the nodes. Individual entities may be schedulable as stand-alone entities, but when put in a multiple-node architecture with the resource sharing, they may become non-schedulable. Multi-Node analysis



System: CORBA Example
Algorithm: DM + PCP
Schedulability Result: Unknown

Figure 2.5: Snapshot of PERTS Multi-Node Analysis

reports the system schedulability and then user may select individual nodes to analyze, with Single-Node analyzer.

Both Single-Node and Multi-Node analysis dialogs offer a node-oriented view of the system under consideration. They do not perform any path analysis in the systems with task dependencies.

End-to-End analysis, shown in Figure 2.6, looks at the schedulability of a system with one or more paths of execution defined by a series of task dependencies. The End-to-End Analysis Window graphically represents all tasks and dependencies (similar to Task Graph Editor). Specifying any path, the user obtains a schedulability report on that particular path. The user may choose Single-Node Analysis for the detailed information on the particular node.

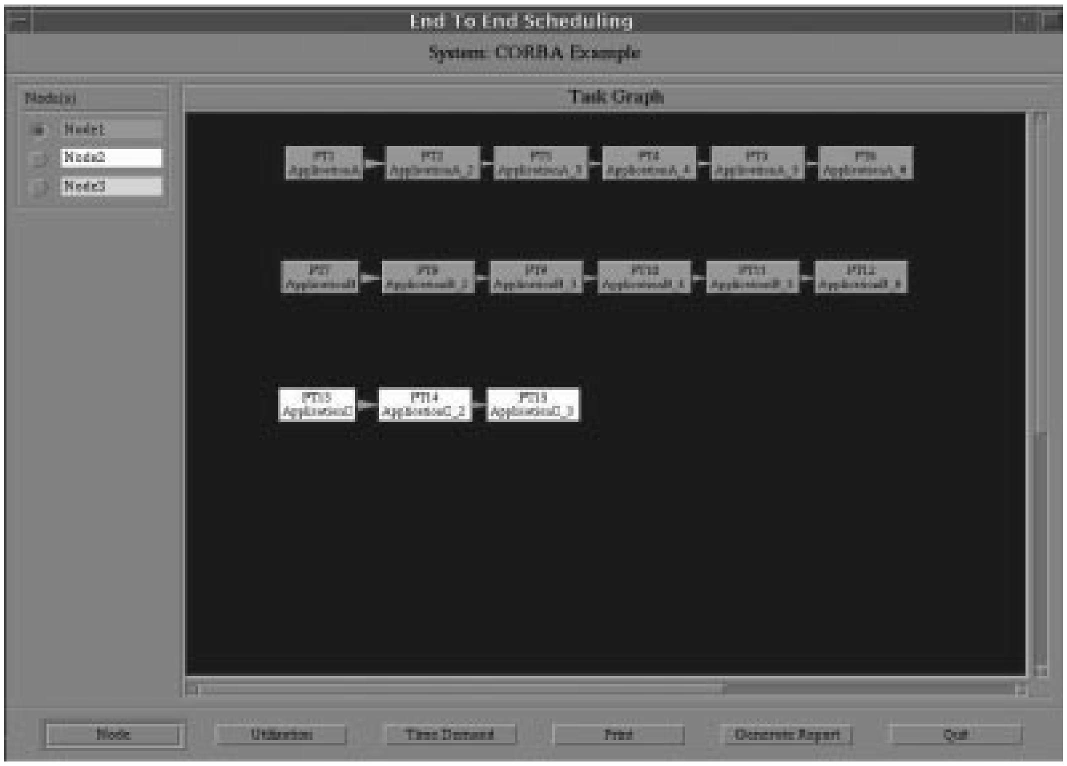


Figure 2.6: Snapshot of PERTS End-to-End Analysis

2.3.3 Theory Behind Schedulability Analysis

The main feature of all three analysis regimes in PERTS is the ability to guarantee the system schedulability. In this section we describe the theory underlying this analysis. There are two sufficient conditions for the schedulability of a real-time system. One of them is based on the concept of Processor Utilization Bound introduced by Liu and Layland [5] and another - based on the concept of Processor Time Demand introduced by Lehoczky et. al. [9]

Liu-Layland's criterion requires satisfaction of the following inequality:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1)$$

(2.1)

where tasks are indexed in the decreasing priority order (task T_1 has the highest priority on the considered node). C_j and T_j denote the worst-case execution time and period of the task T_j. B_i is the worst-case blocking time potentially suffered by any job in the task T_i due to resource contention or non-preemptive execution of lower-priority tasks. If a task T_i satisfies this condition, it is schedulable by the RM or DM and PCP or SBP. Schedulability of all tasks of the system means the system is schedulable.

When a fixed-priority scheduling algorithm is used with a resource access control protocol that effectively bounds priority inversion, there is another more accurate schedulability condition [7, 9]. This condition is stated in terms of the worst-case cumulative demand function $W_i(t)$ for processor time in the interval between the release time of a task T_i and the time t units after its release. The demand function $W_i(t)$ is given by

$$W_i(t) = \sum_{j=1}^{i-1} C_j \lceil \frac{t}{T_j} \rceil + C_i + B_i$$

(2.2)

The demand function has three parts: the processor time demand by all tasks with priorities equal or higher than T_i, the demand of T_i itself, and the worst-case blocking time suffered by each job in T_i. The job released at time t_0 completes at time $t_0 + t$, if $W_i(t) = t$. Consequently, whenever $W_i(t) \leq t$, for some t smaller than task T_i's relative deadline, the task T_i is schedulable.

Chapter 3

Real-Time CORBA

3.1 CORBA

The Common Object Request Broker Architecture (CORBA) is an answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or what underlying system they use. CORBA provides a uniform way for any object to receive and respond to a request from any requester (client).

The *Object Request Broker* (ORB), CORBA's key component, is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass the parameters, invoke its method, and return the results. The ORB facilitates the processing of client requests. A client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

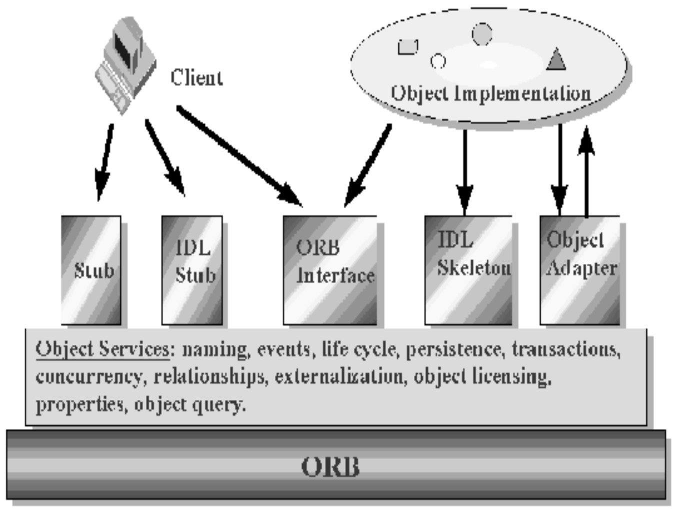

Client
Object Implementation
Stub
IDL Stub
ORB Interface
IDL Skeleton
Object Adapter
Object Services: naming, events, life cycle, persistence, transactions, concurrency, relationships, externalization, object licensing, properties, object query.
ORB

Figure 3.1: CORBA Architecture

The CORBA specification includes: an *Interface Definition Language (IDL)*, that defines the object interfaces within the CORBA environment; an *Object Request Broker (ORB)*, which is the middleware that enables seamless interaction between distributed client objects and server objects; and *Object Services*, which facilitate standard client/server interaction with capabilities such as naming, event-based synchronization and concurrency control.

To provide these capabilities, the CORBA specification defines an architecture of interfaces that may be implemented in different ways by different vendors. The architecture was specifically designed to separate the concerns of interfaces and implementations. The architecture, shown in Figure 3.1, has been described in detail in [2]

3.2 Real-Time CORBA

Real-Time distributed applications such as automated factory control, avionic navigation and simulation have demonstrated the need to extend the CORBA standard to support real-time.

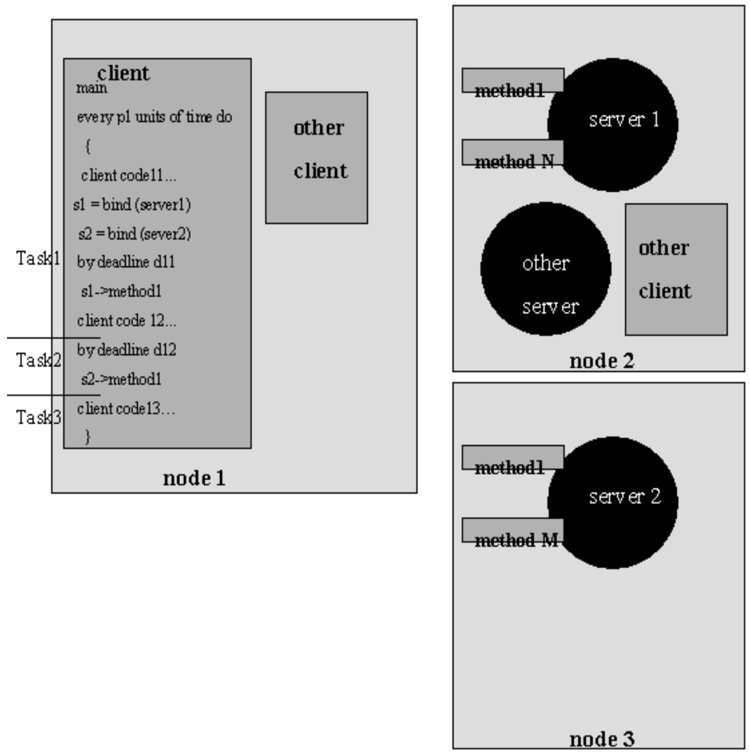


Figure 3.2: Example of the RT CORBA System

A RT CORBA client contains a set of requests to RT CORBA servers (method calls) intermixed with its local code. In addition to its final timing constraint (deadline) a client may contain a series of intermediate timing constraints (*Intermediate Deadlines*), associated with different method calls, calculations and data manipulations. A intermediate deadline is specified by three time parameters: *Start Time*, *End Time* and *Deadline*. The start and end time describe the beginning and end of the portion of the client code to be completed by the Deadline.

We illustrate a typical RT CORBA client in Figure 3.2. This client (residing on $node1$) has a period of $P1$ units of time during which it makes two CORBA calls to remote CORBA servers ($s1 \rightarrow method1$ and $s2 \rightarrow method1$). Each CORBA call has its own pre-period deadline, shown by horizontal line ($d11$ for $s1 \rightarrow method1$ and $d12$ for $s2 \rightarrow method1$). Note that there is also some local client code before, after, and between CORBA method calls. The figure shows remote servers only, while in general some or all servers could reside on the client's node.

The distributed CORBA architecture causes network communication between clients and servers residing on different nodes. The time that clients spend sending requests to remote servers (Network Delay), may be significant enough to make the clients non-schedulable. To model network delay between two nodes in PERTS, a user inputs the worst case estimate of the one-way latency between the two nodes.

3.3 RT CORBA Scheduling Service

A Special Interest Group (SIG) has been formed within the Object Management Group with the goal of extending the CORBA standard to support real-time applications. The real-time SIG put out a request for proposals seeking static real-time scheduling in a real-time CORBA framework. Among other things the RFP asked for end-to-end predictability of client requests, ordered execution of tasks and real-time control of resource allocation.

Real-Time Research group at University of Rhode Island, in association with SPAWAR Systems, Tri-pacific Software Inc. and others responded to the RFP. The group focused on expression and enforcement of various fixed priority scheduling policies across a RT CORBA system.

The proposed OMG Real-Time CORBA standard [10] uses the notion of a global, uniform priority assignment to threads of clients and servants in the CORBA system. Global Priority is a total ordering of those threads in the system that are assigned a priority in application code (e.g servant threads that inherit the priority of the client), but that eventually every thread will be assigned a global priority. The programmer assigns each client thread one or more fixed, unique global priorities from 1 to N, with 1 being the lowest and N being the highest priority. A client may have more than one priority due to parts of its execution that have tighter timing constraints or higher importance.

Fixed priority scheduling entails, whenever possible, resolving scheduling conflicts by allowing the highest global priority thread to use a resource on which the conflict occurs. When, for some reason such as consistency of a shared resource, the RT CORBA system does not resolve conflicts in priority order and causes a higher priority thread to wait for a lower priority thread, *Priority Inversion* is said to occur. Analyzable real-time systems require that the priority inversion be bounded by time.

RT CORBA standard also specifies a *Scheduling Service* that uses the RT CORBA primitives to facilitate enforcing real-time scheduling policies across the RT CORBA system. The scheduling service abstracts away from the application some of the complication of using low-level RT CORBA constructs such as POA policies. A POA is an object in the CORBA server that controls access to server objects. For applications to ensure that their execution is scheduled according to a uniform policy, such as global Rate Monotonic scheduling, RT ORB primitives must be used properly and their parameters must be set properly in all parts of the CORBA system. A scheduling service implementation will choose CORBA priorities, POA policies and priority mappings (from global priorities to priorities on a specific system) in such a way as to realize a uniform real-time scheduling policy.

The scheduling service uses *Names* (character strings) to provide abstractions of scheduling parameters (such as CORBA priorities). The application code uses these names to specify CORBA activites and CORBA objects. The scheduling service internally associates these names with actual scheduling parameters and policies. This abstraction improves portability with regard to real-time features, eases use of real-time features and reduces the chances of errors.

The scheduling service provides a *schedule_activity* method that accepts a name and then internally looks ip a pre-configured CORBA priority for that name. The scheduling service also provides a *create_POA* method to create a POA. A call to *create_POA* sets the POA's RT CORBA policies to support the uniform scheduling policy that the scheduling service is enforcing.

The scheduling service provides a third method called *schedule_object*, that accepts a name for the object and internally looks up scheduling parameters for the object. The example in Figure 3.3, from the RT CORBA draft standard [10] illustrates how the scheduling service could be used and also highlights some of the issues in creating RT CORBA clients and servers. Assume that a CORBA object has two methods: $method1$ and $method2$. A client wishes to call $method1$ under one deadline and $method2$ under a different deadline.

```
0 install_priority_mapping(···);
Client
C1 sched = create scheduling service object;
C2 obj = bind to server object
C3 sched → schedule_activity("activity1");
C4 obj → method1(params); // invoke the object
C5 sched → schedule_activity("activity2");
C6 obj → method2(params);

Server Main

S1 sched = create scheduling service object;
S3 poa1 = sched → create_POA( ··· );
S4 obj = poa1 → create_object(params); // create object
S5 sched → schedule_object(obj, "Object1");
```

Figure 3.3: Example of the RT CORBA Static Scheduling Service

In Step 0, the scheduling service installs a priority mapping that is consistent with the policy enforced by the scheduling service implementation. For instance, a priority mapping for an analyzable deadline monotonic policy might be different from a analyzable rate monotonic policy.

The *schedule_activity* calls on lines $C3$ and $C5$ specify names for CORBA activities. The scheduling service internally associates these names with their respective CORBA priorities.

The server in the example has two scheduling service calls. The call to *create_POA* allows application programmer to set non real-time policies, and internally sets the real-time policies to enforce the scheduling algorithm. The resulting POA is used in line $S4$ to create the object. The second scheduling service call in line $S5$ allows the service to associate a name with the object. Any

RT scheduling parameter for this object, such as the priority ceiling, is assumed to be internally associated with the object's name by the scheduling service implementation.

3.3.1 PERTS for RapidSched

Real-Time Research and Tri-Pacific Software Inc. have developed *RapidSched*, an implementation of the proposed real-time CORBA scheduling service. RapidSched uses a global deadline monotonic priority assignment technique along with distributed priority ceiling resource access protocol for fixed priority static distribured systems. The implementation has been designed to work closely with PERTS. With PERTS, system designers enter information about their RT CORBA clients and servers using a graphical user interface. These high level objects are automatically translated into PERTS primitives [14]. A rate-monotonic analysis then determines whether the system is schedulable.

If the system is schedulable, PERTS produces a file from which the scheduling service (RapidSched in our case) can retrieve scheduling information about all tasks, GCSs [1] and resources. The file associates textual names of tasks, GCSs and resources with their scheduling information. This information is later transparently retrieved by RapidSched in enforcing scheduling decisions for the application. For more information about RapidSched, we refer the reader to [18].

PERTS 3.2 supported limited analysis of RT CORBA clients and servers. It does not address some of the other issues such as Priority Mapping, automatic generation of the configuration file. In the following chapters we discuss some of these and other issues. The possibility of using *Distributed Affected Set Priority Ceiling Protocol (DASPCP)* is discussed in Chapter 5.

[1]A *Critical Section* is a piece of code that is run on behalf of many tasks. A critical section shared by tasks executing on different nodes is called *Global Critical Section (GCS)*.

Chapter 4

Priority Mapping

Schedulability analyzer tools like PERTS assume an unlimited priority range and assign unique priorities to tasks and GCSs based on this assumption. These priorities do not correspond to the run-time priorities supported by the operating system on which the task system is scheduled.

The designer has to decide at what priorities each task should be scheduled. Typically in complex real-time systems, there are hundreds of tasks and it is very difficult to assign run-time priorities to all the tasks manually so that the system is schedulable. The problem is complicated when the operating system's priority range is less than the number of distinct priorities of the tasks.

4.1 Problem Statement

There are two instances of the priority mapping problem:

1. When the number of tasks and GCSs with unique priorities running on a node is *not greater* than the number of priorities supported by the operating system. This is a simple problem and the solution is to simply assign actual run-time priorities to tasks while preserving the original priority order. For example, consider a task system with five tasks T_1, T_2, T_3, T_4, and T_5 each assigned a priority of 10, 20, 30, 40 and 50 respectively by a tool like PERTS. The

operating system on which these tasks are scheduled has a priority range of 1 through 6. One possible run-time priority assignment could be:

$T_1 : 1, T_2 : 2, T_3 : 3, T_4 : 4$ and $T_5 : 5$.

2. When the number of tasks and GCSs with unique priorities running on a node is *greater* than the number of priorities supported by the operating system. This problem is more complicated and the solution is not trivial. For example, suppose there are 100 tasks, each assigned a unique priority (say 0 through 99) by PERTS, to be run on a node with Solaris operating system which supports only 60 (0 through 59) priority levels.

Clearly a solution to the second instance must assign same priorities to more than one task which might cause *priority inversion* since most operating systems use FIFO scheduling for tasks with same priority.

It is important to note that there are an *exponential* number of such mappings for any system. Consider the number of ways to arrange n tasks into m priority groups, where $m \leq n$. The total number of ways n tasks can be arranged into m priority groups [11] is:

$$N = \frac{(n-1)!}{(m-1)!(n-m)!} \tag{4.1}$$

Formally stated, the problem is:

Suppose that in a distributed system, a processor P has m tasks and n GCSs scheduled to be run on it. The global scheduler (like PERTS) will assign a unique priority to every task and

GCS in the system. Thus, on processor P, $m+n$ unique priorities are required. If the operating system running on P has fewer than $m+n$ priorities, two or more tasks and/or GCSs will have to execute at the same priority. This could cause priority inversion, since a task (with originally higher priority) could be blocked by another task (with originally a lower priority) ahead of it in the FIFO queue. This new form of blocking must be taken into account when computing the schedulability of the system. We will also refer the original priority assigned to a task/GCS by PERTS as *Global Priority* and the priority assigned to a task/GCS as a result of priority mapping is called *Local Priority*. Our goal is to find a mapping from global to local priorities, if it *exists*, with the condition that the system is *Schedulable* when all the tasks and GCSs are scheduled at their local priorities on their respective nodes. A subtle point to be noted here is that a mapping algorithm must map even the priority ceilings of resources generated by PERTS to a number in the priority range specified by the operating system on the particular node in consideration.

In the following section we present the modifications to the schedulability criteria and in the subsequent sections discuss algorithms to perfrom priority mapping.

4.2 Effect of Limited Priorities on Schedulability Analysis

The effect of limited priority levels available to a schedulability analyzer tool is discussed in detail in [11] and [13]. Here we present only the results from these works.

To analyze the schedulability of a system with limited priority levels, we check how limited priorities affect the time demand function introduced by Lehoczky's schedulability criterion, Equation 2.2. Assuming FIFO scheduling of tasks with same local priority and making the worst case assumption that each task or GCS falls at the end of the FIFO queue for its priority, the modified time demand function is:

$$W_i(t) = \sum_{j=Alltasksofhigherpriority} C_j \lceil t/T_j \rceil + \sum_{k=Alltasksofsamepriority} (C_k * M_k) + C_i + B_i \leq t$$

(4.2)

Here C_i represents execution time of task T_i, B_i is the blocking time of task T_i and M_k is a factor defined as

$$M_k = min\{\lceil t/T_k \rceil, n_g + 1\}$$

where n_g is the number of remote GCSs executed by the task T_i during a single period. The origin of this factor is that the task T_i may wait for the end of some same priority task's execution. It may wait once, when the task T_i is initialized and every time it releases its CPU for an execution of a remote GCS, since a task of the same priority may get the CPU at that time period. At the same time it may not happen more often than the frequency of the same priority task T_k, $\lceil t/T_k \rceil$

We now present the Lowest Overlap First algorithm for mapping global priorities to local priorities.

4.3 Lowest Overlap First Priority Mapping Algorithm

The *Lowest Overlap First Priority Mapping* algorithm [18], is a possible solution to the priority mapping problem. The algorithm is based on the concept of overlapping multiple tasks and GCSs together into a single local priority. It starts with as many priorities as the global scheduler requires to schedule all tasks and GCSs, which means that, priorities to tasks and GCSs are already assigned assuming an unlimited priority range so that the system is schedulable. It then scans through the tasks in increasing global priority order, overlapping as many tasks and GCSs as possible without allowing the system to become non-schedulable. On each node, it overlaps (maps two or more tasks or GCSs with different global priorities to a same local priority) as many times as necessary to end up with the number of available priorities on that node.

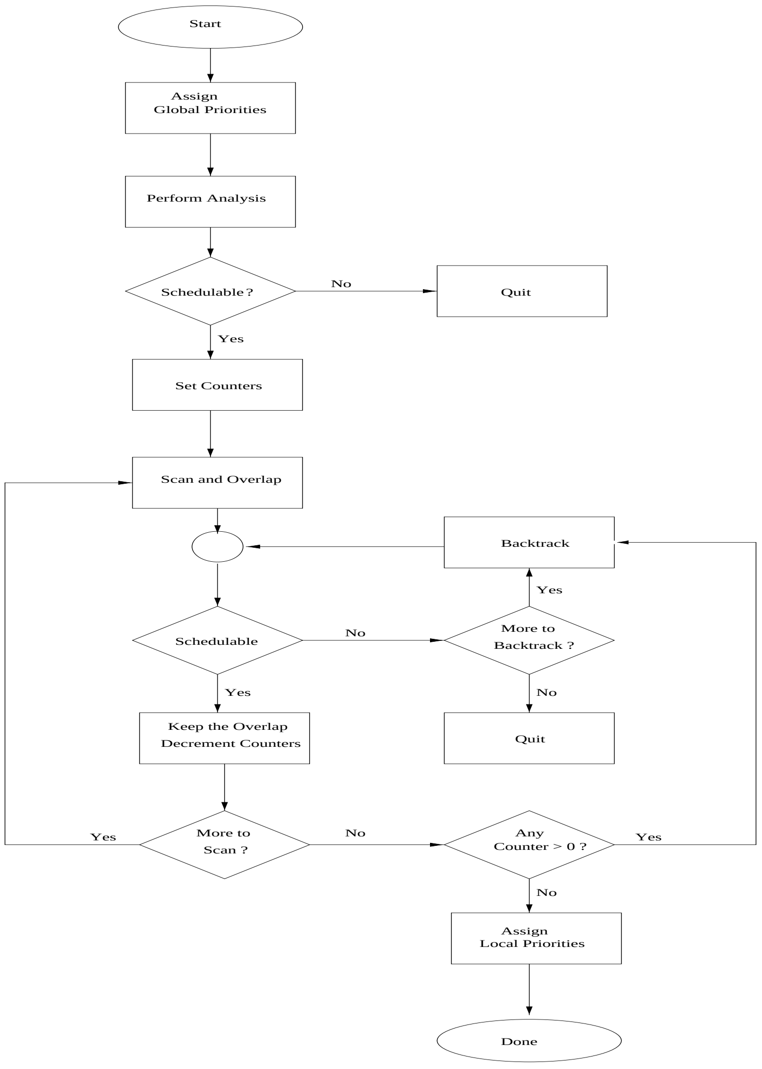


Figure 4.1: Flowchart of Lowest Overlap First Priority Mapping Algorithm

Figure 4.1 displays a high-level flowchart of the algorithm. The details of each step are discussed below.

4.3.1 The Algorithm

Though the first two steps are not a part of the algorithm as such, they are required before the mapping algorithm is invoked.

Assign Global Priorities: The first step in the algorithm is to assign unique global priorities to all tasks and GCSs on all nodes according to a chosen priority assignment mechanism (RM, DM,etc.) under the assumption that the number of priorities available is umlimited.

Perform Analysis: The system is then analyzed for schedulability. If it is schedulable, we invoke the mapping algorithm. If the system is not schedulable, we quit the mapping process since no mapping will ever improve the schedulability of the system.

Set Counters: For every node, a counter is stored that represents the difference between the number of global priorities used on the node and the number of local priorities available. This counter can also be thought of as representing the number of priority overlaps required on the node. Thus, initially we have $COUNT = \#TASK + \#\text{GCS}-\#LOCAL$, where $COUNT$ is the counter, $\#TASK$ is the number of tasks on the node, $\#GCS$ is the number of GCSs on the node, and $\#LOCAL$ is the number of local available priorities on the node.

Scan and Overlap: This is the heart of the priority mapping algorithm. The goal here is to assign tasks and GCSs *temporary* local priorities, overlapping as many as necessary into the most recently allocated local priority without making the system unschedulable. If, on any node, the counter becomes non-positive, then no priority overlaps are necessary, and so tasks and GCSs on that node are assigned to the next available local priority. The algorithm scans the tasks and GCSs in increasing global priority order, regardless of which nodes they reside on.

During the mapping, tasks and GCSs have separate sets of local priorities into which they will be mapped. We will refer to these sets as local task priorities and local GCS priorities. This is done because, under *Distributed Priority Ceiling Protocol (DPCP)*, [7] the GCS priorities must be higher than the task priorities. After the mapping is complete, this distinction is eliminated, and we are left with at most the number of local priorities available on the node.

When a task is chosen during the scan, if its node has a non-positive counter, it is assigned to the next empty local task priority. By empty, we mean a priority that is not yet assigned to any task. Otherwise, if the chosen task is the first (lowest global priority) task on its node, it is assigned the lowest local task priority. If the chosen task is not the first on its node, it is assigned the highest non-empty local task priority, causing an overlap.

When a task is assigned a local task priority, each of its GCSs must also be assigned some local GCS priority on its own node. If the counter on a GCS's node is non-positive, it is assigned the next empty local GCS priority. Otherwise, if the GCS is the first scanned GCS on the node, it is assigned the lowest local GCS priority. If the GCS is not the first on its node, it is assigned the highest non-empty local GCS priority. Thus, in its initial attempt at assigning local priorities to a task and all of its GCSs, the algorithm tries to overlap all of them.

After the assignment of a task and all of its GCSs is done, the algorithm tests the schedulability of the task, accounting for priority mapping by equation 4.2. If it is found to be schedulable, the counters on the task's node and all GCSs' nodes are decremented, and the scan goes on to the next higher global priority task. If the task is found to be non-schedulable, the algorithm backtracks, trying another combination of overlaps and non-overlaps of the task and its GCSs.

After scanning through and assigning a local priority to the highest global priority task, the algorithm goes to the next phase, if the counters on all nodes are non-positive. If there are no more tasks to scan and there are still some positive counters, that is there are still some overlaps required, then the algorithm backtracks to try to find another possible combination of overlaps and non-overlaps of the tasks and their GCSs.

Assign Actual Priorities: The actual priority ranges available could be different on every node and so is the direction of priority (*Lower the number lower the priority* or *vice versa*). In order to make the analysis easier, the Scan and Overlap phase assumes a particular *priority direction* to be same on all nodes and the priority range to be $[0, \#LOCAL]$. After a successful scan and overlap phase, the temporary priorities assigned have to be mapped to actual range on every node. For eg. on a node the actual range could be $[35, 50]$, with priority direction being *Lower Number Lower Priority.* But the scan phase would have assigned priorities in the range $[0, 16]$ with priority direction being *Lower Number Higher Priority.* In this example, all tasks whose temporary priority is 0 will be assigned a *permanent* local priority of 50. Doing this for all tasks and GCSs in the system priority mapping is completed.

Backtrack: The decisions about whether or not to overlap each global priority form a binary tree. The leaves of the tree represent all of the possible combinations of overlaps and non-overlaps in the system. The backtracking involves choosing another one of these combinations and testing its schedulability. In general, the entire tree may be searched in order to find a successful combination (one that is schedulable). After completely searching the tree, if no schedulable solution is found, the algorithm reports that it cannot find a schedulable solution, and then it quits. Otherwise when a schedulable combination is found, the counters on the appropriate nodes (those where overlaps occurred) are decremented and the scan continues.

4.3.2 Example

We now present an example from [18] to illustrate how the priority mapping algorithm works. Figure 4.2 shows a series of snapshots of a system of tasks and GCSs in the process of being mapped. The required number of overlaps for each node is displayed across the top of the figure. The solid lines represent the tasks and the striped lines represent the GCSs. An arrow from a task to a GCS indicates that the task originated the GCS. The curly brackets indicate a local priority to which tasks or GCSs have already been mapped. In each part of the figure, the tasks or GCSs being considered are highlighted. 4.2A represents the system before any local priorities have been mapped. In part 4.2B, the lowest global priority task is mapped to the lowest local task priority on its node. Part 4.2C shows the next two lowest global priority tasks mapped into the lowest local task priorities on their nodes, and the *GCS* associated with one of them is mapped to the lowest local *GCS* priority on its node.

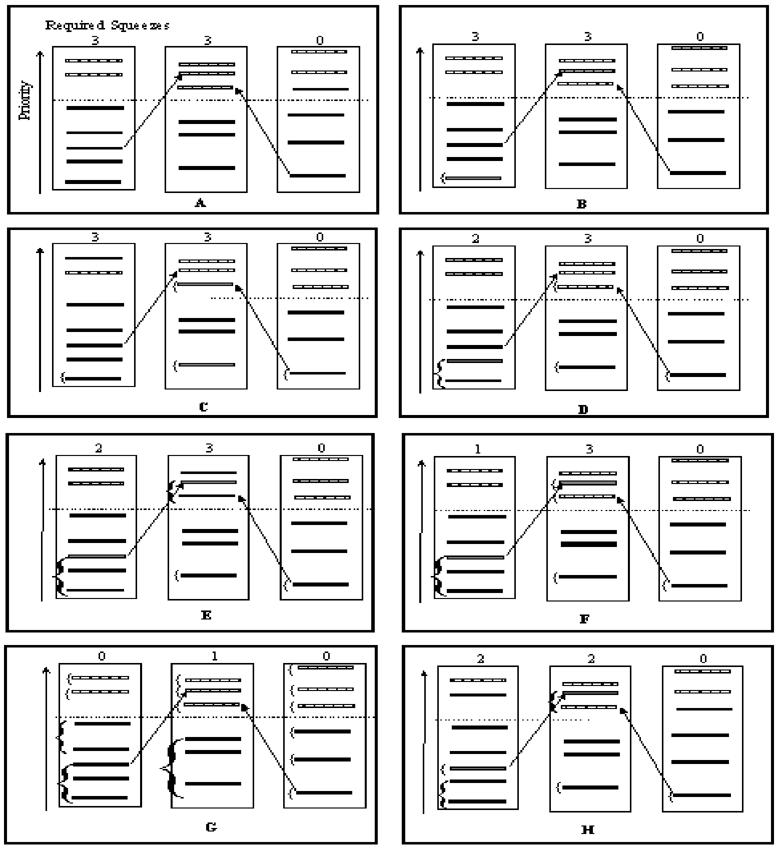


Figure 4.2: Lowest Overlap First Priority Mapping Algorithm Example

Notice that up to this point, no overlaps have been performed because each task and *GCS* that has been considered has been the first on its node. In part 4.2D of the figure, the indicated task is assigned to the highest non-empty local task priority. After the schedulability is tested, the counter on the task's node is decremented. In part 4.2E the task being considered is overlapped into the highest non-empty local task priority, and its *GCS* is also overlapped into the highest non-empty local *GCS* priority. Assuming that, with both the task and the *GCS* being overlapped, the task is not schedulable, so in Figure 4.2F the algorithm has backtracked to attempt another combination of overlaps and non-overlaps of the task and its GCS. The task remains overlapped, but the GCS is assigned to the next lowest empty local GCS priority. Since this configuration is schedulable, the overlap is *committed*, and the counter on the task's node is decremented. Omitting some intermediate steps, in Figure 4.2G the scan is complete, but one of the nodes has a positive counter. So, the algorithm backtracks to the configuration shown in part E of the figure. Recall that in this configuration, when it was found not possible to overlap both the task and its GCS, the algorithm chose not to overlap the GCS. Now, the algorithm attempts another combination of overlaps and non-overlaps of the task and its GCS. In this case, shown in Figure 4.2H, the task is not overlapped and the GCS is overlapped. If this task is schedulable, the algorithm scans the next higher global priority task and continues this way until all tasks have been scanned, and each node has a non-positive counter, or until there are no more configurations to try.

4.3.3 Optimality

In this section we present some of the theorems and their proofs that form the basis for the Lowest Overlap First Priority Mapping algorithm.

A static scheduling algorithm is said to be *Optimal* if, for any set of tasks, it always produces a schedule which satisfies the constraints of the tasks whenever any other algorithm can do so [12].

The Lowest Overlap First priority mapping algorithm produces a direct mapping of global to local priorities. A direct mapping is one in which if any task (GCS) i has higher global priority than any task (GCS) j, then task (GCS) j cannot have higher local priority than that of task (GCS) i. That is, the mapping does not change the relative ordering of task (GCS) priorities. Theorem 1 proves that in the class of direct mappings, the Lowest Overlap First Priority (LOF) Mapping Algorithm is optimal. That is, if there is a direct mapping of global to local priorities that is schedulable, then the mapping produced by LOF algorithm is also schedulable.

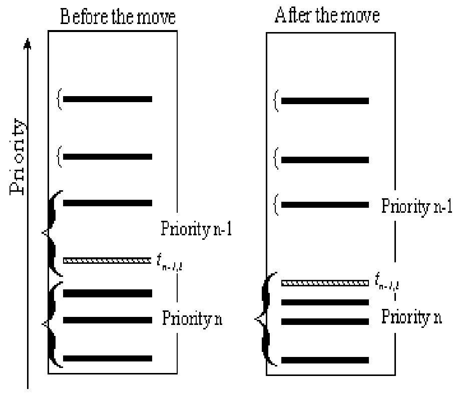


Figure 4.3: Example Priority Move

Theorem 1 *For a given schedulable system of tasks and GCSs with global priority assignments, if there is any direct priority mapping under which the system is schedulable, it is also schedulable under the Lowest Overlap First Priority Mapping Algorithm.*

Proof: The approach we take to proving this theorem is to assume that some schedulable direct mapping exists, and to show that we can derive a Lowest Overlap First mapping from it that is also schedulable.

Let us assume that some direct mapping of global priorities to local priorities exists for a particular node in the system. Assume also that the mapping provides schedulability of the considered system. Let the operating system on the node have n local priorities (where n is the lowest priority). Because the mapping is direct, any task with local priority i, higher than local priority j, has higher global priority than any task with local priority j. Take the lowest global priority task that is assigned to local priority $n-1$ ($t_{n-1,l}$) and temporarily change its local priority to n. We can think of this as moving task $t_{n-1,l}$ out of local priority $n-1$ and overlapping it into the lower local priority n. Figure 4.3 illustrates this move.

We now examine which tasks' schedulability might be affected by this move.

1. The tasks with local priority $n-1$: The worst case completion time of any task with local

priority $n-1$ will not increase because all of these tasks could previously have been blocked under FIFO by $t_{n-1,l}$, and now they cannot.

2. The tasks in local priority n: The worst case execution time of any task with local priority n will not increase because, before the move, any task with local priority n could have been preempted by $t_{n-1,l}$. After the move, the tasks in local priority n can be blocked due to FIFO scheduling within the same priority. The blocking time cannot be greater than the preemption time.

3. Task $t_{n-1,l}$: The worst case completion time of $t_{n-1,l}$ may be affected by the move, making it unschedulable. However, if this were the case, the Lowest Overlap First algorithm would not have made this overlap in the first place, but rather would have mapped $t_{n-1,l}$ to local priority $n-1$.

If task $t_{n-1,l}$ remains schedulable after *moving* it to priority n, we repeat this procedure moving the next lowest global priority from local priority $n-1$ to local priority n, as long as the moved task remains schedulable. Clearly, if we continue this procedure for local priorities $n-2, n-3$ and so on, the resulting mapping will be the one that would have resulted from using the Lowest Overlap First Priority Mapping Algorithm. The procedure for moving GCSs is identical with the exception that on every move, we check the schedulability of the task that generated the GCS in question. Since the schedulability of all tasks is not affected by any of these moves, the system remains schedulable and the theorem is proven. $\square$

The next two theorems prove the optimality of Lowest Overlap First algorithm on Single node systems under all conditions when scheduled under a deadline monotonic priority assignment mechanism.

Theorem 2 *For any system residing on a Single Node and scheduled under Deadline Monotonic priority assignment mechanism, if there is any Indirect priority mapping under which the system is* *schedulable, it is also schedulable under a Direct mapping algorithm.*

Proof: Let us assume that the alternative mapping (not LOF) does exist and provides the schedulability of the system under consideration. Let us assume that the alternate mapping is *Indirect*. Hence, there is atleast one pair of local priorities k and l (priority k is higher than l) such that some task t_i with local priority k has lower global priority than some tasks t_j with local priority l. If the system is schedulable under this mapping then it is also schedulable under a direct mapping algorithm. It is so, because, if we set the local priority of task t_i to l the system still remains schedulable. To demonstrate this, we consider all tasks affected by this change. We remind the reader that a task may become non-schedulable only if its worst case completion time increases.

1. All tasks with local priorities k other than task t_i are still schedulable because their worst case completion time only decreases.

2. All tasks with local priority l are still schedulable because their worst case completion time does not increase.

3. All tasks with local priority less than k but greater than l are still schedulable because their worst case completion time decreases (Earlier they could be preempted by task t_i, now they cannot)

4. While the worst case completion time of task t_i increases it is still schedulable since its worst case completion time is same as the worst case completion time of task t_j, while its deadline is not earlier than that of task t_j. Since task t_j is schedulable, task t_i is also schedulable.

If we continue to lower the local priorities of all *overprioritized* tasks we would have converted the original Indirect mapping to a Direct mapping. □

Theorem 3 *For any system residing on a single node and scheduled under Deadline Monotonic* *priority assignment mechanism, if there is any Indirect mapping under which the system is schedulable, it is also schedulable under Lowest Overlap First priority mapping algorithm.*

Proof: Proof of this theorem is trivial and follows from the previous two theorems. Under the conditions of this theorem, the system residing on a single node and schedulable under deadline monotonic priority assignment, is also schedulable under an Indirect mapping. According to Theorem 2 the system is also schedulable under some direct mapping. But according to Theorem 1 it is also schedulable under the Lowest Overlap First mapping algorithm. □

Chapter 5

The Distributed Affected Set Priority Ceiling Protocol

The advent of real-time object oriented (RTOO) systems, such as Real-Time CORBA middleware [15] and RTOO databases [16], poses the need to control concurrent access to objects under real-time requirements. In a real-time database system, the concurrency control technique manages concurrent accesses by transactions to data objects. In a CORBA system, the middleware must control concurrent access to CORBA objects by remote clients. Concurrency control techniques for RTOO systems must satisfy more requirements than traditional non-real-time concurrency control techniques because they must also meet timing constraints. Among the most important non-real-time requirements are that the technique provides: *high concurrency* to maximize average throughput; *deadlock treatment* that either prevents, avoids or breaks deadlocks; and *logical consistency* such as mutual exclusion or serializability, so that all constraints on the attributes of the object are met. In real-time concurrency control there are similar requirements, along with the requirement that the technique should support *predictable execution*, such as bounded blocking times for locks. Providing predictable blocking times involves, among other things, bounding *priority inversion* that occurs when a lower priority task blocks a higher priority task [7].

In this chapter we briefly describe *Distributed Priority Ceiling Protocol (DPCP)* [7] and DASPCP. We refer the reader to [19] for more information on DASPCP. We use the terms *transaction* and *task* interchangeably. Also the terms *lock* and *semaphore* should be treated similarly. The context should make it clear and we hope there is no ambuiguity in the interpretation of the meanings of these terms.

5.1 DPCP and DASPCP

In this section we describe the Priority Ceiling Protocols and compare concuurency under DPCP and DASPCP.

5.1.1 Affected Set Semantics

A RTOO system consists of *objects*, some of which manage shared resources. The model of a real-time object we use in this discussion is derived from the RTSORAC model [16] for real-time object oriented databases.

Our RTOO system object model extends the traditional object-oriented notion of an object to include attributes that have a value, a timestamp and amount of accumulated imprecision. The imprecision that is recorded accumulates due to the potential relaxation of serializability by semantic concurrency control [16]. Objects also include constraints and a compatibility function. The constraints can be placed on the attributes to express logical and temporal correctness of the objects.

The user-defined compatibility function determines how the methods of the object may interleave. It is through this function that the object designer expresses semantics of allowable concurrency. The flexibility of the compatability function allows the object designer to specify different levels of concurrency for different objects. For instance, one object may require serializability, while another object may tolerate less restrictive form of correctness. To enforce serializability, the object designer may use *affected set semantics* [17] to determine compatability. A method's *Read Affected Set (RA)* is the set of the object's attributes that the method reads. A method's *Write Affected Set (WA)* is the set of attributes that the method writes. Under affected set semantics, two methods m_1 and m_2 are *compatible* if and only if:

$$(WA(m_1) \cap WA(m_2) = \emptyset) \wedge (WA(m_1) \cap RA(m_2) = \emptyset) \wedge (RA(m_1) \cap WA(m_2) = \emptyset)$$

(5.1)

Note that defining lock compatability based on these affected set semantics has been proven to produce serializable object schedules [17].

A less restrictive form of correctness may be needed to express the trade-off between temporal and logical consistency. In such cases, the semantics of compatability between methods are based on dynamic information, including current temporal consistency and imprecision of data. For example, if a method m_1 that reads an attribute *attr* is currently executing, it would violate the logical consistency of m_1's return value if another method m_2 that writes *attr* would execute. However, if the timing constraint on *attr* has been violated, i.e, it has become old, then allowing m_2 to execute would restore the temporal consistency of *attr*. When determining each potential allowable interleaving of method executions, the compatability function can also examine the amount of imprecision that could be introduced by the possible interleaving.

5.1.2 The Priority Ceiling Protocol

Definition 1 (Priority Ceiling) *The* Priority Ceiling *of a resource is defined [7] as the priority of the highest priority transaction that will ever access the resource.*

A priority ceiling protocol [7] uses information about the way in which transactions intend to use the resources of the system to bind priority inversion and to prevent deadlock. It is based on the assumption about the system that every object and every transaction in the system is known *a priori*. Thus, no dynamic information may be used to determine the semantics of concurrency control.

There are three basic steps that apply to any of the priority ceiling protocols:

- Before running, the protocol defines a priority ceiling for every critical section that may be locked. The granularity of these critical sections is the core difference among various priority ceiling protocols.
- At run-time, when a transaction T requests a lock, the lock can be granted only if T's priority is strictly greater than the ceiling of locks held by all other transactions.
- If transaction T's lock request is denied because another transaction T_{low} (a lower priority transaction) holds a lock with priority ceiling equal to or greater than T's priority, T_{low} inherits the priority of T until T_{low}'s lock is released.

Note that no checking of conflict is necessary when granting a lock. This is because conflict in a priority ceiling protocol is captured in the definition of priority ceiling.

5.1.3 The Distributed Priority Ceiling Protocol

The DPCP handles a synchronization of task method calls, executing on distributed systems. Before we start the description of the protocol we introduce the following definitions:

Definition 2 (Global and Local Critical Sections:) *A Semaphore that is accessed by tasks allocated to different processors (a single processor) is referred to as a global (local) semaphore. A critical section guarded by a global semaphore is referred to as a global (local) critical section, GCS (LCS).*

First, all tasks must be bound to processors. A task T executes its non-critical-section code and LCSs on its host processor, while its GCSs may be bound and executed on a processor(s) different than the T's host processor. All GCS's controlled by the same semaphore S_G, and the semaphore S_G itself, are bound to the same synchronization processor. A GCS, generated by task T, is assigned a priority equal to the sum of the *Base Priority Ceiling* P_G [1] and P, the priority of task T. Each processor runs the priority ceiling protocol on the GCSs (considering each thread of execution for executing a GCS as a *task*), the set of application tasks (if any), and the set of global and local semaphores bound to the processor. DPCP prohibits a mixed nesting of LCSs and GCSs.

Example of DPCP

The following example adopted from [19] is not an exhaustive demonstration of possible situations (of blocking, preemption etc.), that may occur under DPCP. Our goal is a simple example, demonstrating benefits of DASPCP relative to DPCP. For more detailed example of application of DPCP we refer reader to the original work by Rajkumar [7].

Example 1 *Consider a distributed system with* 2 *nodes. The application consists of* 3 *tasks and* 2 *databases* (O_{track1} *and* O_{track2}), *guarded by* 2 *locks* (L_1 *and* L_2). *Task* T_3 *is bound to the* $Node1$, *while tasks* T_1 *and* T_4 *are bound to the* $Node2$. P_i *is the priority of of task* T_i. *We will follow the convention* $P_4 > P_3 > P_2 > P_1$. *Let the priorities be:* $P_4 = 4, P_3 = 3, P_2 = 2$ *and* $P_1 = 1$.

[1] A fixed priority, higher than the priority assigned to the highest priority task in the system

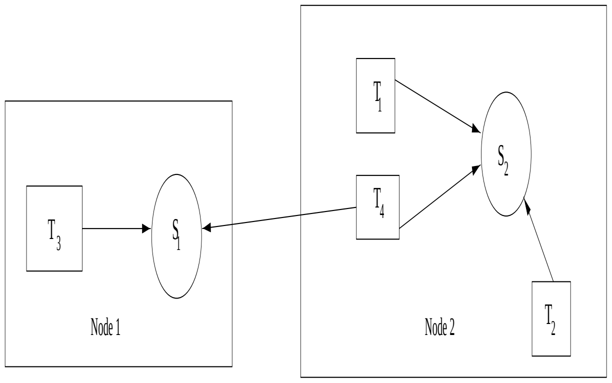


Figure 5.1: A Distributed System with tasks competing for Resources

Tasks T_1, T_3 and T_4 execute the following sequence of steps.

-

T$_1$: $\cdots O_{track2} \rightarrow read\_speed \cdots$

T$_3$: $\cdots O_{track1} \rightarrow write\_speed \cdots$

T$_4$: $\cdots O_{track1} \rightarrow read\_altitude \ldots O_{track2} \rightarrow read\_depth$

Note: In our system the priority of task T_i, $p(T_i)$, is assumed to be lower than that of T_{i+1}.

Object O_{track1} and its lock L_1 are bound to $Node1$. Object O_{track2} and its lock L_2 are bound to $Nodes2$. The priority ceilings of each locks, and the normal execution priority of each critical section thread are listed in Table 5.1.

Priority Ceiling of Locks	
Lock	**Priority Ceiling**
L_1 (Global)	$4 + 4 = 8$
L_2 (Local)	4

Normal Execution Priorities of Critical Sections		
Task	**Critical Section Guarded by**	**Execution Priority**
T_1	L_1	4
T_3	L_2	$3 + 4 = 7$
T_4	L_1	$4 + 4 = 8$
	L_2	4

Table 5.1: Priority Ceilings of Locks in DPCP Example

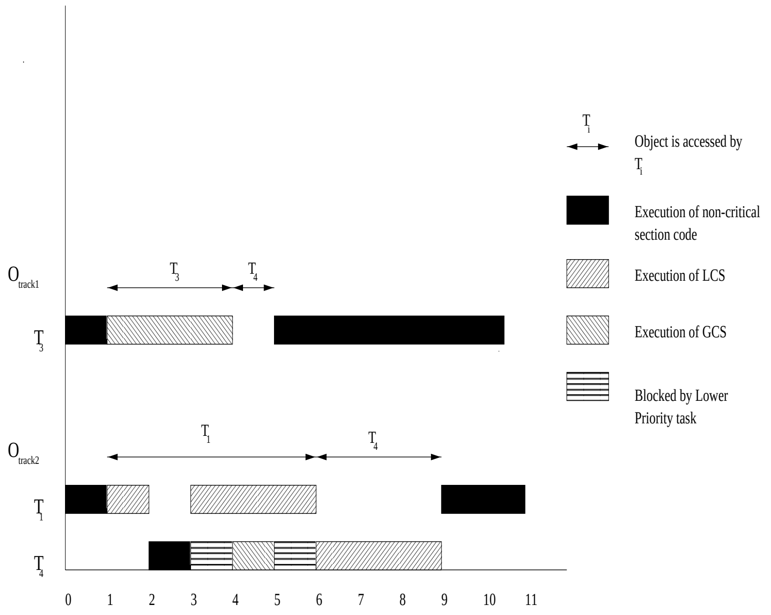


Figure 5.2: Time diagram for task system described in Example 5.1.3

The following example demonstrates the sequence of events in the system under DPCP, presented graphically in Figure 5.2:

- At time t_0, task T_1 arrives on $Node2$ and begins its execution. Similarly, task T_3 begins execution on $Node1$.

- At time t_1, task T_1 gets the local lock L_2 on $Node2$ and begins execution of LCS at its normal execution priority of P_1. Task T_3 locks the global lock L_1 on $Node1$ and begins execution of GCS at its normal execution priority of $P_3 + P_G$.

- At time t_2, task T_4 arrives on $Node2$ and preempts T_1. Task T_3 continues its execution of GCS on $Node2$.

- At time t_3, task T_4 requests the global lock L_1. Since the priority of T_4's $GCS(4 + 4 = 8)$ is not greater than the priority ceiling of the held lock $L_1(8)$, T_4 is blocked and T_3 continues its GCS execution at the inherited priority of $P_4 + P_G(4 + 4 = 8)$. Task T_1 resumes its execution of LCS at $Node2$.

- At time t_4, task T_3 completes the execution of its GCS and releases the global lock L_1 and resumes its own priority. Task T_4 gets the global lock L_1 on $Node1$ and begins execution of GCS at its normal execution priority of $P_4 + P_G(4 + 4 = 8)$. Task T_3 is preempted by higher priority T_4's GCS. Task T_1 continues the execution of its LCS at $Node2$.
- At time t_5, task T_4 completes the execution of its GCS and releases the global lock L_1. T_3 resumes its execution on $Node1$. T_4 attempts to get the lock L_2. However, the priority of $T_4(4)$ is not greater than the priority ceiling of the held lock $L_2(4)$, T_4 is blocked and T_1 continues its execution with inherited priority of $P_4(4)$.
- At time t_6, task T_1 completes the execution of its LCS and releases the lock L_2 and resumes its own assigned priority. Task T_4 locks the local lock L_2 on $Node2$ and begins its execution.
- On completion of execution of T_4 at t_9, task T_1 resumes its execution. T_1 and T_3 complete their executions at some later times.

Note that blocking and priority inheritance occurred at time t_3 and t_5. Although DPCP introduces new forms of blocking, Rajkumar [7], has shown that the blocking is finite and that DPCP prevents deadlocks.

5.1.4 Affected Set Priority Ceiling Protocol

This section describes the *Affected Set Priority Ceiling Protocol (ASPCP)*, which uses the affected set semantics presented in Section 5.1.1 of each method of an object to determine the compatabilities of the methods of the object, which inturn establishes the priority ceilings for each method.

Using affected set semantics, a critical section requires a method lock. Thus ASPCP assigns *conflict priority ceiling* to each method of each object:

Definition 3 (Conflict Priority Ceiling) *The conflict priority ceiling of a method m is the priority of the highest priority transaction that will ever lock a method that is* not *compatible with* *method m; where compatability is defined by affected set semantics.*

In order to determine the priority ceilings used in ASPCP, the following four sub-steps to Step 1 in Section 5.1.2 must be performed:

1a Determine the read/write affected sets for each method.

1b Determine the compatabilities of the methods using the affcted sets.

1c Determine the highest priority transaction that will access each method.

1d Calculate the priority ceiling for each method using the information from Steps 2 and 3.

At run-time, the priority ceilings are used the same way as in original PCP: The ASPCP allows a task T to receive a lock on a method *if and only if* the priority of task T is strictly higher than the conflict priority ceiling of locks held by all other task.

5.1.5 Distributed Affected Set Priority Ceiling Protocol

To increase the concurrency of the task method calls in a distributed system, DASPCP incorporates DPCP with ASPCP. The DASPCP copies all characteristics of the DPCP except the resource access control protocol at a processor level. While under DPCP each processor runs PCP on the GCS's, the set of application tasks, and the set of global and local semaphores bound to the processor, the DASPCP uses ASPCP.

The DASPCP uses the same definition of priority ceiling as the ASPCP, definition 3. The DASPCP also uses the DPCP priority assignment so that GCS's execute at the priority of the requesting task plus base priority ceiling P_G, of the system.

Example of DASPCP

The following example illustrates the application of DASPCP and demonstrates an increased concurrency compared to application of DPCP. Here we consider the system of tasks identical to one described in Example 5.1.3. Also we have the same databases, but instead of associating a lock with each database we provide one lock for each method of a database.

Object O_{track1}				
Method	read_speed	read_depth	write_speed	write_altitude
read_speed	Yes	No	Yes	Yes
write_speed	No	No	Yes	Yes
read_altitude	Yes	Yes	Yes	No
write_altitude	Yes	Yes	No	No

Object O_{track2}			
Method	read_speed	read_depth	write_speed_depth
read_speed	Yes	Yes	No
read_depth	Yes	Yes	No
write_speed_depth	No	No	No

Table 5.2: Affected Set Compatabilities in Example Objects

The priority ceilings of each lock, and the normal execution priority of each critical section thread are listed in Tables 5.3 and 5.4.

Normal Execution Priorities of Methods		
Task	**Method**	**Execution Priority**
T_1	$O_{track1} \rightarrow read\_speed$	$1 + 4 = 5$
	$O_{track2} \rightarrow read\_speed$	1
T_3	$O_{track1} \rightarrow write\_speed$	$3 + 4 = 7$
	$O_{track1} \rightarrow write\_altitude$	3
T_4	$O_{track1} \rightarrow read\_altitude$	$4 + 4 = 8$
	$O_{track2} \rightarrow read\_depth$	4

Table 5.3: Execution Priorities in DASPCP Example

Following example demonstrates the sequence of events in our system under DASPCP, illustrated in Figure 5.3:

- At time t_0, task T_1 arrives on $Node2$ and begins its execution. Similarly, task T_3 begins

Object O_{track1}				
Method	read_speed	read_depth	write_speed	write_altitude
Highest Priority Transaction	T_1	T_4	T_3	T_3
DASPCP Priority Ceiling	$3+4=7$	3	$3+4=7$	$4+4=8$
DPCP Priority Ceiling	$4+4=8$			

Object O_{track2}			
Method	read_speed	read_depth	write_speed_depth
Highest Priority Transaction	T_1	T_4	T_2
DASPCP Priority Ceiling	2	2	4
DPCP Priority Ceiling	4		

Table 5.4: Priority Ceilings in DASPCP Example

execution on $Node1$.

- At time $t1$, task T_1 gets a local lock $O_{track2} \rightarrow$ *read_speed* on $Node2$ and begins execution of LCS at its normal execution priority of $P_1(1)$. Task T_3 get the global lock $O_{track1} \rightarrow$ *write_speed* on $Node1$ and begins execution of LCS at its normal execution priority of $P_3 + P_G(3+4=7)$.

- At time t_2, task T_4 arrives on $Node2$ and preempts T_1. Task T_3 continues its execution of GCS.

- At time t_3, task T_4 requests the global lock on $O_{track1} \rightarrow$ *read_altitude*. Since its GCS's priority, $P_4 + P_G(4+4=8)$, is higher than the priority ceiling of $O_{track1} \rightarrow$ *write_speed*, $P_3 + P_G(3+4=7)$, it gets lock on $O_{track1} \rightarrow$ *read_altitude* and preempts T_3's GCS. Task T_1 continues the execution of its LCS at $Node2$.

- At time t_4, task T_4 completes the execution of its GCS and releases the global lock on $O_{track1} \rightarrow$ *read_altitude*. Task T_3 resumes the execution of its GCS at $O_{track1} \rightarrow$ *write_speed*. Task T_4 requests a local lock $O_{track2} \rightarrow$ *read_depth*. Since its priority, $P_4(4)$, is higher than the priority ceiling of $O_{track2} \rightarrow$ *read_speed*, $(PC = 2)$, it gets lock on $O_{track2} \rightarrow$ *read_depth*

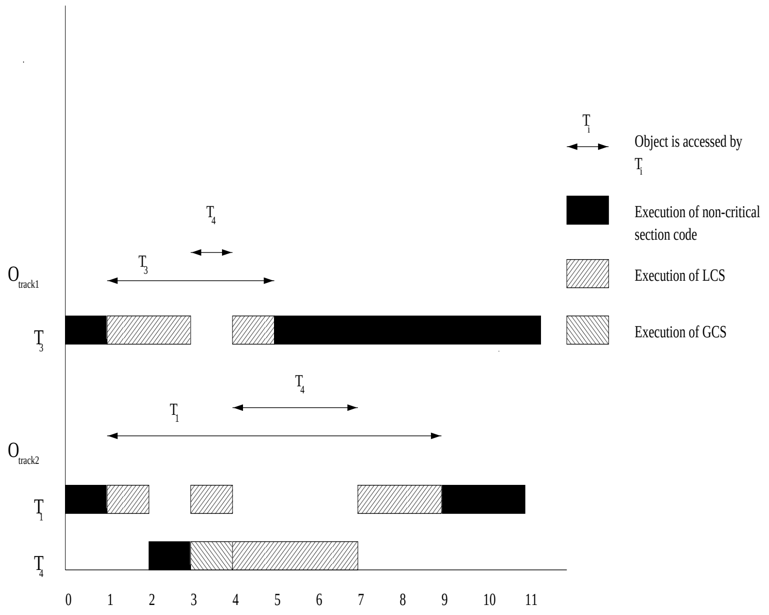


Figure 5.3: Time diagram for task system described in Example 5.1.5

and preempts T_1.

- At time t_5, task T_3 completes the execution of its GCS. No changes on $Node2$.

- At time t_7, task T_4 completes its execution, as well as the execution of its LCS with $O_{track2} \rightarrow read\_depth$ and releases the lock. T_1 resumes its execution of LCS on $O_{track2} \rightarrow read\_speed$ on $Node2$. T_1 and T_3 complete their executions at some later times.

The main advantage of the DASPCP compared to the DPCP may be seen in Figure 5.3 and two considered sequences of events: under DASPCP there were no blocking, while running it under DPCP, T_4 was blocked twice, once for a global and once for a local resource.

To conclude the discussion of the DASPCP we state its main properties, we refer the reader to [14] for the proofs of these properties.

- Under DASPCP deadlocks are avoided.

- Maximum blocking time is finite under DASPCP.

- Introduction of DASPCP never can decrease concurrency of the system in comparison with DPCP.

5.2 DASPCP for RT CORBA and PERTS

DASPCP, discussed in 5.1.5, developed at University of Rhode Island, exploits the semantics of object oriented paradigm. Affected Set semantics use method-level locking, where a task locks a particular method on an object. Method locking is of finer granularity than exclusive locking, where the task locks the entire object exclusively, thereby preventing other tasks in the system to simultaneously access methods of the object that do not affect the logical consistency of the object's data.

Since RT CORBA applications are primarily comprised of clients and servers each being implemented as *objects* and *methods* on those objects, the applications can take advantage of a superior protocol such as the DASPCP. By Examples 5.1.3 and 5.1.5 it is clear that DASPCP lowers the priority ceilings of resources in distributed object-oriented systems than the existing Priority Ceiling protocols. Lower the priority ceiling of a resource, greater is the potential concurrency among the tasks accessing the resource. Higher concurrency automatically increases the chances of the tasks meeting their their timing constraints, thereby, increasing the chance of the of the system being schedulable. Hence DASPCP is an ideal resource control protocol for RT CORBA systems.

DASPCP needs to be incorporated in the PERTS analysis of RT CORBA systems. Prior to DASPCP, CORBA servers were modeled as PERTS resources. With DASPCP, particular methods of a CORBA server (not the entire server) has to be modeled as PERTS resources and each method needs to have its own priority ceiling based on other methods of the server with which it conflicts. In the next chapter we consider the implementation issues and discuss in greater detail modifications to PERTS in lieu of Priority Mapping and DASPCP.

Chapter 6

Implementation

This chapter presents the implementation of Lowest Overlap First Priority Mapping algorithm and DASPCP in PERTS. The implementation was done in C++ on a Sun Sparc5 workstation running Sun Microsystems' Solaris 2.5 operating system. Throughout this chapter we will illustrate the new features with actual screenshots from the new PERTS version 3.2.1 and provide comparitive screenshots from PERTS 3.2 wherever necessary.

The main limitations of PERTS 3.2 are:

1. Lack of support for mapping global priorities of tasks to local priorities on their respective nodes.

2. There is no interface with RT CORBA Scheduling Service.

3. Need for a better resource access protocol such as DASPCP.

We hope to eliminate these limitations with the implementation strategy described below.

6.1 Implementation Plan

To incorporate the new features, described in the previous chapters, we address three different issues each with respect to Priority Mapping, DASPCP, and interface to RapidSched:

1. Enhancements to PERTS Resource Graph Editor, Graphic User Interface (GUI).

2. Enhancements to the Schedulability Analyzer.

3. Enhancements to the PERTS output.

In the following sections we elaborate each of these with regards to the priority mapping and DASPCP.

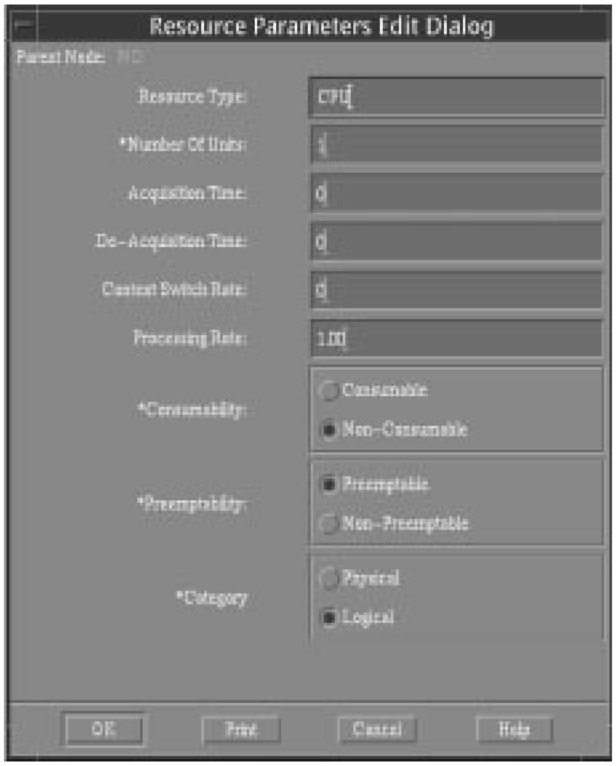
6.2 Priority Mapping

6.2.1 Enhancements to the Resource Graph Editor

Priority Information: Input

PERTS Resource Graph Editor, Section 2.2, allows a user to graphically input information about resources in a real-time system. Examples of resources are the CPU which carries out the workload of the tasks, Databases, Files, Printers and such others. Each of these resources belongs to an entity called a *Node* in PERTS. The resource graph (RG) defines certain parameters to be associated with the resources as illustrated in the Edit Parameters Dialog box, Figure 6.1. The Dialog box is activated when a user selects *Edit Parameters* from the options listed in the pull down menu associated with the Edit option of the RG and clicks on a desired resource. Notice that it does not allow a user to input the number of available priorities.

In general, priority queues can be associated with any resource and are not necessarily restricted to the operating system alone (running on the node's CPU). Priority queues commonly exist on



Parent Node: PIC

Resource Type:	CPU
*Number Of Units:	1
Acquisition Time:	0
De-Acquisition Time:	0
Context Switch Rate:	0
Processing Rate:	1.00
*Consumability:	Consumable [✓ Non-Consumable]
*Preemptability:	[✓ Preemptable] Non-Preemptable
*Category:	Physical [✓ Logical]

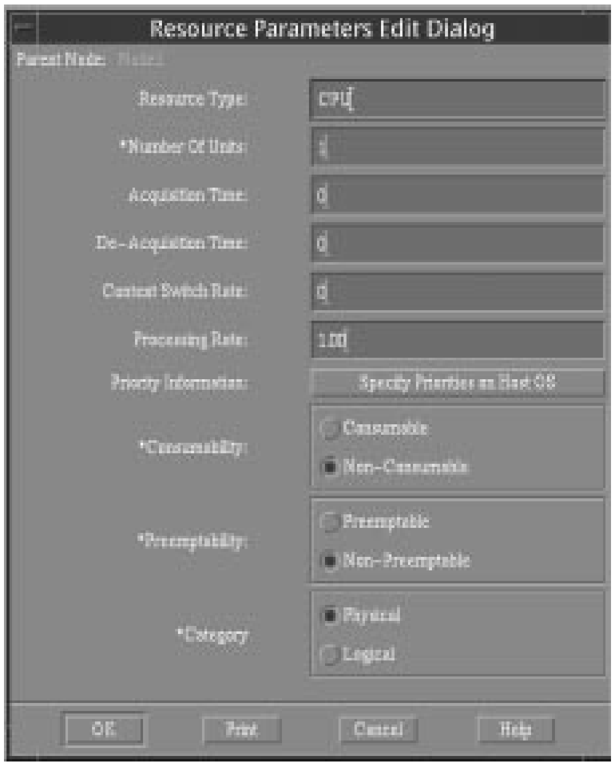
Figure 6.1: Edit Parameter Dialog Box in PERTS 3.2

Network Switches, Backplane hardware, Hardware Bus etc. Keeping this generality in view, we have allowed input of number of available priorities for any PERTS resource and not just the CPU. Figure 6.2 illustrates the modified Edit Parameter Dialog box. Notice the addition of a button associated with the new paramter *Priority Information.*

Priority Information Edit Dialog

By clicking on the *Priority Information* button in the Figure 6.2 the Dialog box correponding to the input of Priority Information 6.3 for a particular resource is activated.

Priorities are defined by *Range(s) of Numbers* and *Direction.* **Ranges** are specified by specifying a *Start of Range* and *End of Range.* All numbers within this range, inclusive of the range boundaries are considered to be valid priorities associated with the particular resource. The important design



Resource Parameters Edit Dialog

Parent Node: [illegible]

Resource Type:	CPU
*Number Of Units:	1
Acquisition Time:	0
De-Acquisition Time:	0
Context Switch Rate:	0
Processing Rate:	100
Priority Information:	Specify Priorities on Host OS
*Consumability:	Consumable [✓ Non-Consumable]
*Preemptability:	Preemptable [✓ Non-Preemptable]
*Category	[✓ Physical] Logical

Figure 6.2: Edit Parameter Dialog Box in PERTS 3.2.1

decision to input priorities as block of numbers is consistent with reality where certain priorities are *reserved* by the system and hence not available to the application programs. This naturally leads to possibility of multiple ranges available for applications. Hence we have constructed the dialog box in such a way that a user can input multiple ranges and PERTS stores these ranges in a list data structure. A range is an instance of a `PriorityRange Class` which we defined for this purpose.

```
   class PriorityRange :  public PObject {
public:
int start, end;
PriorityRange(int from, int to) {start = from; end = to;};
PriorityRange(){};
};
```

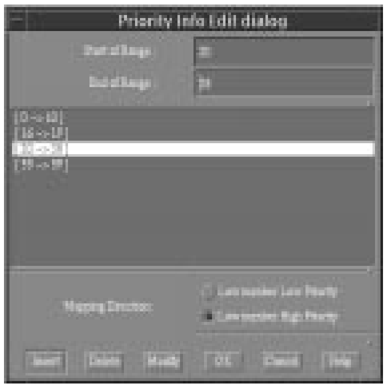


Figure 6.3: Priority Information Edit Dialog Box in PERTS 3.2.1

Direction is an important attribute which is necessary to capture the two conventions adopted by systems:

- *Lower the number higher the Priority.* Eg. Solaris Operating System in which 0 is the highest priority and 59 is the lowest priority.

- *Lower the number lower the Priority.* Eg. Chorus Operating System in which 0 is the lowest priority and 255 is the highest priority.

To input the Direction, the dialog box is provided with a toggle button.

To summarize, the fields associated with the Priority Information Edit Dialog box are:

- **Start of Range:** specifying the beginning of a range.

- **End of Range:** specifying the end of a priority range.

- **Scrolled List:** of all previously specified ranges.

- **Direction:** A toggle to specify direction of priorities.

The Priority Information Dialog Window contains six buttons:

- **Insert** - inserts new block of numbers into the list of priority ranges;

- **Delete** - deletes selected (one or more) range from the list;
- **Modify** - modifies the parameters of a specified range;
- **OK** - saves current list of priority ranges in the increasing order of start of range;
- **Cancel** - cancels an Insert, Delete or Modify operation and closes the Dialog;
- **Help** - pops up the window describing features of the Priority Information Dialog Window.

The complete description of a resource is encapsulated in the `Class ResourceNode`. We have added two new members to this class:

```
List *priorityList;
int mappingDirection;
```

The first member `priorityList` is a pointer to an object of class `List`, containing the list of pointers to the objects of class `PriorityRange`. The second new member `mappingDirection` is of type `int` and takes a value of 0 corresponding to Lower number being lower priority or 1 corresponding to Lower number being higher priority.

Along with these members, we have introduced a method:

```
void DefinePriorityList(int start_range, int end_range);
```

The method inserts a new set of priority range into the `priorityList`, sorting it in the increasing order of *Start of Range.*

Priority Information: Open, Read and Save

To maintain all the existing GUI operations after the addtion of new data members, we have modified/changed methods of some classes which we describe very briefly here.

To **Save/Save As** the priority information along with all other attributes of a resource into a textual file, we have introduced two new keywords PRIO_RANGE and MAPPING_DIR corresponding to the priority ranges and mapping direction associated with the priority infromation. The save operation is performed by the method:

```
static void save_resource_node(GResourceNode *node, int indent, ofstream &fout);
```

in the file `gresgraph.cc`

To **Open/Reopen** a resource graph with the new attributes for resources, we have modified the resource graph Compiler to read the new fields from the text file description. Methods `cResourceNode` and `cResourceParameter` of the class `RG_Compiler` have been modified to recognize the new keywords PRIO_RANGE and MAPPING_DIR.

A new method:

```
int RG_Compiler::Priority_List(ResourceNode *resrc);
```

has been introduced to construct the `priorityList` after reading the text file description of a resource graph node.

The method:

```
static void copy_priority_list(ResourceNode *node);
```

has been added to the file `gresgraph.cc` to open a resource graph.

To incorporate the new parameters into the PERTS report, **Generate Report** command of the resource graph, appropriate changes have been made to the methods, `generate_report` and `generate_report_node` in the file `gresgraph.cc`.

Appropriate descriptions have been added to the file `help.h` and `help.cc` to reflect the addtion of new attributes to a resource.

6.2.2 Enhancements to Schedulability Analyzer

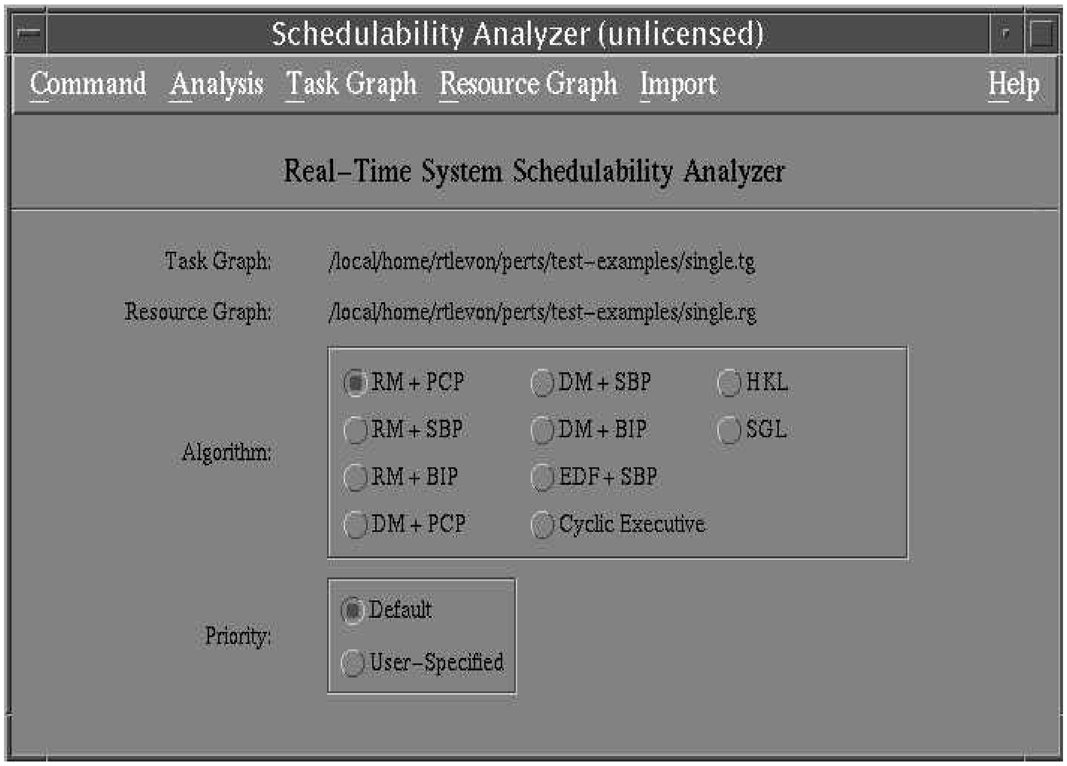
The Schedulability Analyzer described in Section 2.3, is the essence of PERTS. PERTS Engine is comprised of the various internal classes that belong to the analyzer. Changes to the analyzer can be classified into 2 categories:

- Changes to the Analyzer GUI.

- Changes to the Engine.

GUI Enhancements

The Scheduler GUI shown in Figure 6.4, allows a user to specify the tasks and resources in the system through their respective text filenames, and carryout the analysis of the system. The enhanced version also allows the user to select a type of Priority Mapping to be carried out, as shown in the Figure 6.5. Currently, we provide two alternatives:



Real–Time System Schedulability Analyzer

Task Graph: /local/home/rtlevon/perts/test–examples/single.tg

Resource Graph: /local/home/rtlevon/perts/test–examples/single.rg

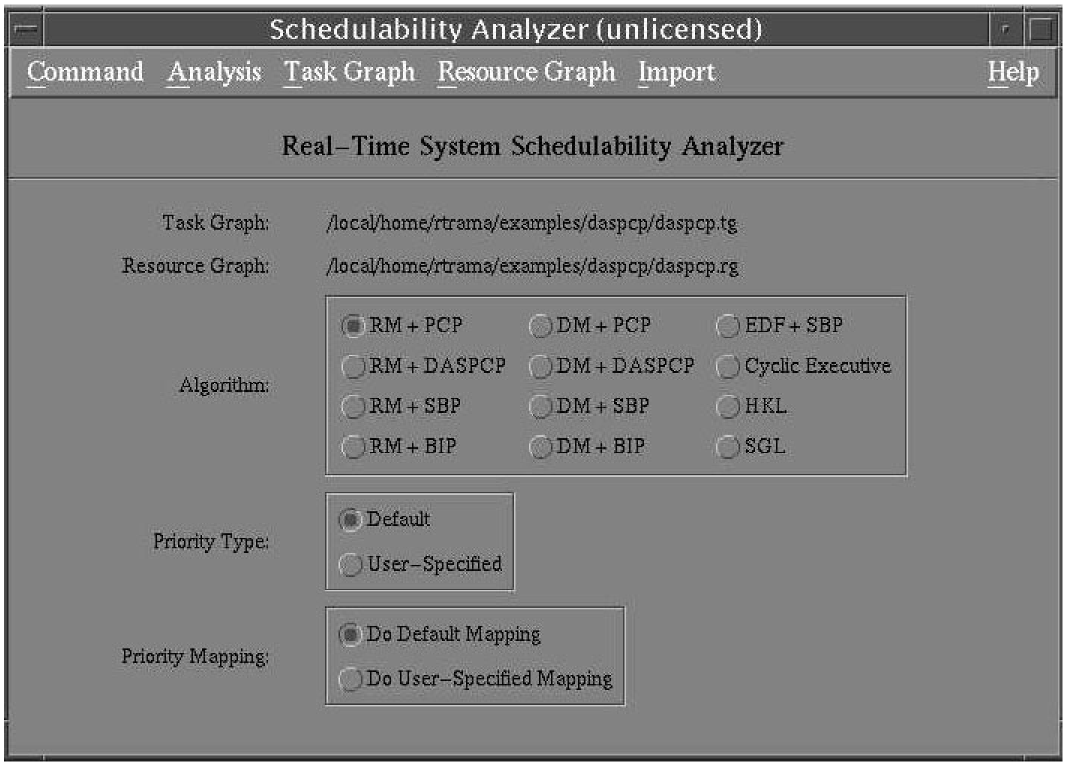
Algorithm: [✓ RM + PCP] RM + SBP RM + BIP DM + PCP DM + SBP DM + BIP EDF + SBP Cyclic Executive HKL SGL

Priority: [✓ Default] User–Specified

Figure 6.4: PERTS 3.2 Scheduler

- **Lowest Overlap First Priority Mapping:** described in Chapter 4.

- **Default Mapping:** in which local priority of each task is same as its global priority.



Real–Time System Schedulability Analyzer

Task Graph: /local/home/rtrama/examples/daspcp/daspcp.tg

Resource Graph: /local/home/rtrama/examples/daspcp/daspcp.rg

Algorithm: [✓ RM + PCP] DM + PCP EDF + SBP RM + DASPCP DM + DASPCP Cyclic Executive RM + SBP DM + SBP HKL RM + BIP DM + BIP SGL

Priority Type: [✓ Default] User–Specified

Priority Mapping: [✓ Do Default Mapping] Do User–Specified Mapping

Figure 6.5: PERTS 3.2.1 Scheduler with Option to Select a Priority Mapping

The function:

```
static void _sched_priority_mapping_type_select(Widget w, XtPointer client_data, void
*cbs);
```

has been added to the file `scheduler.cc` to provide a choice of mapping algorithms.

A new member, `MappingType _mapping_type` is added to the class `sa_system.h` to store the type of mapping.

The method:

```
MappingType SA_System::mapping_Type();
```

returns the type of priority mapping selected by the user.

Changes to PERTS Engine

Most of the additions to the PERTS engine are to the classes `class SA_System`, `class SA_Node` and `class SA_Task`. In this section we briefly describe changes to these modules under two categories:

- New Data members.
- New Methods

We consider each of these categories for the above mentioned classes.

class SA_System:

Class `SA_System` provides the abstraction of a system. A system represents a collection of nodes where tasks execute governed by a scheduling algorithm and a control access protocol. This class is intended to serve as the root of the system class hierarchy implementing several scheduling algorithms and resource access protocol.

The class `SA_System` is a generic system that does not implement any global scheduling algorithm or global control access protocol. All the nodes in the system are `SA_Nodes`, which implement a generic priority-driven scheduling algorithm without any control access protocol.

New Data members:

```
    Inverse_Priority_List _all_Periodic;
// List of periodic tasks (including servers)
// in increasing priority assigned to a node
```

Recall from the description of the Lowest Overlap First algorithm in Section 4.3.1, that the algorithm requires all the tasks in the system to be arranged in increasing priority order. Since the data type `Priority_List`, orders tasks in decreasing order of priorities, we introduce a data type `Inverse_Priority_List` that orders tasks in the increasing priority order.

Note: The Lowest Overlap First mapping algorithm is also referred to as *Compact Squeeze* algorithm in this literature.

```
   MappingType _mapping_type;

   enum MappingType {DEFAULT_PRIORITY_MAPPING, LOWEST_OVERLAP_FIRST };
```

The member `_mapping_type` was described earlier. The data type `MappingType` defines two possible choices for the mapping as already mentioned.

New Methods:

We define a new class called `Task_Node` which contains a pointer to an instance of class `SA_Task` and also the Priority associated with that task object, in the file `sa_task.h`

```
   Task_Node * first_all_Periodic() {return (Task_Node *) _all_Periodic.First();}

Task_Node * next_all_Periodic() {return (Task_Node *) _all_Periodic.Next(); }

Task_Node * next_all_Periodic(Task_Node *current) {return (Task_Node *)

_all_Periodic.Next(current); }
```

The above methods are iterators for the list `_all_Periodic`.

```
   BOOLEAN compact_Squeeze();
// PURPOSE : The schedulabilty analyzer assigns global priorities for
// all the tasks in the system.  If the system is schedulabale, these
// global proirities have to be mapped to the priorities available
// on the task's host operating system for the system to meet scheduling
// criteria.  This functions maps global priorities to local priorities
// based on the Lowest Overlap First algorithm.

   BOOLEAN backtrack();
// PURPOSE: Bactrack to find another mapping in case of a failure.

   int compact_Squeeze_Init();
// PURPOSE: Initialization steps for the compact squeeze algorithm.  Set
// counters on all nodes and arrange tasks and servers in the system in
// increasing priority order.

   void compact_Squeeze_Finish();
// PURPOSE: Perform one-to-one mapping from temporary local priorities
// assigned to tasks by the compact_squeeze to the actual priorities
// specified by the user on each node.

   BOOLEAN priorities_Specified();
// PURPOSE: Return TRUE if priority information is specified on all nodes
// in the system FALSE otherwise.
```

Class SA_Node

Class SA_Node provides the abstraction of a node in the Schedulability Analyzer System. A node is an schedulable entity of the system. This class encapsulates the scheduling algorithm and control access protocol used to schedule the task set in the node. The class SA_Node assumes a priority-driven scheduling algorithm and no control access protocol. The second assumption implies that resource conflicts are not considered.

New Data Members:

```
   int _counter;
// PURPOSE: If the counter is positive, the number of squeezes yet to
// to be performed by the compact squeezes algorithm to map global
// priorities of tasks on this node to the local priority on the host
// system.

   int _num_Local_Priorities;
// PURPOSE: The total number of available local priorities supported
// by the host system.

   int _highest_non_empty_prio;
// PURPOSE: Most recently allocated task priority.

   int _empty_local_prio;
// PURPOSE: Next available local task priority to be considered by the
// compact squeeze algorithm.

   int _highest_non_empty_gcs_prio;
// PURPOSE: Most recently allocated GCS priority.

   int _empty_local_gcs_prio;
// PURPOSE: Next available local GCS priority.

   int _num_tasks;
// PURPOSE: Number of periodic Tasks and Servers on this node.

   int _num_gcs;
// PURPOSE: Number of GCS on this node.

   List _gcs_List;
// PURPOSE: List of all global critical sections assigned to this
// node in decreasing priority order.
```

New Methods:

```
   void generate_configuration_file();
// PURPOSE: Generate an output file with a list of Global to Local
// Priority Mappings for all tasks, servers, GCSs on this node.

   void dec_Counter() {_counter-- ;}
// PURPOSE: Decrement the Squeeze counter on this node.

   int counter() const { return _counter;}
// PURPOSE: Return the value of the squeeze counter on this node.

   int num_local_priorities() const { return _num_Local_Priorities;}
// PURPOSE: Return the total number of available priorities on
// this node.
```

The next 3 methods are iterators for the `_gcs_list` on this node.

```
   Critical_Section *first_GCS() { return (Critical_Section *) _gcs_List.First();}
Critical_Section *next_GCS() { return (Critical_Section *) _gcs_List.Next();}
Critical_Section *next_GCS(Critical_Section * current) { return (Critical_Section *)
_gcs_List.Next(current);}

   virtual BOOLEAN test_Schedulability(SA_Task *inTask) {}
// PURPOSE: Perform the schedulability test of inTask on the node.
// This operation should be redefined in derived
// classes implementing scheduling algorithms and/or control access
// protocols.  Required by compact squeeze priority mapping algorithm.

   BOOLEAN Squeeze(SA_Task *);
// PURPOSE: Map Global priority of a task to a local priority
// according to the Lowest Overlap First Algorithm.

   BOOLEAN Squeeze(Critical_Section *);
// PURPOSE: Map Global priority of a GCS to a local priority
// according to the Lowest Overlap First Algorithm.

   void assign_Local();
// PURPOSE : Assign local priorities for tasks on this nodes based on
// temporary priorities assigned by compact squeeze algorithm.

   int set_Counter();
// PURPOSE : Initialize the counter for compact squeeze algorithm.

   BOOLEAN priorities_Specified();
// PURPOSE : Check to see if the priority information is specified on
// this node.
```

class SA_Task:

Class `SA_Task` is an abstraction of a Task. It keeps all the schedulability information concerning a PERTS task.

New Data Members:

```
   Priority _local_Priority;
// PURPOSE: The priority assigned to the task after mapping global
// priority to local priority.

   Priority _global_Priority;
// PURPOSE: The global priority of the task.

   Priority _corba_Priority;
// PURPOSE: The CORBA priority of the task.  A CORBA Priority is between
// 0 and 32K with 0 being the lowest priority.

   BOOLEAN _squeezed;
// PURPOSE: Whether this task caused a overlap or not, when assigned a
// local priority.
```

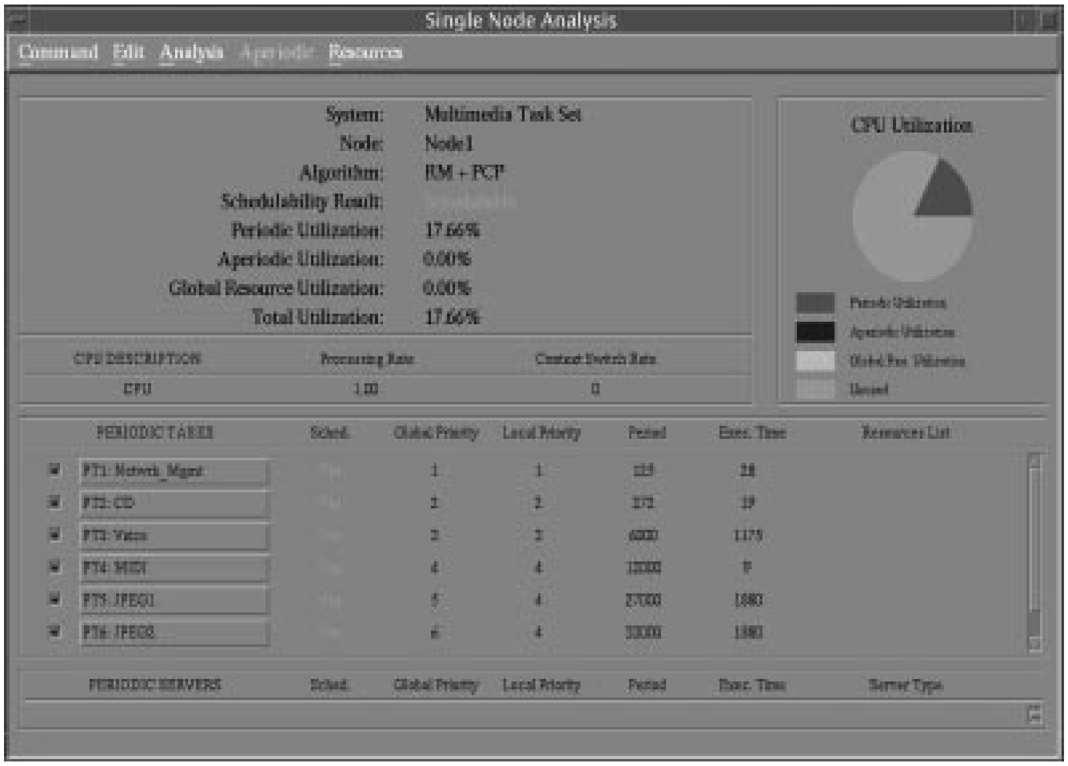
New Methods:

```
   Priority local_Priority() const return _local_Priority;
// PURPOSE: Returns the local priority of the task.

   Priority global_Priority() const return _global_Priority;
// PURPOSE: Returns the global priority of the task.

   Priority corba_Priority() const return _corba_Priority;
// PURPOSE: Returns the CORBA priority of the task.

   BOOLEAN squeezed() return _squeezed;
// PURPOSE: Returns TRUE if the task's local priority is overlapping with
// another task's local priority.
```

Similar additions have been made to the class, `Critical_Section`.

6.2.3 Enhancements to PERTS Output

The GUI classes that display graphical output as shown in Figure 6.6 have been appropriately modified to display the additional task attribute, its *Local Priority*, which is same as the priority assigned by a mapping algorithm.



System:	Multimedia Task Set
Node:	Node1
Algorithm:	RM + PCP
Schedulability Result:	[illegible]
Periodic Utilization:	17.66%
Aperiodic Utilization:	0.00%
Global Resource Utilization:	0.00%
Total Utilization:	17.66%

CPU DESCRIPTION	Processing Rate	Context Switch Rate
CPU	1.00	0

PERIODIC TASKS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Resources List
PT1: Netwrk_Mgmt	[illegible]	1	1	125	28	
PT2: CD	[illegible]	2	2	272	19	
PT3: Video	[illegible]	3	3	6000	1175	
PT4: MIDI	[illegible]	4	4	12000	9	
PT5: JPEG1	[illegible]	5	4	27000	1880	
PT6: JPEG2	[illegible]	6	4	33000	1880	

PERIODIC SERVERS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Server Type

Figure 6.6: Output of PERTS 3.2.1 with Local Priorities of Tasks

To incorporate the new parameters into the PERTS report, **Generate Report** command of the Scheduler, appropriate changes have been made to the methods, `sn_generate_report_file` in the file `sn_analysis.cc`.

6.3 DASPCP

Introduction of the DASPCP slightly modifies the mapping of the RT CORBA to PERTS, described in Chapter 5. Namely, PERTS resources do not represent whole CORBA servers, instead they represent the methods of the servers. The ability to specify a set of conflicting methods, Section 5.1.1 (resources) for a particular method has to be added to the PERTS Resource Graph Editor and the revision of the calculation of the priority ceiling in PERTS Schedulability Analyzer as definied in the definition 3.

6.3.1 Enhancements to Resource Graph Editor

Figure 6.7, shows the Pull Down Menu options provided by PERTS 3.2 for resources in the system. We have added additional features, shown in Figure 6.8 that allows a user to specify incompatible methods. Incompatability is represented by *double headed dashed orange arrows* in the resource graph.

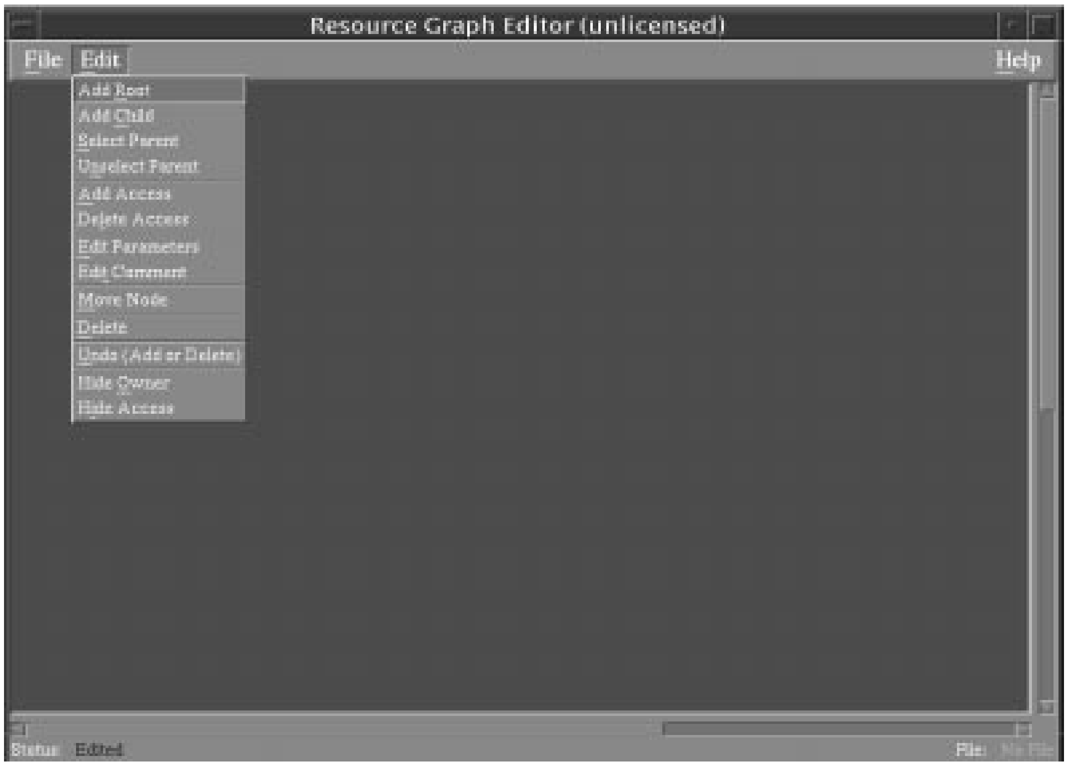


Figure 6.7: Edit Menu Options in PERTS 3.2 Resource Graph Editor

Conflict Resources: Input

The features provided are:

- **Select** any resource (*a method* of an object)
- **Unselect** an already selected resource.

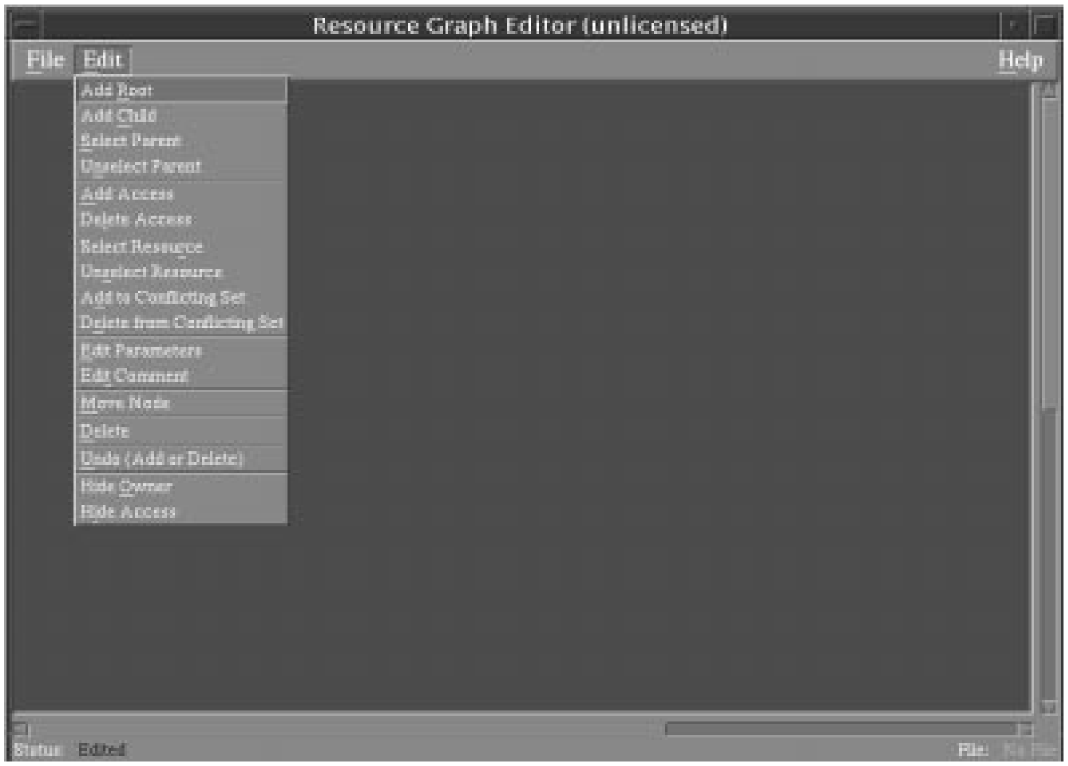


Figure 6.8: Edit Menu Options in PERTS 3.2.1 Resource Graph Editor

- **Add** to the conflicting set of resources (incompatability set of the method) by subsequently clicking on another resource under the same node (because, Objects can reside on a single node and cannot spawn node boundaries).

- **Delete** from the conflicting set of resources by subsequently clicking on a resource that is a part of the conflicting set for this resource.

Methods to do the above mentioned functionality have been added to the class `GResourceGraph`. They are:

```
   void SetSelectConflict(GResourceNode *node);
// PURPOSE: Select/Unselect a resource node.

   void AddConflictSet(GResourceNode *node);
// PURPOSE: Add two resource nodes to each other's conflict set.

   void DelConflictSet(GResourceNode *node);
// PURPOSE: Remove two resource nodes from each other's conflict set.
```

We have added a new data member:

```
List *conflictList;
```

to the class `ResourceNode`. The data member `conflictList` is a pointer to an object of class `List`, containing the list of pointers to the objects of class `ResourceNode`.

We have also provided methods to manipulate this member. They are:

```
   int IsInConflictList(ResourceNode *node);
// PURPOSE: Check if node is already in the conflict
// list of this node.

   void DelConflictNode(ResourceNode *node);
// PURPOSE: Remove node from the conflict list of this node.

   void AddConflictNode(ResourceNode *node);
// PURPOSE: Add node to the conflict list of this node.
```

Figure 6.9, shows resource graph with double headed arrows denoting resources conflicting each other.

Conflict Resources Information: Open, Read and Save

To maintain all the existing GUI operations after the addtion of new data members, we have modified/changed methods of some classes which we describe very briefly here.

To **Save/Save As** the conflicting resources information along with all other attributes of a

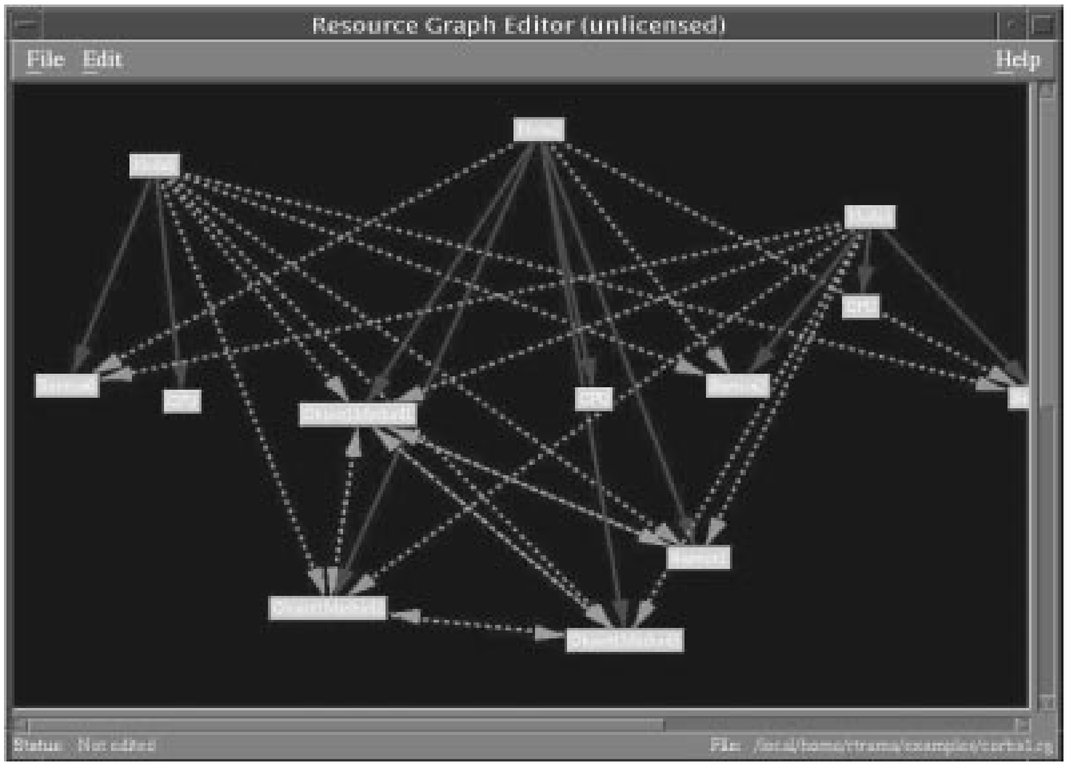


Figure 6.9: PERTS 3.2.1 Resource Graph Editor with Ability to Specify Incompatible Resources (Object Methods) via double headed arrows

resource into a textual file, we have introduced a new keyword CONFLICT_RESRC_SET corresponding to the set of conflicting resources associated with the resource under consideration. The save operation is performed by the method:

```
static void save_resource_node(GResourceNode *node, int indent, ofstream &fout);
```

in the file `gresgraph.cc`

To **Open/Reopen** a resource graph with the new attribute for resources, we have modified the resource graph compiler to read the new field from the text file description. Methods `cResourceNode` and `cResourceParameter` of the class `RG_Compiler` have been modified to recognize the new keyword CONFLICT_RESRC_SET.

New methods provided to the class `RG_Compiler` are:

```
   int cResourceConflictList(ResourceNode *);
// PURPOSE: Parse the text file description of the resource graph
// to regognize the new keyword.

   ResourceNode *GetResourceNodeByName(ResourceGraph *, char *, ResourceNode *);
// PURPOSE: Returns an object reference of type ResourceNode,
// given its textual name, the node it belongs to and the
// ResourceGraph.

   int BuildConflictList(ResourceGraph *, ResourceNode *, ResourceNode *);
// PURPOSE: Constructs (or Adds if already exists) the
// conflictList for the two resources selected by the user.

   int BuildConflictList(ResourceGraph *);
// PURPOSE: Builds the conflictList for all the resources
// in the resource graph.
```

The method:

```
static void copy_conflict_access(ResourceNode *node);
```

has been added to the file `gresgraph.cc` to open a resource graph with the new attributes.

To incorporate the new parameters into the PERTS report, **Generate Report** command of the resource graph, appropriate changes have been made to the methods, `generate_report` and `generate_report_node` in the file `gresgraph.cc`.

Appropriate descriptions have been added to the file `help.h` and `help.cc` to reflect the addtion of new attributes to a resource.

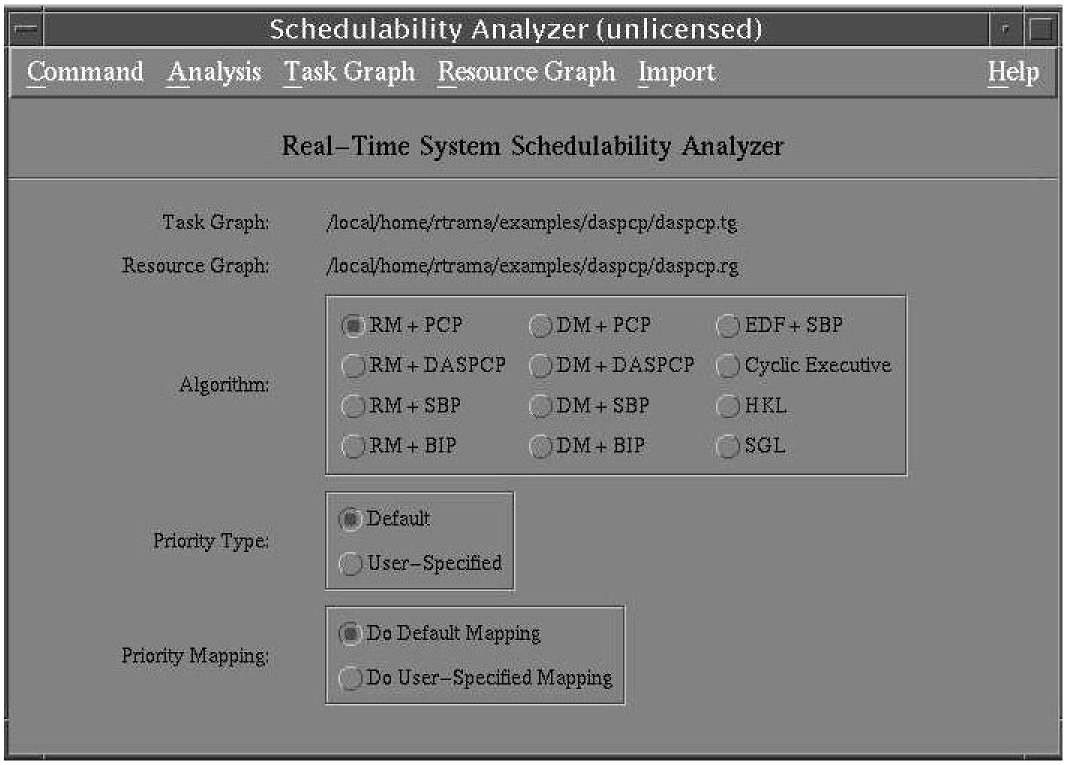
6.3.2 Enhancements to Schedulability Analyzer

The Schedulability Analyzer is described in detail in Section 2.3. As in the previous section, we present the changes to the analyzer in 2 categories:

- Changes to the Analyzer GUI.
- Changes to the Engine.

GUI Enhancements

The Scheduler GUI shown in Figure 6.4, allows a user to specify the tasks and resources in the system through their respective text filenames, and carryout the analysis of the system. The enhanced version provides DASPCP as an additional resource contention protocol. A user can now select DASPCP with Rate Monotonic (RM) or Deadline Monotonic (DM) Priority assignment mechanism. This is illustrated in the Figure 6.10.



Real–Time System Schedulability Analyzer

Task Graph: /local/home/rtrama/examples/daspcp/daspcp.tg

Resource Graph: /local/home/rtrama/examples/daspcp/daspcp.rg

Algorithm: [✓ RM + PCP] DM + PCP EDF + SBP RM + DASPCP DM + DASPCP Cyclic Executive RM + SBP DM + SBP HKL RM + BIP DM + BIP SGL

Priority Type: [✓ Default] User–Specified

Priority Mapping: [✓ Do Default Mapping] Do User–Specified Mapping

Figure 6.10: PERTS 3.2.1 Scheduler with DASPCP

The function:

```
static void _sched_algorithm_select(Widget w, XtPointer client_data, void *cbs);
```

has been added to the file `scheduler.cc` to provide a choice of DASPCP with other mechanisms.

Changes to PERTS Engine

Recall from the example 5.1.5 that the only difference between DPCP and DASPCP is the way Priority Ceiling of a resource is defined.

Since the DPCP module is already implemented in PERTS, we have retained the same framework. We have used inheritance in C++ to take advantage of the *is-a* nature of DASPCP with regards to DPCP. The following new classes have been added to PERTS.

```
class DASPCP_Node : public PCP_Node {}

class DASPCP_Resource : public PCP_Resource {}

class DASPCP_Scheduler : public PCP_Scheduler {}

class RM_DASPCP_Scheduler : public DASPCP_Scheduler {}

class DM_DASPCP_Scheduler : public DASPCP_Scheduler {}
```

We have altered the method, `void update_SA_Parameters();` in the class `DASPCP_Resource` which calculates the priority ceiling of a resource.

6.3.3 Enhancements to PERTS Output

To incorporate the new protocol into the PERTS report, **Generate Report** command of the Scheduler, appropriate changes have been made to the methods, `sn_generate_report_file` in the file `sn_analysis.cc`.

6.4 PERTS Interface to RapidSched

RapidSched, see Section 3.3.1 is an RT CORBA Scheduling Service implementation 3.3 developed at the Real-Time Research Lab in URI. It conforms to the RT CORBA 1.0 draft standard. The implementation has been designed to work closely with PERTS. With PERTS, system designers enter information about their RT CORBA clients and servers. These high level objects are automatically translated into PERTS primitives [14]. A rate-monotonic analysis then determines whether the system is schedulable.

If the system is schedulable, PERTS produces a configuaration file from which RapidSched can retrieve scheduling information at run-time, about all tasks, GCSs and resources in the system and make appropriate scheduling decisions. The file is generated automatically by the method:

```
void SA_Node::generate_configuration_file();
```

of the class `SA_Node`. For every node in the system a file is generated with the name same as the name of the node. Currently, the information in the file contains:

- Resources (specified by their names) and their corresponding *Priority Ceiling* as calculated by PERTS.
- Tasks (specified by their names), their *CORBA Priority* see Section 6.2.2 and the *Local (Run-Time) Priority* of the task on its node, which is obtained as a result of priority mapping, discussed in Chapter 4.
- The Global Critical Sections (GCSs) and their corresponding *CORBA Priority* and local priority.

Table 6.1, shows a sample PERTS output file.

In the next chapter we present some test cases for the new PERTS features and discuss the results.

PERTS Configuaration File for RapidSched		
Node Name	Node1	
Resource Name	**Priority Ceiling**	
Object1Method2	23	
Object1Method3	20	
End		
Task Name	**Corba Priority**	**Local Priority**
ApplicationC	14	251
ApplicationC_2	9	250
ApplicationC_3	3	249
End		
Extra Mappings (GCS)	**Corba Priority**	**Local Priority**
	23	255
	20	254
	20	253
End		

Table 6.1: Sample PERTS Output Configuration File for RapidSched

Chapter 7

Evaluation

So far we have been concerned with describing a problem, its solution and presenting an implementation. However, it is equally important to verify the solution once it has been implemented. In this chapter we will present some test cases for both Priority Mapping and DASPCP.

7.1 Testing Lowest Overlap First Priority Mapping Algorithm

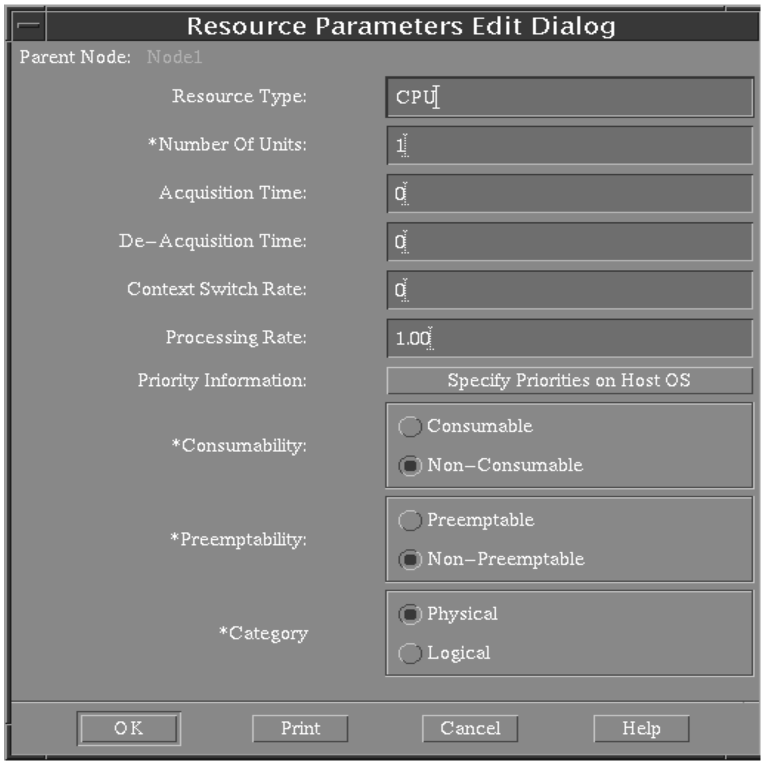
We will adopt a two track approach to test the LOF algorithm:

1. Test the Resource Graph Editor GUI changes for input of priority information.

2. Test the Implementation of the LOF algorithm.

7.1.1 Resource Graph Editor GUI Enhancements

Tests of the modified Resource Graph Editor GUI have demonstrated presence and correctness of all desired features. The Resource Parameters Edit Window, shown in Figure 7.1 contains a new parameter *Priority Information* with an associated button. The introduction of the new field did not affect others. A user can specify the Priority Information (along with previously presented in the dialog parameters). It is mandatory to specify the priority information for the resource labeled CPU, otherwise an error message is raised. By clicking on *OK* button the priority information is saved (along with other resource parameters).


Resource Parameters Edit Dialog

Parent Node: Node1

Resource Type:	CPU
*Number Of Units:	1
Acquisition Time:	0
De–Acquisition Time:	0
Context Switch Rate:	0
Processing Rate:	1.00
Priority Information:	Specify Priorities on Host OS
*Consumability:	Consumable [✓ Non–Consumable]
*Preemptability:	Preemptable [✓ Non–Preemptable]
*Category	[✓ Physical] Logical

Figure 7.1: Edit Parameter Dialog Box in PERTS 3.2.1

We have tested the *Priority Information Edit Dialog*, 7.2 which enables a user to edit the priority information fields. The dialog window properly:

- **Inserts** a new set of priority ranges into the scrolled list of ranges already specified.

- **Deletes** selected (one or more) ranges from the list.

- **Modifies** the parameters of a specified selected range.

- **Saves** the current list of priority ranges in the increasing *Start of Range* order, along with

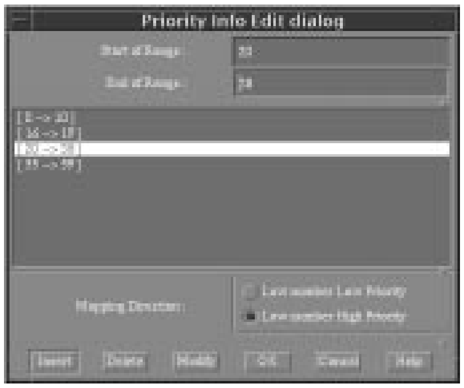


Figure 7.2: Priority Information Edit Dialog in PERTS 3.2.1

the *Direction* of priorities.

- **Closes** the dialog window and

- **Provides** the Help window, describing features of the dialog box.

The **Generate Report** command verifies that the priority information is included in the report.

The **Save** command, on a resource graph with the new feature is found to save the priority information of all the resources, along with previous parameters. This is verified by opening the text file description of the resource graph in a standard text processing editor.

Opening the saved text file description with the **Open/Reopen** command of the resource graph verifies that the saved priority information is correctly read in by the resource graph.

7.1.2 Lowest Overlap First Priority Mapping

As an example, we consider the case study that is discussed in [11]. Consider a high-speed network that connects one or more multimedia servers to multimedia workstations. We assume that the traffic consists of a mixture of video, audio, voice, MIDI, and large file transfers in addition to periodic and aperiodic network management messages. The network is scheduled using a fixed priority algorithm, such as rate monotonic algorithm. The multimedia task set is summarized in Table 7.1. More detailed discussion of the development of the task set can be found in [11].

Multimedia System					
Task	**Type**	**Natural Priority**	**C (usecs)**	**T (usecs)**	**Local Priority**
T_1	Network Mgmt.	1	28	125	1
T_2	CD	2	19	272	2
T_3	Voice	3	1175	6000	3
T_4	MIDI	4	9	12000	4
T_5	JPEG1	5	1880	27000	4
T_6	JPEG2	6	1880	33000	4
T_7	File Transfer	7	5000	100000	4

Table 7.1: Multimedia Experimental Task Set

The analysis is carried out as follows: assuming an unlimited number of priority levels, natural priorities are assigned to the tasks in the multimedia task set according to rate monotonic algorithm as shown in table 7.1. Note that smaller numbers indicate higher priority. However the network described above is limited to four priority levels and according to equation 4.2, there are N = 20 different mappings that are possible for this task system. We are only interested in a mapping that results in a schedulable system. Figure 7.3, is a screen shot of PERTS analysis of this task set. The local priority in the figure refers to the priority assigned by the Lowest Overlap First Priority Mapping algorithm.

7.2 Testing DASPCP Implementation

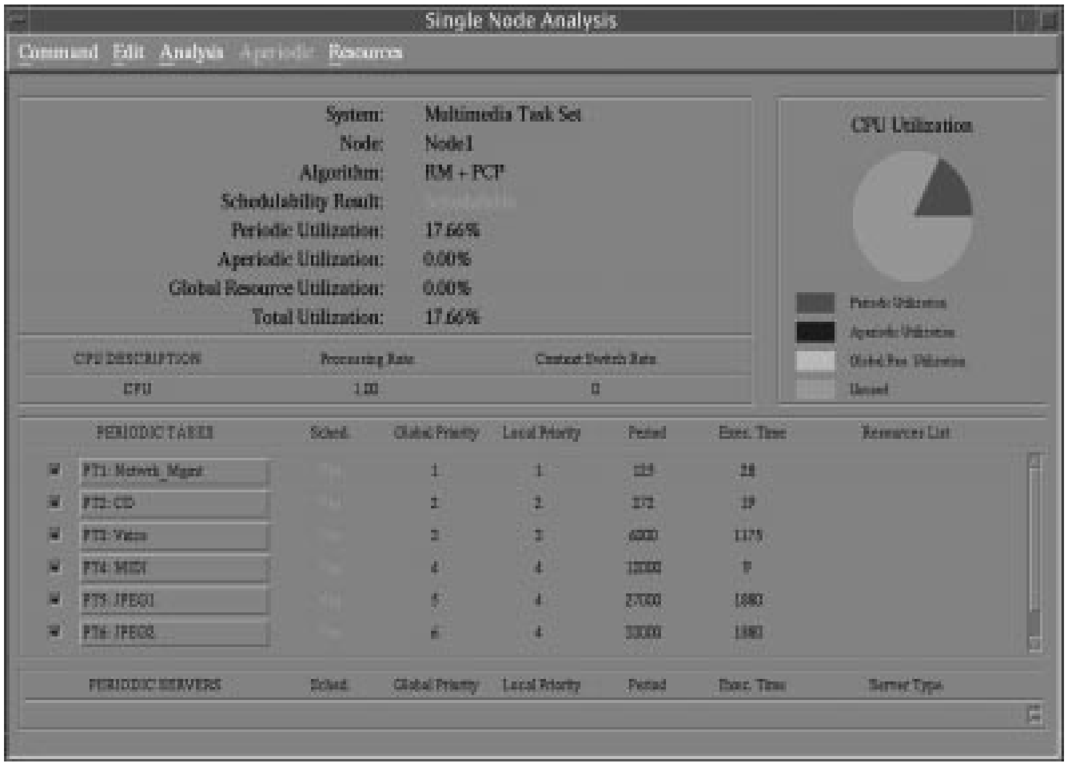
We will adopt the same two track approach to test DASPCP:

1. Test the Resource Graph Editor GUI changes for input of conflicting resources information.

2. Test the PERTS Calculation of Priority Ceilings under DASPCP

7.2.1 Resource Graph Editor GUI Enhancements

Tests of the modified Resource Graph Editor GUI have demonstrated presence and correctness of all desired features. The pull down menu under Edit option of the resource graph, shown in Figure 7.4 contains new choices: **Select a Resource**, **Unselect a Resource**, **Add to Conflict Set**



System: Multimedia Task Set
Node: Node1
Algorithm: RM + PCP
Schedulability Result: [illegible]
Periodic Utilization: 17.66%
Aperiodic Utilization: 0.00%
Global Resource Utilization: 0.00%
Total Utilization: 17.66%

CPU DESCRIPTION	Processing Rate	Context Switch Rate
CPU	1.00	0

▨ Periodic Utilization
■ Aperiodic Utilization
□ Global Res. Utilization
▨ Unused

PERIODIC TASKS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Resources List
PT1: Network_Mgmt	[illegible]	1	1	125	28	
PT2: CD	[illegible]	2	2	272	19	
PT3: Video	[illegible]	3	3	6000	1175	
PT4: MIDI	[illegible]	4	4	12000	9	
PT5: JPEG1	[illegible]	5	4	27000	1880	
PT6: JPEG2	[illegible]	6	4	33000	1880	

PERIODIC SERVERS	Sched.	Global Priority	Local Priority	Period	Exec. Time	Server Type

Figure 7.3: A Mapping Generated by the Lowest Overlap First Algorithm

and **Delete from Conflict Set**. The introduction of the new options does not affect others. A user can specify which resources under a particular node are conflicting with other resources on the same node. Note that we mean *Incompatability*, as described in Section 5.1.1, when we use the term *Conflicting*.

We have tested the new resource graph, which enables a user to specify incompatability set for methods of an object (modeled are PERTS resources). The resource graph properly:

- **Selects** a PERTS resource.
- **Adds** subsequently selected resources to the conflict set of the resource selected previously and vice versa. It does not permit resources residing on a node other than the selected resource's node, to be a part of the conflict set because that would amount to modeling an *Object* distributed across nodes, which is beyond the scope of our study.

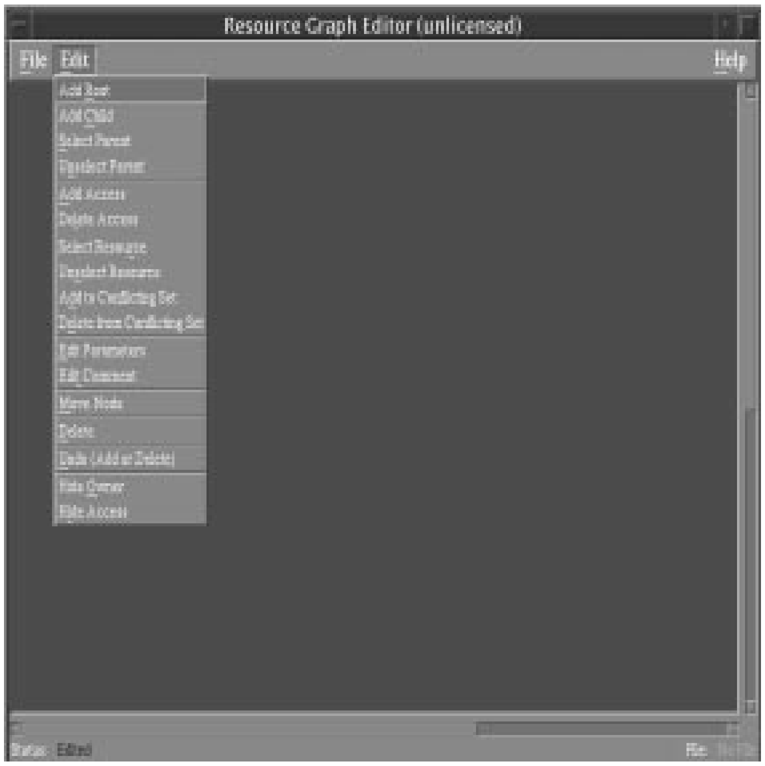


Figure 7.4: Edit Parameter Dialog Box in PERTS 3.2.1

- **Deletes** from the conflict set a resource that is already a part of the set.
- **Unselects** a selected resource.

The **Generate Report** command verifies that the conflict set information is included in the report.

The **Save** command, on a resource graph with the new feature is found to save the conflict set information of all the resources, along with previous parameters. This is verified by opening the text file description of the resource graph in a standard text processing editor.

Opening the saved text file description with the **Open/Reopen** command of the resource graph verifies that the saved conflict set information is correctly read in by the resource graph.

7.2.2 DASPCP: Prioirty Ceiling Calculation

We model the Example 5.1.5, that was presented in Section 5.1.5 in PERTS and verify the correctness of the Priority Ceiling calculations.

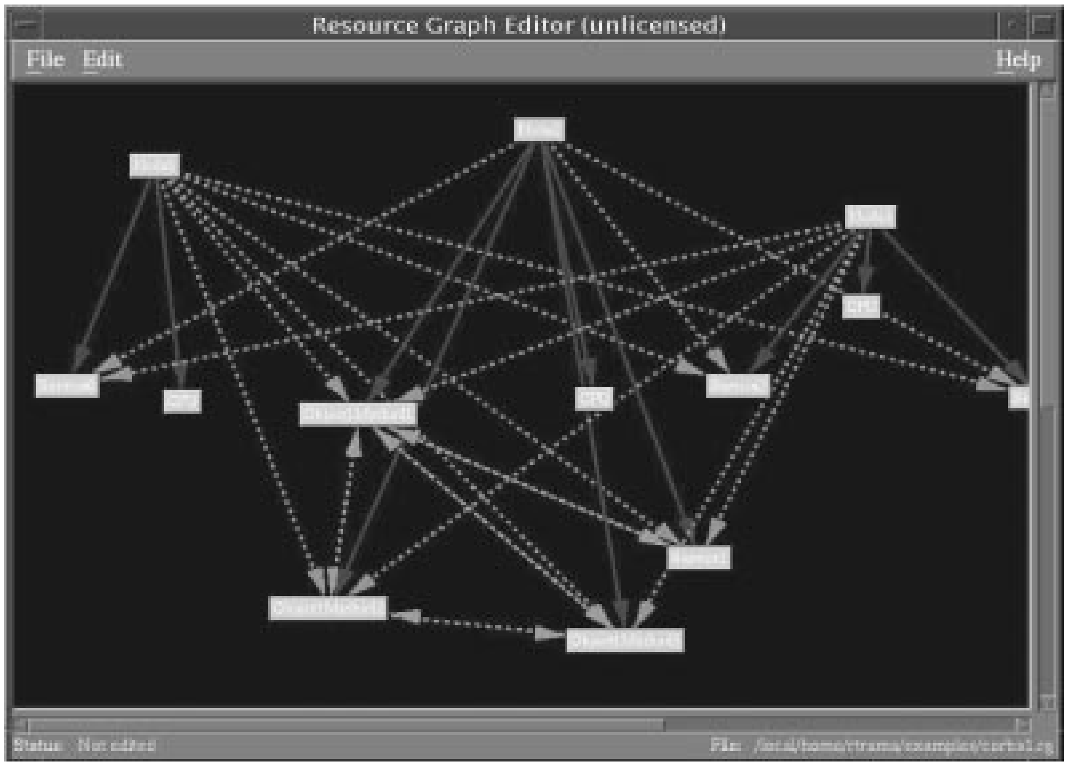


Figure 7.5: Resource Graph Model of the DASPCP Example

Figure 7.5, shows the resource graph description of the system and a complete task graph description is presented in appendix A. An End-to-End analysis of the system was carried out under *DM + DASPCP*, with priority type being set to *User-Defined Priority* and priority mapping set to *Default Priority Mapping.* Figures 7.6 and 7.7 show the output of PERTS analysis. For the benefit of the reader we reproduce the results discussed in Section 5.1.5 here, recalculating the ceilings using the conventions adopted by PERTS. PERTS adopts the following conventions:

- Lower a number, higher the priority of the task associated with it.
- Priorities can take negative values.

- Priority Ceiling of a global resource is calculated as:

  Priority Ceiling = Priority of highest priority task that will ever access this resource - Priority of the lowest priority task in the system.

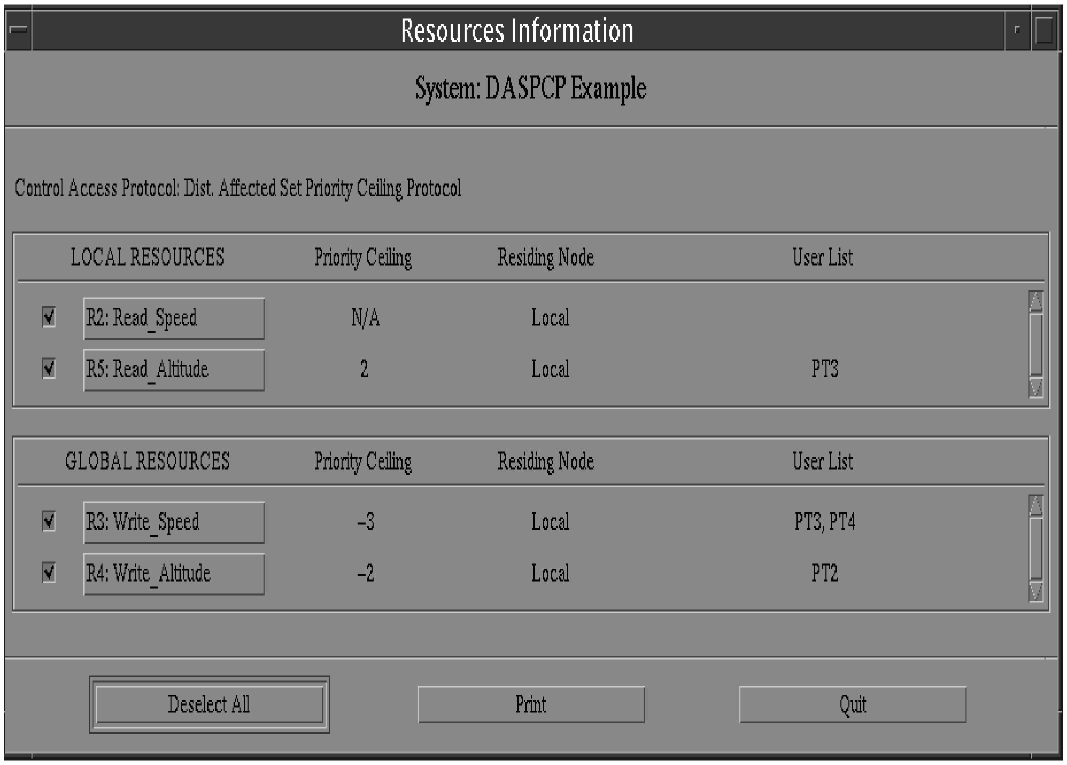
Note: We have retained the condition used earlier that: $P_4 > P_3 > P_2 > P_1$, the priority assignment is altered to reflect the PERTS convention. Hence $P_4 = 1, P_3 = 2, P_2 = 3, P_1 = 4$.

Object O_{track1}				
Method	read_speed	read_altitude	write_speed	write_altitude
Highest Priority Task	N/A	T_3	T_4	T_2
DASPCP Priority Ceiling	N/A	2	$1-4=-3$	$2-4=-2$
DPCP Priority Ceiling	$1-4=-3$			

Object O_{track2}			
Method	read_speed	read_depth	write_speed_depth
Highest Priority Task	T_4	N/A	N/A
DASPCP Priority Ceiling	1	N/A	N/A
DPCP Priority Ceiling	1		

Table 7.2: Priority Ceilings in DASPCP Example

Notice that the priority ceilings of resources calculated by PERTS, shown in Figures, 7.6 and 7.7, are same as those hand calculated in table 7.2. Hence our implementation of DASPCP appears to be correct.


Resources Information

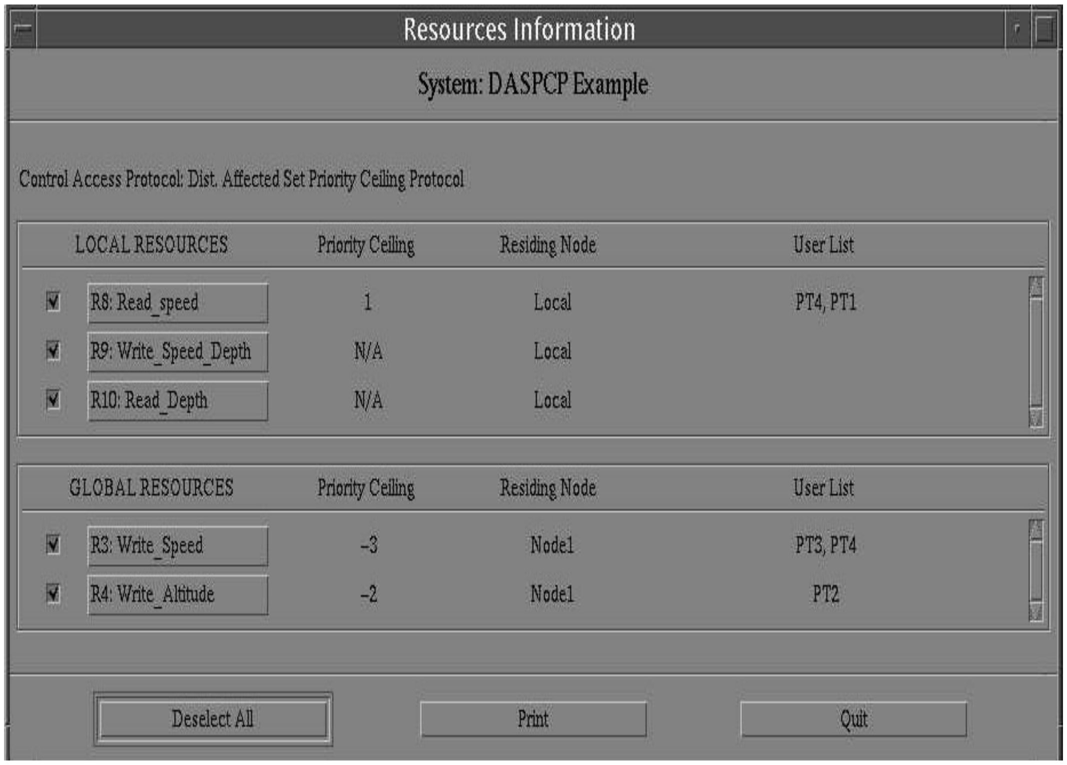
System: DASPCP Example

Control Access Protocol: Dist. Affected Set Priority Ceiling Protocol

LOCAL RESOURCES	Priority Ceiling	Residing Node	User List
✓ R2: Read_Speed	N/A	Local	
✓ R5: Read_Altitude	2	Local	PT3

GLOBAL RESOURCES	Priority Ceiling	Residing Node	User List
✓ R3: Write_Speed	–3	Local	PT3, PT4
✓ R4: Write_Altitude	–2	Local	PT2

Figure 7.6: Output of PERTS Analysis of the DASPCP Example: Node 1



System: DASPCP Example

Control Access Protocol: Dist. Affected Set Priority Ceiling Protocol

LOCAL RESOURCES	Priority Ceiling	Residing Node	User List
✓ R8: Read_speed	1	Local	PT4, PT1
✓ R9: Write_Speed_Depth	N/A	Local	
✓ R10: Read_Depth	N/A	Local	

GLOBAL RESOURCES	Priority Ceiling	Residing Node	User List
✓ R3: Write_Speed	–3	Node1	PT3, PT4
✓ R4: Write_Altitude	–2	Node1	PT2

Figure 7.7: Output of PERTS Analysis of the DASPCP Example: Node 2

Chapter 8

Conclusions and Future Work

In this chapter we summarize the work done so far and discuss potential future work.

8.1 Summary of Work done so far

To summarize, we have achieved the following at the end of this project :

1. Provided a clean interface to input Priority Information associated with each entity.
2. Enhanced PERTS to support mapping of global task priorities to their local priorities by implementing the Lowest Overlap First priority mapping algorithm.
3. Enabled a user to specify incompatabilities among an *Object's* methods graphically.
4. Augmented PERTS Engine to incorporate DASPCP as an alternative resource control protocol.
5. Designed and implemented an interface between PERTS and the RT CORBA Scheduling Service.

8.2 Future Work

There can be no end to advancement of human knowledge and every effort should be made to do a job better if there is a way to do it. With this view in mind we outline possible future work with regards to Priority Mapping, RT CORBA and PERTS.

8.2.1 Priority Mapping

Recall from the description of the Lowest Overlap First algorithm, Chapter 4 that the algorithm requires a test of schedulability to be done every time a overlap of local priority is made. This test can be very time consuming and will add a considerable overhead to the overall computation time if the system has a large number of tasks running on a node while there are relatively fewer number of priorities available on that node. We could do better if there is an efficient *Heuristic Algorithm* that can do the mapping faster with a high degree of succes ratio. By success, we mean that, the system of tasks is schedulable with the mapping produced by the heuristic algorthm. In this direction, it is worth studying the work done by Lehoczky and Sha [20]. They suggest a *Grid* approach to solve the mapping problem. It is a heuristic approach but nevertheless could be of great value to a system designer who wants the analysis done faster. The algorithm's feasibility vis a vis its implementation in PERTS has to be studied. PERTS can then offer a variety of Priority Mapping solutions which is very important to the ORB vendors developing ORBs that conform to RT CORBA 1.0 standard.

8.2.2 Network Delay in PERTS

PERTS allows a user to input a *worst case* estimate of the overhead due to communication across a network suffered by applications. The worst case estimate might actually affect the schedulability result of the system. There is a need to model network traffic better. A variety of solutions are possible. PERTS can provide the user with a host of probability distributions of the network traffic and the user can select one that most appropriately models his/her environment. This will most definitely alter the schedulability equation 2.2 and has to be studied further.

8.2.3 PERTS and RT CORBA

Recall from the discussion of RT CORBA in Chapter 3, that the RT CORBA 1.0 Scheduling Service Interface, has a function *create_POA*, which guarantees the application that uses this call, that all the real-time policies listed in the draft standard will set transparently and the application does not have to worry about it. It is easier said than done. A number of issues are involved in this, mainly:

- We have to *identify* what the real-time policies are, that the RT CORBA 1.0 draft requires an ORB to support.
- We need to study if any of these policies affect the schedulability of the system, if so, it has to be appropriately taken into the schedulability analysis.
- The PERTS-RT_CORBA configuration file presently is limited to Priority information of the system. It has to be augmented to include all the POA features, so that, the scheduling service transparently sets these parameters in the ORB on a call to *create_POA*.

We hope that the reader has been benefited by this report and conclude with a famous quote:

> We dance around the circle and suppose, the answer sits there and knows.

References

[1] Tri-Pacific Software, at *http://www.tripac.com*

[2] Object Management Group, at *http://www.omg.org*

[3] Jane W.S. Liu et. al. *PERTS : A Prototyping Environment for Real-Time Systems.* Technical Report UIUCDCS-R-93-1082, The University of Illinois, Urbana, May 1993.

[4] PERTS On-line Documents at *http://pertsserver.cs.uiuc.edu/perts/*

[5] C.L.Liu and J.W.Layland, *Scheduling Algorithms for Multiprogramming in Hard Real-Time Systems* Journal of the Association of the Computing Machinery, Vol.20, No.1, pp. 46-61, January 1973.

[6] J. Leung and J.Whitehead, *On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks*, Performance Evaluation 2, pp. 237-250, 1982.

[7] Liu Sha, R. Rajkumar and J.P. Lehoczky *Priority Inheritance Protocols : An Approach to Real-Time Synchronization.* IEEE Transactions on Computers, Vol. 39, No 9, September 1990.

[8] T.P.Baker *Stack-Based Allocation Policy for Real-Time Processes*, Proceedings of 11th Real-Time Systems Symposium, pp 191-200, December 1990.

[9] J.P. Lehoczky, L.Sha and Y.Ding, *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*, Proceedings of 10th Real-Time Systems Symposium, December 1989.

[10] OMG. Realtime CORBA. Electronic document at *http://www.omg.org/docs/orbos/98-10-05.pdf*

[11] Daniel I. Katcher, Shirish S. Sathaye and Jay K. Strosnider. *Fixed Priority Scheduling with Limited Priority Levels.* IEEE Transactions on Computers. Vol. 44, No 9, pp. 1140-1144, Sept 1995.

[12] S.Cheng, J.A.Stankovic and K.Ramamritham, *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey*, IEEE Real-Time Systems Symposium 1988.

[13] Jane Liu, *Real-Time Systems* Rough Draft, October 1997.

[14] Levon Esibov. *Support for Automated Schedulability Analysis for Distributed Real-Time Middleware.* Masters Thesis. University of Rhode Island, 1998.

[15] V.F.Wolfe, L.C.DiPippo et.al, *Real-Time CORBA* In Proceedings of the Third IEEE Real-Time Technology and Applications Symposium, June 1997.

[16] L.C.DiPippo and V.F.Wolfe *Object-based Semantic Real-Time Concurrency Control with Bounded Imprecision*, IEEE Transactions on Knowledge and Data Engineering, vol. 9 No.1, pp. 135-147, Jan-Feb 1997.

[17] B.Badrinath and K.Ramamritham, *Synchronizing Transactions on Objects*, IEEE Transactions on Computers, vol.37, No.5, pp. 541-547, May 1988.

[18] Lisa DiPippo, Vic Wolfe, et. al. *Scheduling and Priority Mapping for Static Real-Time Middleware.* To be published in Real-Time Systems Journal, special issue on real-time middleware.

[19] Micheal Squadrito, Levon Esibov et. al. *Concurrency Control in Real-Time Object-Oriented Systems: The Affected Set Priority Ceiling Protocols.* University of Rhode Island Technical Report. 1997.

[20] J.P.Lehoczky and L.Sha, *Performance of Real-Time Bus Scheduling Algorithms*, ACM Performance Evaluation Review, Special Issue, vol.14, May 1986.

Appendix A

Test Cases

We present details of task and resource graph descriptions of the test cases discussed in Chapter 7.

Multimedia Task System

```
GRAPH Multimedia_Task_Set:
IS
  TASK Netwrk_Mgmt:
                              X_COORD              91;
                              Y_COORD              88;
                              READYTIME             0;
                              DEADLINE            125;
                              PERIOD              125;
                              MIN_PERIOD          125;
                              MAX_PERIOD          125;
                              PHASE                 0;
                              WEIGHT                1;
                              INSTANCE              0;
                              LAXITY_TYPE           1;
                              PROCESSOR     Node1.CPU;
                              LENGTH               28;
                              MIN_WORK             28;
                              MAX_WORK             28;
                              NETWORK_DELAY         0;
                              U_PRIORITY            1;
  END
  TASK CD:
                              X_COORD              57;
                              Y_COORD             170;
                              READYTIME             0;
                              DEADLINE            272;
                              PERIOD              272;
                              MIN_PERIOD          272;
                              MAX_PERIOD          272;
                              PHASE                 0;
                              WEIGHT                1;
                              INSTANCE              0;
                              LAXITY_TYPE           1;
                              PROCESSOR     Node1.CPU;
                              LENGTH               19;
                              MIN_WORK             19;
                              MAX_WORK             19;
                              NETWORK_DELAY         0;
                              U_PRIORITY            2;
  END
```

```
TASK Voice:
                 X_COORD                    62;
                 Y_COORD                   238;
                 READYTIME                   0;
                 DEADLINE                 6000;
                 PERIOD                   6000;
                 MIN_PERIOD               6000;
                 MAX_PERIOD               6000;
                 PERIOD_DISTR                0;
                 PHASE                       0;
                 WEIGHT                      1;
                 INSTANCE                    0;
                 LAXITY_TYPE                 1;
                 PROCESSOR           Node1.CPU;
                 LENGTH                   1175;
                 MIN_WORK                 1175;
                 MAX_WORK                 1175;
                 NETWORK_DELAY               0;
                 U_PRIORITY                  3;
END
TASK MIDI:
                 X_COORD                    65;
                 Y_COORD                   314;
                 READYTIME                   0;
                 DEADLINE                12000;
                 PERIOD                  12000;
                 MIN_PERIOD              12000;
                 MAX_PERIOD              12000;
                 PERIOD_DISTR                0;
                 PHASE                       0;
                 WEIGHT                      1;
                 INSTANCE                    0;
                 LAXITY_TYPE                 1;
                 PROCESSOR           Node1.CPU;
                 LENGTH                      9;
                 MIN_WORK                    9;
                 MAX_WORK                    9;
                 NETWORK_DELAY               0;
                 U_PRIORITY                  4;
END
```

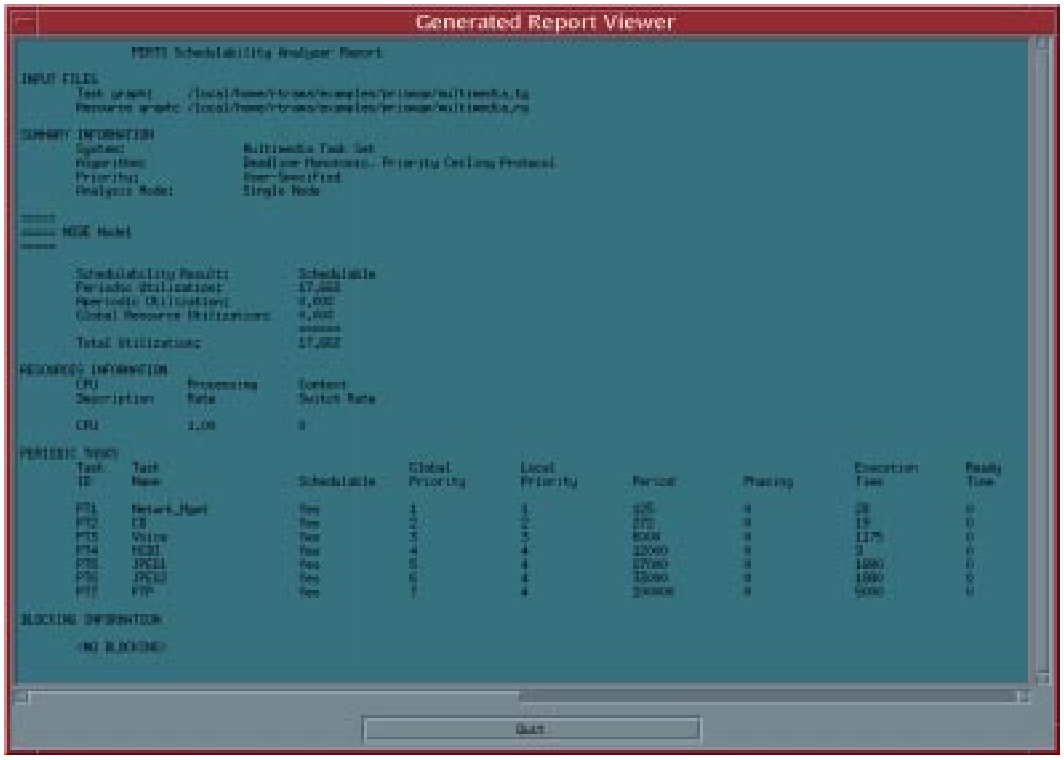
```
TASK JPEG1:
                  X_COORD                  67;
                  Y_COORD                 377;
                  READYTIME                 0;
                  DEADLINE            270004;
                  PERIOD                27000;
                  MIN_PERIOD            27000;
                  MAX_PERIOD            27000;
                  PERIOD_DISTR              0;
                  PHASE                     0;
                  WEIGHT                    1;
                  INSTANCE                  0;
                  LAXITY_TYPE               1;
                  PROCESSOR         Node1.CPU;
                  LENGTH                 1880;
                  MIN_WORK               1880;
                  MAX_WORK               1880;
                  NETWORK_DELAY             0;
                  U_PRIORITY                5;
END
TASK JPEG2:
                  X_COORD                  64;
                  Y_COORD                 453;
                  READYTIME                 0;
                  DEADLINE              33000;
                  PERIOD                33000;
                  MIN_PERIOD            33000;
                  MAX_PERIOD            33000;
                  PERIOD_DISTR              0;
                  PHASE                     0;
                  WEIGHT                    1;
                  INSTANCE                  0;
                  LAXITY_TYPE               1;
                  PROCESSOR         Node1.CPU;
                  LENGTH                 1880;
                  MIN_WORK               1880;
                  MAX_WORK               1880;
                  NETWORK_DELAY             0;
                  U_PRIORITY                6;
END
```

```
  TASK FTP:
                 X_COORD                       54;
                 Y_COORD                      506;
                 READYTIME                      0;
                 DEADLINE                   10000;
                 PERIOD                    100000;
                 MIN_PERIOD                100000;
                 MAX_PERIOD                100000;
                 PERIOD_DISTR                   0;
                 PHASE                          0;
                 WEIGHT                         1;
                 INSTANCE                       0;
                 LAXITY_TYPE                    1;
                 PROCESSOR              Node1.CPU;
                 LENGTH                      5000;
                 MIN_WORK                    5000;
                 MAX_WORK                    5000;
                 NETWORK_DELAY                  0;
                 U_PRIORITY                     7;
  END
END
```

Resource Graph Description

```
RESOURCE GRAPH:
IS
  RESOURCE Node1:
                              X_COORD                   349;
                              Y_COORD                   118;
                              NUMBER_OF_UNITS             0;
                              ACQUISITION_TIME            0;
                              DEACQUISITION_TIME          0;
                              CONTEXT_SWITCH_TIME         0;
                              PROCESSING_RATE             0;
                              CONSUMABILITY               0;
                              PREEMPTABILITY              1;
                              CATEGORY                    0;
                              MAPPING_DIR                 0;
    RESOURCE CPU:
                                X_COORD                 342;
                                Y_COORD                 193;
                                NUMBER_OF_UNITS           1;
                                ACQUISITION_TIME          0;
                                DEACQUISITION_TIME        0;
                                CONTEXT_SWITCH_TIME       0;
                                PROCESSING_RATE           1;
                                CONSUMABILITY             0;
                                PREEMPTABILITY            1;
                                CATEGORY                  0;
                                MAPPING_DIR               1;
                                PRIO_RANGE           [1, 4];
    END
  END
END
```

The output of the priority mapping is illustrated in Figure A.1.



```
                PERTS Schedulability Analyzer Report

INPUT FILES
        Task graph:     /local/home/[illegible]/examples/[illegible]/multimedia.tg
        Resource graph: /local/home/[illegible]/examples/[illegible]/multimedia.rg

SUMMARY INFORMATION
        System:                 Multimedia Task Set
        Algorithm:              Deadline Monotonic, Priority Ceiling Protocol
        Priority:               User-Specified
        Analysis Mode:          Single Node

=====
===== NODE Model
=====

        Schedulability Result:          Schedulable
        Periodic Utilization:           17.88%
        Aperiodic Utilization:          0.00%
        Global Resource Utilization:    0.00%
                                        ======
        Total Utilization:              17.88%

RESOURCES INFORMATION
        CPU             Processing      Context
        Description     Rate            Switch Rate

        CPU             1.00            0

PERIODIC TASKS
        Task    Task                                    Global          Local                                                   Execution       Ready
        ID      Name                    Schedulable     Priority        Priority        Period          Phasing         Time            Time

        PT1     Network_Mgmt            Yes             1               1               125             0               28              0
        PT2     C8                      Yes             2               2               272             0               19              0
        PT3     Voice                   Yes             3               3               8000            0               1175            0
        PT4     MIDI                    Yes             4               4               32000           0               3               0
        PT5     JPEG1                   Yes             5               4               27000           0               1000            0
        PT6     JPEG2                   Yes             6               4               33000           0               1000            0
        PT7     FTP                     Yes             7               4               100000          0               5000            0

BLOCKING INFORMATION

        <NO BLOCKING>
```

Figure A.1: PERTS Single Node Analysis Report

Task Graph Description of DASPCP Example

```
GRAPH DASPCP Example:
IS
  TASK Task1:
                                               X_COORD                          138;
                                               Y_COORD                          156;
                                               READYTIME                          0;
                                               DEADLINE                          30;
                                               PERIOD                            30;
                                               MIN_PERIOD                        30;
                                               MAX_PERIOD                        30;
                                               PHASE                              0;
                                               WEIGHT                             1;
                                               INSTANCE                           0;
                                               LAXITY_TYPE                        1;
                                               PROCESSOR                  Node2.CPU;
                                               LENGTH                             5;
                                               MIN_WORK                           5;
                                               MAX_WORK                           5;
                                               U_PRIORITY                         4;
                                               RESOURCE    Read_speed, 1, 0, 0, [3, 6];
  END
  TASK Task2:
                                               X_COORD                          318;
                                               Y_COORD                          154;
                                               READYTIME                          2;
                                               DEADLINE                          28;
                                               PERIOD                            30;
                                               MIN_PERIOD                        30;
                                               MAX_PERIOD                        30;
                                               PHASE                              0;
                                               WEIGHT                             1;
                                               INSTANCE                           0;
                                               LAXITY_TYPE                        1;
                                               PROCESSOR                  Node2.CPU;
                                               LENGTH                             5;
                                               MIN_WORK                           5;
                                               MAX_WORK                           5;
                                               U_PRIORITY                         3;
                                               RESOURCE Write_Altitude, 1, 0, 0, [2, 4];
  END
  TASK Task3:
                X_COORD                                            472;
                Y_COORD                                            149;
                READYTIME                                            0;
                DEADLINE                                            30;
                PERIOD                                              30;
                MIN_PERIOD                                          30;
                MAX_PERIOD                                          30;
                PHASE                                                0;
                WEIGHT                                               1;
                INSTANCE                                             0;
                LAXITY_TYPE                                          1;
                PROCESSOR Node1.CPU;
                LENGTH                                               5;
                MIN_WORK                                             5;
                MAX_WORK                                             5;
                U_PRIORITY                                           2;
                RESOURCE                   Write_Speed, 1, 0, 0, [2, 4];
                RESOURCE                 Read_Altitude, 1, 0, 0, [1, 2];
  END
  TASK Task4:
                X_COORD                                            624;
                Y_COORD                                            142;
                READYTIME                                            2;
                DEADLINE                                            28;
                PERIOD                                              30;
                MIN_PERIOD                                          30;
                MAX_PERIOD                                          30;
                PHASE                                                0;
                WEIGHT                                               1;
                LAXITY_TYPE                                          1;
                PROCESSOR Node2.CPU;
                LENGTH                                               5;
                MIN_WORK                                             5;
                MAX_WORK                                             5;
                U_PRIORITY                                           1;
                RESOURCE                   Write_Speed, 1, 0, 0, [3, 4];
                RESOURCE                    Read_speed, 1, 0, 0, [4, 7];
  END
END
```

Resource Graph Description of DASPCP Example

```
RESOURCE_GRAPH:
IS
  RESOURCE Node1:
                                        X_COORD                    258;
                                        Y_COORD                    128;
                                        NUMBER_OF_UNITS              0;
                                        ACQUISITION_TIME             0;
                                        DEACQUISITION_TIME           0;
                                        CONTEXT_SWITCH_TIME          0;
                                        PROCESSING_RATE              0;
                                        CONSUMABILITY                0;
                                        PREEMPTABILITY               1;
                                        CATEGORY                     0;
                                        MAPPING_DIR                  0;
    RESOURCE CPU:
                                          X_COORD                  251;
                                          Y_COORD                  203;
                                          NUMBER_OF_UNITS            1;
                                          ACQUISITION_TIME           0;
                                          DEACQUISITION_TIME         0;
                                          CONTEXT_SWITCH_TIME        0;
                                          PROCESSING_RATE            1;
                                          CONSUMABILITY              0;
                                          PREEMPTABILITY             1;
                                          CATEGORY                   0;
                                          MAPPING_DIR                0;
                                          PRIO_RANGE           [0, 59];
    END
```

```
RESOURCE Read_Speed:
                        X_COORD                          137;
                        Y_COORD                          314;
                        NUMBER_OF_UNITS                    1;
                        ACQUISITION_TIME                   0;
                        DEACQUISITION_TIME                 0;
                        CONTEXT_SWITCH_TIME                0;
                        PROCESSING_RATE                    0;
                        CONSUMABILITY                      0;
                        PREEMPTABILITY                     1;
                        CATEGORY                           0;
                        MAPPING_DIR                        0;
                        CONFLICT_RESRC_SET      Write_Speed;
END
RESOURCE Write_Speed:
                        X_COORD                          211;
                        Y_COORD                          406;
                        NUMBER_OF_UNITS                    1;
                        ACQUISITION_TIME                   0;
                        DEACQUISITION_TIME                 0;
                        CONTEXT_SWITCH_TIME                0;
                        PROCESSING_RATE                    0;
                        CONSUMABILITY                      0;
                        PREEMPTABILITY                     1;
                        CATEGORY                           0;
                        MAPPING_DIR                        0;
                        CONFLICT_RESRC_SET       Read_Speed;
                        ACCESS_BY_LIST                Node2;
END
```

```
  RESOURCE Write_Altitude:
                              X_COORD                          380;
                              Y_COORD                          406;
                              NUMBER_OF_UNITS                    1;
                              ACQUISITION_TIME                   0;
                              DEACQUISITION_TIME                 0;
                              CONTEXT_SWITCH_TIME                0;
                              PROCESSING_RATE                    0;
                              CONSUMABILITY                      0;
                              PREEMPTABILITY                     1;
                              CATEGORY                           0;
                              MAPPING_DIR                        0;
                              CONFLICT_RESRC_SET     Read_Altitude;
                              ACCESS_BY_LIST                Node2;
  END
  RESOURCE Read_Altitude:
                              X_COORD                          430;
                              Y_COORD                          283;
                              NUMBER_OF_UNITS                    1;
                              ACQUISITION_TIME                   0;
                              DEACQUISITION_TIME                 0;
                              CONTEXT_SWITCH_TIME                0;
                              PROCESSING_RATE 0;
                              CONSUMABILITY                      0;
                              PREEMPTABILITY                     1;
                              CATEGORY                           0;
                              MAPPING_DIR                        0;
                              CONFLICT_RESRC_SET    Write_Altitude;
  END
END
```

```
RESOURCE Node2:
                          X_COORD                                578;
                          Y_COORD                                116;
                          NUMBER_OF_UNITS                          0;
                          ACQUISITION_TIME                         0;
                          DEACQUISITION_TIME                       0;
                          CONTEXT_SWITCH_TIME                      0;
                          PROCESSING_RATE                          0;
                          CONSUMABILITY                            0;
                          PREEMPTABILITY                           1;
                          CATEGORY                                 0;
                          MAPPING_DIR                              0;
  RESOURCE CPU:
                            X_COORD                              516;
                            Y_COORD                              184;
NUMBER_OF_UNITS           1;
                            ACQUISITION_TIME                       0;
                            DEACQUISITION_TIME                     0;
                            CONTEXT_SWITCH_TIME                    0;
                            PROCESSING_RATE                        1;
                            CONSUMABILITY                          0;
                            PREEMPTABILITY                         1;
                            CATEGORY                               0;
                            MAPPING_DIR                            0;
                            PRIO_RANGE                       [0, 59];
  END
  RESOURCE Read_speed:
                            X_COORD                              573;
                            Y_COORD                              283;
                            NUMBER_OF_UNITS                        1;
                            ACQUISITION_TIME                       0;
                            DEACQUISITION_TIME                     0;
                            CONTEXT_SWITCH_TIME                    0;
                            PROCESSING_RATE                        0;
                            CONSUMABILITY                          0;
                            PREEMPTABILITY                         1;
                            CATEGORY                               0;
                            MAPPING_DIR                            0;
                            CONFLICT_RESRC_SET    Write_Speed_Depth;
  END
```

```
    RESOURCE Write_Speed_Depth:
                                      X_COORD                                    654;
                                      Y_COORD                                    392;
                                      NUMBER_OF_UNITS                              1;
                                      ACQUISITION_TIME                             0;
                                      DEACQUISITION_TIME                           0;
                                      CONTEXT_SWITCH_TIME                          0;
                                      PROCESSING_RATE                              0;
                                      CONSUMABILITY                                0;
                                      PREEMPTABILITY                               1;
                                      CATEGORY                                     0;
                                      MAPPING_DIR                                  0;
                                      CONFLICT_RESRC_SET    Read_Depth, Read_speed;
    END
    RESOURCE Read_Depth:
                                      X_COORD                                    694;
                                      Y_COORD                                    281;
                                      NUMBER_OF_UNITS                              1;
                                      ACQUISITION_TIME                             0;
                                      DEACQUISITION_TIME                           0;
                                      CONTEXT_SWITCH_TIME                          0;
                                      PROCESSING_RATE                              0;
                                      CONSUMABILITY                                0;
                                      PREEMPTABILITY                               1;
                                      CATEGORY                                     0;
                                      MAPPING_DIR                                  0;
                                      CONFLICT_RESRC_SET         Write_Speed_Depth;
    END
  END
END
```

Bibliography

Baker, T.P. *Stack-Based Allocation Policy for Real-Time Processes*, Proceedings of 11th Real-Time Systems Symposium, pp 191-200, December 1990.

Esibov, Levon. *Support for Automated Schedulability Analysis for Distributed Real-Time Middleware.* Masters Thesis. University of Rhode Island, 1998.

Katcher, Daniel I., Sathaye, Shirish S. and Strosnider, Jay K. *Fixed Priority Scheduling with Limited Priority Levels.* IEEE Transactions on Computers. Vol. 44, No 9, pp. 1140-1144, Sept 1995.

Lehoczky, J.P., Sha, L. and Ding Y. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*, Proceedings of 10th Real-Time Systems Symposium, December 1989.

Liu, C.L. and Layland, J.W. *Scheduling Algorithms for Multiprogramming in Hard Real-Time Systems* Journal of the Association of the Computing Machinery, Vol.20, No.1, pp. 46-61, January 1973.

Liu, Jane W.S. et. al. *PERTS : A Prototyping Environment for Real-Time Systems.* Technical Report UIUCDCS-R-93-1082, The University of Illinois, Urbana, May 1993.

Liu, Jane *Real-Time Systems* Rough Draft, October 1997.

Rajkumar R., *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

Sha, Liu, Rajkumar R. and Lehoczky, J.P. *Priority Inheritance Protocols : An Approach to Real-Time Synchronization.* IEEE Transactions on Computers, Vol. 39, No 9, September 1990.

Squadrito, Micheal A., *Extending the Priority Ceiling Protocol Using Read/Write Affected Sets*, M.S.Thesis, URI, 1996.