12-1-1996

On Scheduling Real-Time Periodic Tasks in a Multiprocessor Environment.

Stephen J. Bernard

On Scheduling Real-Time Periodic Tasks in a Multiprocessor Environment

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

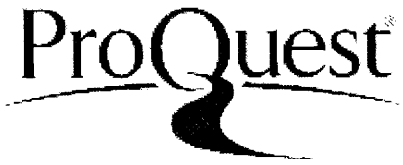
Master of Science

University of Nebraska at Omaha

by

Stephen J. Bernard

December 1996



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree Master of Science, University of Nebraska at Omaha.

Committee

Name	Department/School
[illegible]	Computer Science
Pamela Spurlock	Management
Hesham H. Ali (HE)	Computer Science

Chairperson H. El-Rewini

Date Dec 5, 1996

Abstract

Real-Time periodic tasks are at the heart of many critical processing computer systems. Nuclear power plants, military command and control systems, aircraft automatic flight control systems, hospital life-support equipment all require precise processing performed within very strict timelines. Missing deadlines can have catastrophic consequences. The rate-monotonic priority assignment policy for scheduling hard-deadline periodic tasks was developed to guarantee those deadlines.

In this thesis, we study the problem of scheduling hard-deadline periodic tasks. We begin by surveying the current state of multiprocessor rate-monotonic scheduling and reviewing earlier work. We present the results of a number of experiments we conducted to evaluate the performance of several scheduling heuristics. These heuristics assumed a homogeneous multiprocessing environment. We relax that restriction and introduce three allocation heuristics for scheduling tasks on heterogeneous multiprocessors. Furthermore, we analyze the performance of the proposed algorithms. We compare the quality of the solutions produced by these algorithms and measure them against the optimal solution. Lacking in the current set of real-time multiprocessor heuristics is the awareness of communication between tasks. We add communication into the scheduling model and provide an algorithm to minimize the amount of data transfer between tasks. Furthermore, we examine the performance of this heuristic and compare the schedules it produces with optimal solutions. Lastly, we introduce a scheduling and analysis tool that

incorporates several scheduling heuristics. New heuristics are easily added to the tool. The goal of the tool is to help system designers/developers study the performance of different heuristics in scheduling real-time periodic tasks. The tool helps answer "what if" questions, which may also help designers tune their systems to achieve better performance while meeting deadlines.

Acknowledgment

I would like to take this opportunity to express my genuine gratitude to Hesham El-Rewini. His patience, tutelage, and never ending optimism throughout our research, class work, and especially this thesis, has made my experience at UNO enjoyable and informative.

Though my ever loving wife may not read this work, I must say that without her, this thesis would not be possible. Thank you!

Table of Contents

Figures

Tables

1 Introduction

Real-time computers control some of the most critical processing systems in use today including aircraft autopilot and navigation systems, nuclear power plants, military command and control systems, medical equipment, and air traffic control systems. What distinguishes real-time systems from other types of computer systems is the nature of its processing. Not only must real-time systems produce precise and correct output, they must also produce results within strict time lines. The consequences of not meeting a time requirement can be catastrophic.

1.1 Real-Time Systems

We can clarify what a real-time system is by considering the aspects of a real-time system. Most real-time systems share the characteristics of task deadlines, strict control over allocation of resources, and system predictability. In addition, a special class of real-time systems contain the added requirement of periodic tasks. An aircraft autopilot system is a good example of a real-time system with these characteristics. Suppose that there is a requirement for the autopilot to update the aircraft heading every second. The process of updating the heading can be composed if three independent operations: 1) read a sensor, 2) determine if and how much of a heading change is required, and 3) format the result and actually make the change. Of course all three operations could be combined in one task and with an assigned deadline of one second. But if the system timing cycle

requires the sensor reading operation to input data every 25 milliseconds and average the samples before sending them to the processing operation, then the processor would be constantly busy with this one task and will not be able to service the other autopilot operations.

A better approach is to give each operation its own task with its own period and deadline. The period of the sensor task will be smaller than the processing task, the processing task must end before the control task starts, and the control task must complete processing before the one second deadline is up. Under this scheme, the processor can service other sensor, processing, and control tasks in between requests for the set of tasks used for updating the heading.

Tasks in a real-time system may need to share some physical resources and global data areas. Since each task has a deadline, we must minimize the time high priority tasks wait on lower priority tasks to use and free up the needed resource. Scheduling is one area of real-time system design and maintenance affecting the ability of tasks to meet their deadlines while ensuring effective use of the hardware and software resources. Encompassing the real-time system characteristics of deadlines and scheduling of resources is predictability. System predictability ensures we know in advance and at all times that all tasks are guaranteed to meet their deadlines without unnecessarily waiting for resource availability. By using well developed and tested algorithms for assigning

priorities to tasks and sharing resources between tasks, we can have a safe and predictable real-time system.

1.2 The Scheduling Problem

Given a set of tasks and a set of processors, the scheduling problem is to determine an assignment of tasks to processors such that particular performances goals are met. This problem is known to be NP-Complete in its general form as well as in many restricted cases [29]. In an attempt to solve this problem fast heuristics are introduced. These heuristics do not guarantee optimal solutions, but try to find near optimal solutions most of the time.

Due to the many different flavors of task scheduling, there is an enormous number of possible combinations of characteristics to work within. We shall group the most important characteristics into four categories: Priority Assignment, Task Dependence, Task Timing, Task Deadlines. These four basic categories will supply enough distinction to provide a good definition of our scheduling model.

Priority Assignment

Operating systems use priorities when determining which task to execute. Tasks with higher priorities run before those with lower priorities. A priority assignment scheme determines how and when priorities are assigned to tasks. Priorities are either assigned

statically or dynamically. If priorities are set once and never modified by the run-time system, they are considered static. Conversely, when a task's priority is modified during system execution, the assignment scheme is considered dynamic.

Similar to, but distinctly different than the static/dynamic nature of priority setting, is the on-line/off-line process of assigning priorities. Off-line priority assignment policies set the priority of each task in the application before the application has started executing on the processor. When priorities are assigned to tasks during application initialization or while the application is running, the assignment policy is considered on-line.

Priority assignment policies can be any combination of the above mentioned schemes. On-line definition of priorities does not necessarily imply dynamic determination of the priority of each task. Hence, if the priority is set at application initialization time but never modified during execution time, the assignment scheme is considered on-line and static. For this thesis, we only consider static off-line task assignment algorithms.

Task Dependence

Different tasks depend on each other in three distinct ways: precedence, synchronization, and communication. A precedence relation refers to the execution order of tasks within the task set. If task τ_i must complete execution before τ_j can start processing, then τ_i has a precedence relation with task τ_j.

Synchronization is another type of dependence between tasks. Two tasks use synchronization to maintain a constant relative timing between them. Furthermore, synchronization also provides exclusive access to logical and physical devices. Logical devices refer to global data areas where only one task may modify the value at a time and physical devices are hardware that can only be accessed by one task at a time.

The final type of task dependence we shall consider is communication. Communication between two tasks occurs when one task requires data from another task. This data sharing can be accomplished by making use of shared memory areas or message passing. In addition, communication can occur with or without synchronization.

Task Timing

One of the fundamental distinctions of real-time systems is the timing nature of their task sets. Real-time tasks are considered periodic or aperiodic. Periodic tasks are ready to execute at regular, well defined intervals (monotonic) while aperiodic tasks occur whenever some nonrecurring event requires attention. Consider a task τ_i with a period of $T_i = 13$. Task τ_i will then be ready for execution at times 0, 13, 26, 39, ..., $j * T_i$; where j is the j^{th} instance of τ_i.

Task Deadlines

The last aspect of task scheduling we shall consider is task deadlines. A deadline for a task designates the time at which all computations must be complete. We can further describe the task set by how strict the deadlines are. Tasks with hard deadlines must complete processing by the predetermined deadline or catastrophic consequences can occur. These are often called hard tasks or the system is considered a "hard real-time" system. In this sense, late tasks have no value. Conversely, when tasks have soft deadlines, processing must complete as soon as possible but not to the detriment of causing a hard task to miss its deadline. In a system with soft real-time tasks, results from late tasks still have value even though deadlines were missed. Understandably, systems can have strictly hard tasks, soft tasks, or a combination of both types of deadlines.

With the four scheduling characteristics of task priority assignment, task dependence, task timing, and task deadlines we can now formulate a problem model to lay the foundation for scheduling periodic tasks in a real-time system.

1.3 Scheduling Model

We describe the scheduling model as a 3-tuple ($\mathfrak{T}$, R, M) containing the set of real-time tasks, $\mathfrak{T}$, a set of resources, R, and a target machine, M.

Tasks

The task set $\mathfrak{T} = \{\tau_i = (T_i, C_i, D_i) \mid i = 1, ..., n\}$ has n periodic tasks with each task τ_i containing a period, T_i, the amount of computation time required for execution, C_i, and a deadline of D_i. In our scheduling model, $T_i \geq 1$ for periodic tasks and $T_i = 0$ for aperiodic tasks. Furthermore, the deadline for task τ_i must be equal to its period; $T_i = D_i$. This implies that all computation requirements must be completed before the next instance of the task is ready to execute.

Target Machine

Tasks make use of resources to complete their required processing. Resources are either physical (tape drives, communication ports, ...) or logical (critical sections of code, shared data, ...) and since only one task may access each resource at a time, tasks must synchronize in order to share resources. We declare the set of resources as: $R = \{r_k \mid k = 0, ..., o\}$ of o resources. Our target machine is defined as: $M = \{\rho_j \mid j = 1, ..., m\}$ of m identical processors. Furthermore, we define the load task τ_i places on processor ρ_j as $u_i = C_i/T_i$. The total load placed on processor ρ_j for the entire task set is called the utilization, U, of the task set and is defined by the sum of the individual load factors; $U = \sum_{i=1}^{n} C_i/T_i$.

Performance Measures

Now that we have defined the scheduling model, we can enumerate a set of performance measures to work within. For periodic tasks the most important goal is to meet all required deadlines. After that, we seek to maximize the CPU utilization task sets can have on the processor.

When scheduling tasks on multiprocessor machines, a standard goal is to minimize the number of processors required to execute the task set and still guarantee task deadlines. Additionally, if two algorithms use the same number of processors we can look at the average utilization per processor. An algorithm producing a higher average load is desirable.

If a task completes its execution before the deadline, the task is considered "early" and we define the earliness as the difference between the completion time and the deadline; D_i - Completion Time. The goal is to seek maximization of the average earliness of the task set. When an algorithm fails to guarantee that all tasks will meet their deadlines we might want to minimize the amount of tardiness in the task set [11]. Tardiness is defined as the difference between a tasks deadline and its completion time; Completion Time - D_i.

When considering aperiodic tasks the optimization goal will be to minimize the average response time for all aperiodic tasks. Response time is defined as the difference between

when a task is ready to execute and when it has completed execution. Thus, the response time for an aperiodic task is Completion Time - Ready Time.

The last performance measure deals with task synchronization. During synchronization only one task can access a resource at a time. Therefore, if two tasks require access to the resource, only one should gain control and the other will become blocked (placed in a non-running state) until the resource is free. The standard goal is to minimize the amount of time tasks are blocked. Table I summarizes real-time scheduling performance goals.

Scheduling Periodic Tasks	Meet all deadlines Maximize processor utilization Maximize average earliness Minimize average lateness
Scheduling Multiprocessors	Minimize number of processors Maximize the average processor utilization
Scheduling Aperiodic Tasks	Minimize average response time
Tasks Synchronization	Minimize average blocked time

Table I Summary of Performance Goals

1.4 Contributions

The research presented here incorporates several facets of the real-time scheduling problem. First, we present a survey of the previous work in scheduling periodic real-time tasks in uniprocessor and multiprocessor environments. This survey provides an up-to-date and comprehensive compilation of the algorithms and theories introduced by researchers in this field.

We also present the results of an empirical study we conducted to compare a number of multiprocessor scheduling algorithms and to test their suitability for different task sets. The results from the study will help us understand what algorithms are good for what types of task sets.

Next, we extend the problem definition by relaxing some of the original restrictions enforced in the previous work in periodic task scheduling. We allow the target system to be heterogeneous and incorporate communication cost in modeling the problem of scheduling periodic tasks. Developing new algorithms and models adds to the current field of real-time scheduling.

We also present a tool for evaluating periodic task sets against scheduling algorithms. No known standard testbed exists to verify new task sets against the latest algorithms in literature. By modifying the parameters of task sets, designers can tune the performance of their system by running "what if" scenarios against different heuristics without compromising task deadlines.

The remainder of this thesis is organized as follows: Previous work is covered in Chapter 2. In Chapter 3, we present the results of our empirical study of rate-monotonic multiprocessor scheduling algorithms. Chapter 4 extends the original problem definition to allow heterogeneous processors, introduces three heuristics to solve the extended problem, and evaluates the algorithms. In Chapter 5, we expand the model of scheduling

periodic tasks to incorporate communication. We also introduce a heuristic for scheduling with communication and evaluate its performance. We introduce our scheduling tool in Chapter 6 and give our conclusions in Chapter 7.

2 Previous Work

Though the first published result of scheduling periodic tasks appeared over 20 years ago, the field of scheduling periodic tasks still sees extensive success in the literature today. This chapter presents a summary of results in the area of scheduling real-time tasks. We begin the summary with a detailed look at the foundation of the rate-monotonic theory. Then, we examine extensions to the fundamental problem by adapting the theory to multiprocessor systems.

2.1 Scheduling on Uniprocessors

In their 1973 paper, Liu and Layland [1] set the foundation for scheduling periodic tasks in a real-time environment. They studied the scheduling problem under the following assumptions:

1. The request for all tasks are periodic with constant intervals between request. Tasks have hard deadlines.
2. Each task must complete before the next request for the task. The deadline of a task equals the period of the task.
3. No precedence relationship, communication, synchronization, or exclusion between tasks. All tasks are completely independent.
4. The computation time for each task is constant and does not vary from run to run.
5. Aperiodic tasks can exist for initialization and error-recovery only, don't have hard deadlines, but do displace the periodic tasks.

Under these well defined restrictions, Liu and Layland established the basic theory of rate-monotonic scheduling. Moreover, most of the current research in scheduling periodic real-time tasks today is still centered around their theory. The rate-monotonic (RM) theory considers task periods the single most important factor when scheduling tasks. This policy assigns priorities to tasks in inverse order of their rate request; τ_i will have a higher priority than τ_j if $T_i < T_j$ [1]. A task having a high request rate (small period) will get a high priority independent of the amount of computation time required to complete the task. Rate-Monotonic priority designations are considered "optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be schedule by the rate-monotonic priority assignment." [1]

RM Schedulability Criteria

In addition to the priority assignment policy, Liu and Layland presented a schedulability criteria that defines a worst-case upper bound on the utilization of a task set. The task set utilization must be less than the bound in order to guarantee all tasks will meet their deadlines with the rate-monotonic priority rule.

Theorem 1. *Let* $\mathfrak{T} = \{\tau_i = (C_i, P_i) \mid i = 1, 2, ..., n\}$ *be a set of n tasks. If the total utilization of the task set satisfies:*

$$U \leq n\left(2^{\frac{1}{n}} - 1\right)$$

then all task deadlines will be met with the RM priority assignment policy [1].

We will refer to this function as the Rate-Monotonic (RM) criteria. The RM criteria determines if a task set is feasibly scheduled strictly on the knowledge of the number of tasks in task set. As the task set grows large, the least upper bound approaches *ln* 2 (.693) [1].

Since the rate-monotonic assignment rule was proven "optimal" and the RM schedulability criteria produced low utilization rates for large task sets, work by other researchers continued in area of new schedulability functions. The goal is to increase the utilization of the task set on the processor and still guarantee deadlines under the rate-monotonic rule.

IP Schedulability Criteria

The next schedulability criteria appearing in literature is based on ordering the task set before determining if it is schedulable. In [2], Dall and Liu determined that by sorting the task set in non-decreasing order of period values and applying the schedulability function one task at a time, one can achieve higher utilization rates.

Theorem 2. *Let* $\mathfrak{I} = \{\tau_i = (C_i, P_i) \mid i = 1, 2, ..., n\}$ *be a set of n tasks with* $T_1 \leq ... \leq T_n$.

Let $u = \sum_{i=1}^{n-1} \frac{C_i}{T_i} \leq (n-1)(2^{\frac{1}{n-1}} - 1)$.

If the utilization of task τ_n satisfies: $C_n/T_n \leq 2\,(1+u/(n-1))^{-(n-1)} - 1$, then all task deadlines will be met with the RM priority assignment policy [2].

We will call this criteria Increasing Period (IP). As the number of tasks in the set approach infinity, the load of task τ_n approaches 2^{-u}-1 [2].

UO Schedulability Criteria

Recently, in [4] Oh and Son provided a schedulability criteria based strictly on the utilization's of the task set. By including the utilization information in the algorithm, Oh and Son were able to achieve higher loads on the processor than the two previous algorithms. The following theorem defines the enhanced function.

Theorem 3. *Let $\mathfrak{T} = \{\tau_i = (C_i, T_i) \mid i = 1, 2, ..., n\text{-}1\}$ be a set of (n - 1) tasks with utilization's of $\{u_1, u_2, ..., u_{n-1}\}$, and feasibly scheduled by the rate-monotonic algorithm. A new task $\tau_n = (C_n, T_n)$ can be feasible scheduled together with the (n - 1) tasks on a single processor by the rate-monotonic algorithm, if*

$$\frac{C_n}{T_n} \leq 2\left[\prod_{i=1}^{n-1}(1+u_i)\right]^{-1} - 1 \text{ [4].}$$

Since this condition only considers utilization when determining the feasibility of a task set, it is called Utilization Oriented (UO).

PO Schedulability Criteria

Appearing in literature soon after the UO criteria, a new schedulability criteria founded on task periods was presented. This new function makes use of an idea alluded to in the original work of Liu and Layland specifying if the tasks are "suitably related," higher processor utilization's are attainable [1]. Let's look at two definitions before examining the new function [5].

Let $\mathfrak{I} = \{ \tau_i = (C_i, T_i) \mid i = 1, ..., n\}$ be a set of n tasks. We define the following:

$$S_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor \text{ for } i = 1, ..., n \text{ and}$$

$$\beta = \max_{1 \le i \le n} S_i - \min_{1 \le i \le n} S_i$$

We now define the Period Oriented schedulability criteria.

Theorem 4. *Let $\mathfrak{I} = \{\tau_i = (C_i, T_i) \mid i = 1, 2, ..., n\}$ be a set of n real-time tasks. If the total load satisfies*

$$U \le \max\{\ln 2 - \beta \ln 2\}$$

the task set can be scheduled on one processor [5].

The value β is defined as the largest difference between all the S_i values of the tasks in the set. Or more appropriately, how close the task periods are to each other within the task set. As β gets small, task periods are close to each other and the possible processor utilization will get large (up to 100%). Notice further that the other term in the "max" statement is the same as the upper bound specified in the rate-monotonic schedulability criteria of Liu and Layland, implying the worst the Period Oriented (PO) schedulability criteria will do is no worse than the upper bound on the original algorithm.

The RM, IP, UO, and PO schedulability criteria all specify necessary conditions which must be met in order to use the rate-monotonic priority assignment rule for periodic tasks. A necessary condition in the context of rate-monotonic scheduling provides a criterion which must be met in order to guarantee task deadlines when using the rate-monotonic assignment scheme. If the criterion is not met, the task set may or may not be scheduled with the RM priority scheme. Conversely, a necessary and sufficient condition specifies if the condition is satisfied task deadlines are guaranteed to be met, but if the condition is not met, the RM priority rule cannot guarantee all task deadlines. A necessary and sufficient condition is also called an exact characterization of the task set.

EX Schedulability Criteria

Lehoczky, Sha, and Ding [17] provide a necessary and sufficient condition to guarantee task deadlines will be met when using the rate-monotonic priority assignment strategy. The exact characterization makes use of the following function known as the completion time test:

Theorem 5. *Given tasks* $\tau_1, \ldots, \tau_i$. *We can determine the total demand for the processor in the time frame [0, t] by the expression:*

$$W_i(t) = \sum_{j=1}^{i} C_j \times \left\lceil \frac{t}{T_j} \right\rceil$$ [17].

Two important results from the Liu and Layland paper helped set the stage for developing the exact characterization. First, a critical instance occurs when all tasks are ready to execute at the same time. The best example is time 0. Second, the entire task set is schedulable if the first instance of each task meets its deadline [1]. Therefore, one only needs to test from time 0 up to the largest deadline to determine if all tasks are schedulable.

We define the following:

$L_i(t) = W_i(t) / t$ (as the ratio of the demand on the processor up to time t)

$L_i = \min \{0 < t \leq T_i\} L_i(t)$ (as the smallest ratio of demand to time on the processor over the entire period of τ_i

$L = \max \{1 \leq i \leq n\} L_i$ (as the largest ratio of demand to time on the processor for all tasks in the task set)

Given these definitions, the following Theorem states the exact characterization of the task set as:

Theorem 6. *Let* $\mathfrak{I} = \{\tau_i = (C_i, T_i) \mid i = 1, 2, ..., n\}$ *be a set of n tasks assigned priorities with the rate-monotonic rule.*

- *Task* τ_i *will meet all deadlines under all phasings iff* $L_i \leq 1$, *and*
- *All tasks in the task set will meet all deadlines iff* $L \leq 1$ [17].

With the EX algorithm we know for sure whether our task set is schedulable under the rate-monotonic priority assignment scheme or not. But, this knowledge does not come

without added cost. As we shall see, the performance of the algorithm with respect to the size of the task set can be unbounded.

Complexity Analysis

One of the critical evaluations of an algorithm is the complexity analysis. Researches not only desire algorithms to produce correct results, there is a never ending goal of searching for as low time complexity as possible. The time complexity of an algorithm is a measure of its running time as the size of the input data grows very large. The following table compiled from several sources enumerates the complexity in relation to the size of the task set for the single processor scheduling criteria [1][2][4][5][17][24].

Criteria	Time Complexity
RM	O (n)
IP	O (n log n)
UO	O (n)
PO	O (n)
EX	Unbounded

Table II Single Processor Time Complexity Analysis (n = number of tasks)

2.2 Scheduling on Multiprocessors

The algorithms just presented all provide schedulability criteria on single processor systems. Optimality of the single processor RM criteria, in the sense of priority

assignments, was proven early on by Liu and Layland. Conversely, in [20] Leung and Whitehead showed that finding an optimal assignment of periodic tasks in a multiprocessor environment is computationally intractable (NP-Hard). Since an optimal algorithm is not possible without searching for all possible task orderings and choosing the best one, researchers have turned to heuristics in order to find sub-optimal schedules. Based on intuition, heuristics provide non-optimal answers in less than exponential time by exploiting some characteristic of the system. Their effectiveness depends on the nature of the task set and the target system.

Rate-Monotonic-First-Fit Scheduling

Before Leung and Whitehead proved multiprocessor scheduling of rate-monotonic tasks was NP-Hard, Dhall and Liu presented the first heuristic for allocating periodic tasks to multiprocessor machines. The rate-monotonic-First-Fit Scheduling (RMFFS) heuristic makes use of the first-fit bin-packing algorithm combined with the IP schedulability criteria [2]. The RMFF algorithm is shown below.

```
        Rate-Monotonic-First-Fit Scheduling

Sort tasks by increasing period
FOR  i  = 1 to n DO
      j = 1
      done = False
      DO
            u  = Σ of all u_j on ρ_j
            k = number of tasks on ρ_j
            IF u_i ≤ 2(1 + u/k)^-k - 1 THEN
                  add τ_i to ρ_j
                  done = True
            ELSE
                  j = j + 1
            END IF
      WHILE done = False
END FOR
```

The heuristic assigns tasks to a processor one at a time until the schedulability criteria is no longer met. At that time, another processor is allocated. This procedure continues until all tasks have been assigned to a processor.

With the RMFFS algorithm, Dall and Liu set the foundation for almost all of the current multiprocessor periodic task scheduling heuristic algorithms. All of the multiprocessor scheduling heuristics reviewed in this thesis combine one of the single processor schedulability criteria with some form of a bin-packing heuristic.

First-Fit Decreasing Utilization Factor

Davari and Dhall improved upon the RMFFS algorithm with the First-Fit Decreasing Utilization Factor (FFDUF) heuristic [3]. By changing the sort order from increasing

periods to decreasing utilization values, utilization rates on processors was increased thereby decreasing the number of processors required to schedule the task set.

Rate-Monotonic - First Fit Decreasing Utilization

Similar to how Davari and Dhall improved upon RMFFS, Oh and Son improved upon Davari and Dhall's FFDUF. The Rate-Monotonic - First Fit Decreasing Utilization (RM-FFDU) combines a first-fit decreasing bin-packing heuristic with the UO schedulability criteria [4]. The RM-FFDU algorithm is presented below.

Rate-Monotonic-First Fit Decreasing Utilization

```
Sort tasks by decreasing utilization
FOR  i  = 1 to n DO
       j = 1
       done = False
       DO
              k = number of tasks on ρ_j
              IF u_i ≤ 2 / ∏_{l=1}^{k} (u_l + 1) THEN
                     add τ_i to ρ_j
                     done = True
              ELSE
                     j = j + 1
              END IF
       WHILE done = False
END FOR
```

Rate-Monotonic General-Task

The last multiprocessor heuristic we shall cover makes use of the PO schedulability function and the special property of the rate-monotonic priority rule. By assigning tasks

with periods close to each other on the same processor, Burchard, Liebeherr, Oh, and Son attempt to increase the load on each processor resulting in a decrease of the number of processors required [5]. In order to achieve the higher utilization rates, two functions are specified: Rate-Monotonic Small Task (RMST) is applied to tasks with low utilization rates and Rate-Monotonic General Task (RMGT) is applied to the remainder of the tasks in the set [5].

Rate-Monotonic Small Task

Sort the task set by increasing S_i (as previously defined) values
$i = 1$
$j = 0$

(2) $j = j + 1$
Allocate ρ_j
Add τ_i to ρ_j
$U_j = u_i$
$S = S_i$
β_j

(3) $i = i + 1$
$\beta_j = S_i - S$
IF $u_i + U_j \leq \max\{\ln 2, 1 - \beta_j \ln 2\}$ THEN
Add τ_i to ρ_j
$U_j = U_j + u_i$
GOTO 3
ELSE
GOTO 2

Where RMST attempts to place as many tasks on a processor as it can, the G_2 part of heuristic RMGT will place a maximum of two tasks on a processor.

Rate-Monotonic General Task

Partition the set of tasks into two groups:

$G_1 = \{ \tau_i \mid u_i \leq 1/3 \}$

$G_2 = \{ \tau_i \mid u_i > 1/3 \}$

Schedule G_1 using RMST.

Schedule G_2 as follows:

```
        i = 1
        j = 1
        Add τ_i to ρ_j
        U_j = u_i

(1)     i = i + 1
        j = 1
        done = False
        DO
              IF ρ_j has only one task THEN

                    IF T_i < T_j THEN
                          IF T_j ≥ ⌈T_j/T_i⌉ C_i + C_j THEN
                                add τ_i to ρ_j
                                U_j = U_j + u_i
                                done = True
                          END IF
                    ELSE
                          IF T_i ≥ ⌈T_i/T_j⌉ C_j + C_i THEN
                                add τ_i to ρ_j
                                U_j = U_j + u_i
                                done = True
                          END IF
              ELSE
                    add τ_i to ρ_j
                    U_j = U_j + u_i
                    done = True
              END IF
              IF done = False THEN
                    j = j + 1
              END IF
        WHILE done = False
        GOTO 1
```

Thus far, we have covered almost all of the multiprocessor periodic task allocation heuristics for the rate-monotonic priority sequence. Before concluding this chapter, we shall examine the performance bounds and complexity of the multiprocessor heuristic algorithms.

Complexity Analysis

In his doctoral dissertation [24], Oh presents the results from a complexity analysis and theoretical performance bounds study performed on existing and proposed multiprocessor algorithms. This study is of value to us because later in the thesis we present the outcome of our empirical of the average case behavior of the same algorithms.

To compare the bounds of the algorithms, we need to look at how many processors were required by the heuristic (N_A) as compared to the number of processors for the optimal assignment (N_{opt}). This ratio is evaluated as number of tasks approaches infinity. More formally:

Theorem 7. *Let the worst case asymptotic bound for a multiprocessor algorithm be defined as:*

$$\Re^{\infty} = \lim_{N_{opt} \to \infty} \frac{N_A}{N_{opt}} \text{ [24]}$$

Since N_A / N_{opt} is a simple ratio, it can never be less than 1. Furthermore, lower ratios indicate the algorithm will produce results closer to the optimal number than higher

ratios. The table below shows the bounds and complexity for most of the multiprocessor algorithms existing in the literature [4][5][17].

Algorithm	$\Re\infty$	Time Complexity
RMFFS	2.33	$O(n \log n)$
FFDUF	2	$O(n \log n)$
RM-FFDU	1.67	$O(n \log n)$
RMST	$1/(1-\alpha)$	$O(n \log n)$
RMGT	1.75	$O(n \log n)$

Table III Asymptotic Bound and Complexity of Multiprocessor Algorithms

(α = maximum load of any task)

As seen in the table, the bound of RMST is in direct relation to the largest load ratio, α, of any task in the set [5][17]. If the maximum load a task places on the processor is 0.2, the upper bound will be 1.25. A high load task set with a maximum load value of 0.8, can create a theoretical upper bound of 5. Much worse than other algorithms.

2.3 Summary

In this chapter, we reviewed the most well known schedulability criteria for single processor systems. Next, we examined several allocation schemes based on the single processor criteria for scheduling periodic tasks in multiprocessor environments. As the development of the criteria functions progressed over time, the load allowed on the processors by the single processor functions increased. This increased utilization enabled

the multiprocessor algorithms to attain higher processor utilization rates and therefore require less processors for large task sets.

As we have shown, the schedulability criteria provides only a necessary condition when determining if the task set is feasible. Though one can use the exact characterization to determine if the task set is feasible, the complexity is unbounded with respect to the size of the task set. Additionally, from the performance bounds study by Oh in [24], we can only achieve close to 100% processor utilization with the RMST criteria and task sets with extremely low load ratios. The theoretical performance of the algorithms concerning the utilization bounds and algorithm complexity provide a good measure of how the heuristics react with very large task sets. What is also important is the average case behavior of the algorithm when studied under a more realistic environment. In the next chapter we present an empirical evaluation of the algorithms presented here to determine their average case performance.

3 Comparison of Multiprocessor Scheduling Heuristics

In the previous chapter we presented a current set of periodic task multiprocessor allocation heuristics. We showed the complexity and worst-case performance bounds of how the algorithms react to extremely large task sets. In this chapter, we present the results of a number of experiments we conducted to evaluate the average case behavior of the algorithms. We'll begin by enumerating the algorithms implemented and evaluated. Following the algorithms, we present the details of the evaluation environment and experimentation parameters. The last section will discuss the results of the study.

3.1 Heuristic Algorithms

For this study, we implemented three of the multiprocessor algorithms from the previous chapter and adapted two of the single processor schedulability criteria for allocating tasks on multiprocessor systems. The three multiprocessor heuristic algorithms implemented were Rate-Monotonic-First-Fit Scheduling (RMFFS), Rate-Monotonic First-Fit Decreasing Utilization (RM-FFDU), and Rate-Monotonic General Task (RMGT). In addition, we adapted the Rate-Monotonic (RM) and Exact Characterization (EX) single processor schedulability criteria to allocate tasks for multiprocessors. Though a multiprocessor algorithm with EX has not appeared in the published literature, Oh proposed a rough algorithm [24].

The transformation of the RM and EX criteria to multiprocessor scheduling simply consisted of applying the schedulability criteria with a first-fit bin-packing heuristic. Basically, we took the RMFFS algorithm and replaced the IP condition with EX and RM. Additionally, the RM multiprocessor algorithm did *not* prepare the task set by sorting it first, while the EX multiprocessor algorithm placed all the tasks in increasing order of periods before assigning them to processors. Our preparation of the tasks in the EX multiprocessor heuristic differs than the decreasing utilization sort given in [24]. Since the completion time test evaluates the load on the processor in an increasing time sequence, we conjectured an increasing period order would yield better results than a decreasing utilization sequence.

As an example of our modifications, below is the multiprocessor RM followed by the multiprocessor EX.

Multiprocessor RM

```
FOR  i  = 1 to n DO
       j = 1
       done = False
       DO
             x = number of tasks on ρ_j
             IF u_i + U ≤ (x + 1)(2^(1/(x+1)) - 1) THEN
                   add τ_i to ρ_j
                   done = True
             ELSE
                   j = j + 1
             END IF
       WHILE done = False
END FOR
```

```
                    Multiprocessor EX

Sort tasks by increasing period
FOR  i  = 1 to n DO
       j = 1
       done = False
       DO
              t = completion time of all tasks on ρj including τi
              Di = deadline of τi
              IF t ≤ Di THEN
                     add τi to ρj
                     done = True
              ELSE
                     j = j + 1
              END IF
       WHILE done = False
END FOR
```

We chose to implement the RM and EX criteria as multiprocessor allocation schemes in attempt to rate the "standard" algorithms against what should be worst (RM) and best (EX) algorithms. The following list summarizes the heuristic algorithms evaluated in this study:

- RM Multiprocessor
- RMFFS
- RM-FFDU
- RMGT
- EX Multiprocessor

3.2 Evaluation Methodology

For this study, we wanted to examine how the algorithms perform under two different situations: increasing loads and different data distributions. In order to accomplish this

type of experimentation, we generated random task sets with different input parameters, tested each algorithm against the task set, and recorded the results.

Workload

We use random number generators to create the workload used in our experiments. In order to avoid any bias that may result from using an inaccurate random number generator, we implemented RAN3 from [13]. This generator employs a *subtractive method* to obtain a random number with a uniform deviate and has no known weaknesses [13]. In order to meet the goal of testing the algorithms against different data distributions, we also implemented the functions necessary to transform random numbers with a uniform deviate to either a normal, or exponential deviate.

Each task in the task set contained a randomly generated period (T_i) and computation time (C_i). The period was a random number between 20 and 500; $20 \leq T_i \leq 500$. A maximum period of 500 allows large task sets the opportunity to contain many different task periods. If we had used a smaller number, many tasks would have had the same period. On the low end, we chose 20 as the minimum period to prevent a task from "dominating" a processor. With tasks having periods greater than 20, we can allocate more tasks to each processor.

After generating the period for a task, we generate the computation time. Since a task's utilization is a function of its computation time and period, we can generate a random

computation time as a ratio of the period to obtain a target utilization. We define the load ratio, α, as the maximum load a task places on a processor in relation to its period. A task with a low utilization will have a small α close to 0. Similarly, a high load task will have an α close to 1.0. To control the load a task will place on the processor, we generate a random computation time between 1 and αT_i; $1 \leq C_i \leq \alpha T_i$.

With the capability to control the load of the tasks in the set and the different data distributions, we have the tools necessary to measure the performance of the algorithms.

Performance Measures

To evaluate the algorithms, we gathered three different performance measures: Percent Extra Processors, Average Processor Utilization, and the Standard Deviation on the number of processors required for the heuristic. The primary metric is the Percent Extra Processors.

Percent Extra Processors (PEP) measures how many processors the heuristic algorithm required to schedule the task set against what an optimal schedule would need. Although for large task sets we can't determine the optimal solution, we can approximate it by using the total utilization of the task set. Hence, the optimal number of processors required assumes 100% processor utilization and is calculated as $N(O) = \sum_{i=1}^{n} C_i/T_i$.

The number of processors required by the heuristic is labeled $N(\mathfrak{I})$. Therefore, the PEP is computed as :

$$\frac{N(\mathfrak{I}) - N(O)}{N(O)} \times 100$$

The lower the percentage the better the performance.

Unlike the PEP, the goal is to achieve high Average Processor Utilization (APU) rates. The APU is simply the mean of the utilization for each processor averaged over all the samples of the experiment.

Experiments

Our evaluation involves nine experiments with ten runs in each experiment. Each run, increases the number of tasks by 100. Furthermore, each run has a sample size of 50 task sets. Other researchers selected sample sizes of 15 and 20 in their comparisons [5][24], but we chose 50 to ensure there were enough data points to obtain a good average of the results.

Experiments 1, 2, and 3 will indicate how an increasing task load with uniform deviate affects the number of processors required by the scheduling algorithms. Similarly, experiments 4 though 6 and, 7 through 9, show the processor requirements under normal and exponential distributions, respectively. Table IV summarizes the evaluation parameters for all experiments.

	Minimum Period	Maximum Period	Sample Size	Load Ratio	Distribution
1	20	500	50	**0.1**	Uniform
2	20	500	50	**0.5**	Uniform
3	20	500	50	**0.9**	Uniform
4	20	500	50	**0.1**	Normal
5	20	500	50	**0.5**	Normal
6	20	500	50	**0.9**	Normal
7	20	500	50	**0.1**	Exponential
8	20	500	50	**0.5**	Exponential
9	20	500	50	**0.9**	Exponential
Number of Tasks: {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}					

Table IV Experimentation Parameters

We can also look at experiments 1, 4, and 7 to determine how the heuristics perform with different distributions under low utilization conditions. Experiments 2, 5, 8 and 3, 6, 8 can be viewed in the same manner for medium and high utilization task sets, respectively.

3.3 Results

The goal of this experiment is to evaluate how the heuristics react to increasing task loads, increasing numbers of tasks, and different data distributions. The figures on the following pages show the Percent Extra Processors results from the experiments performed.

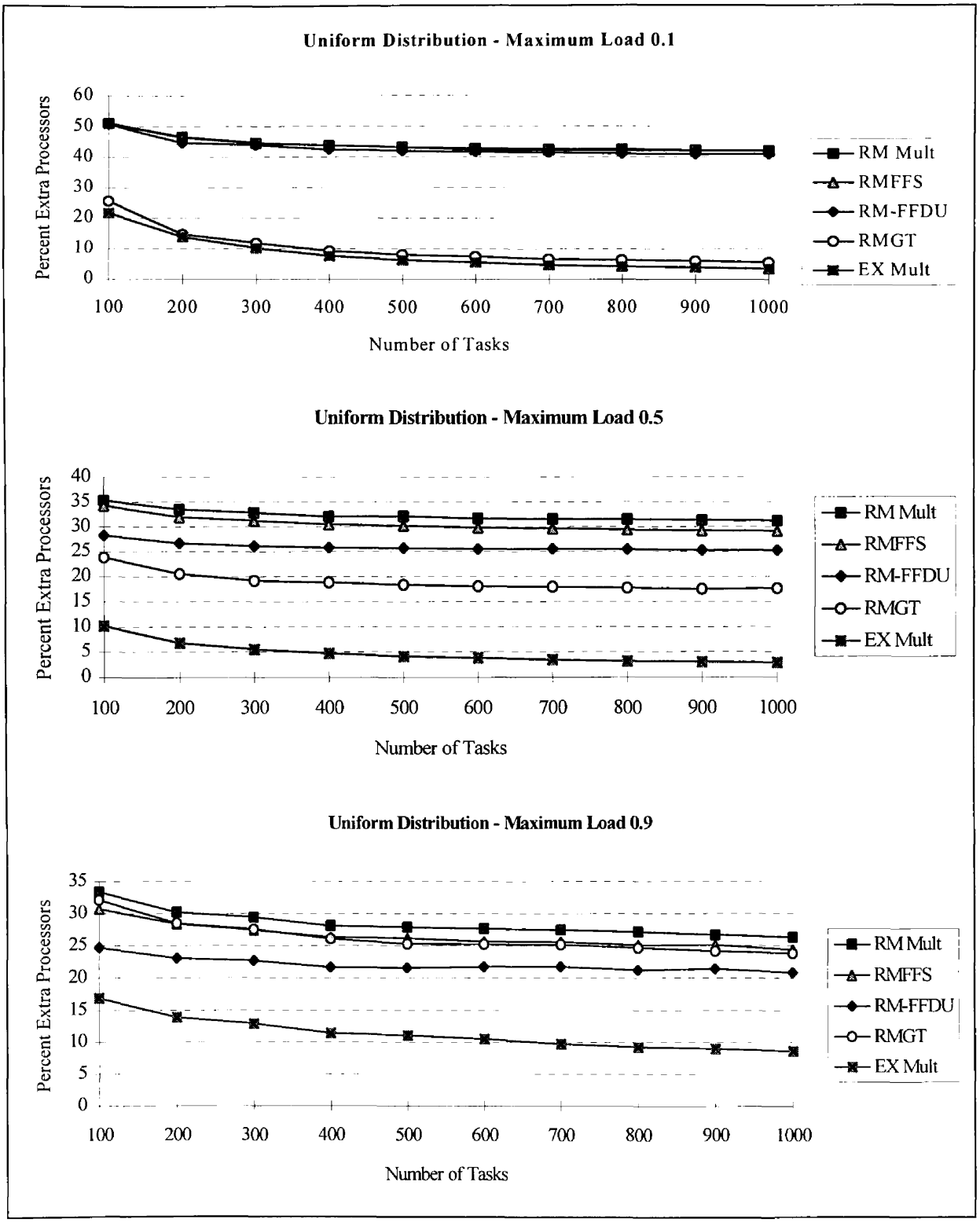


Figure 1 - Uniform Distribution Results

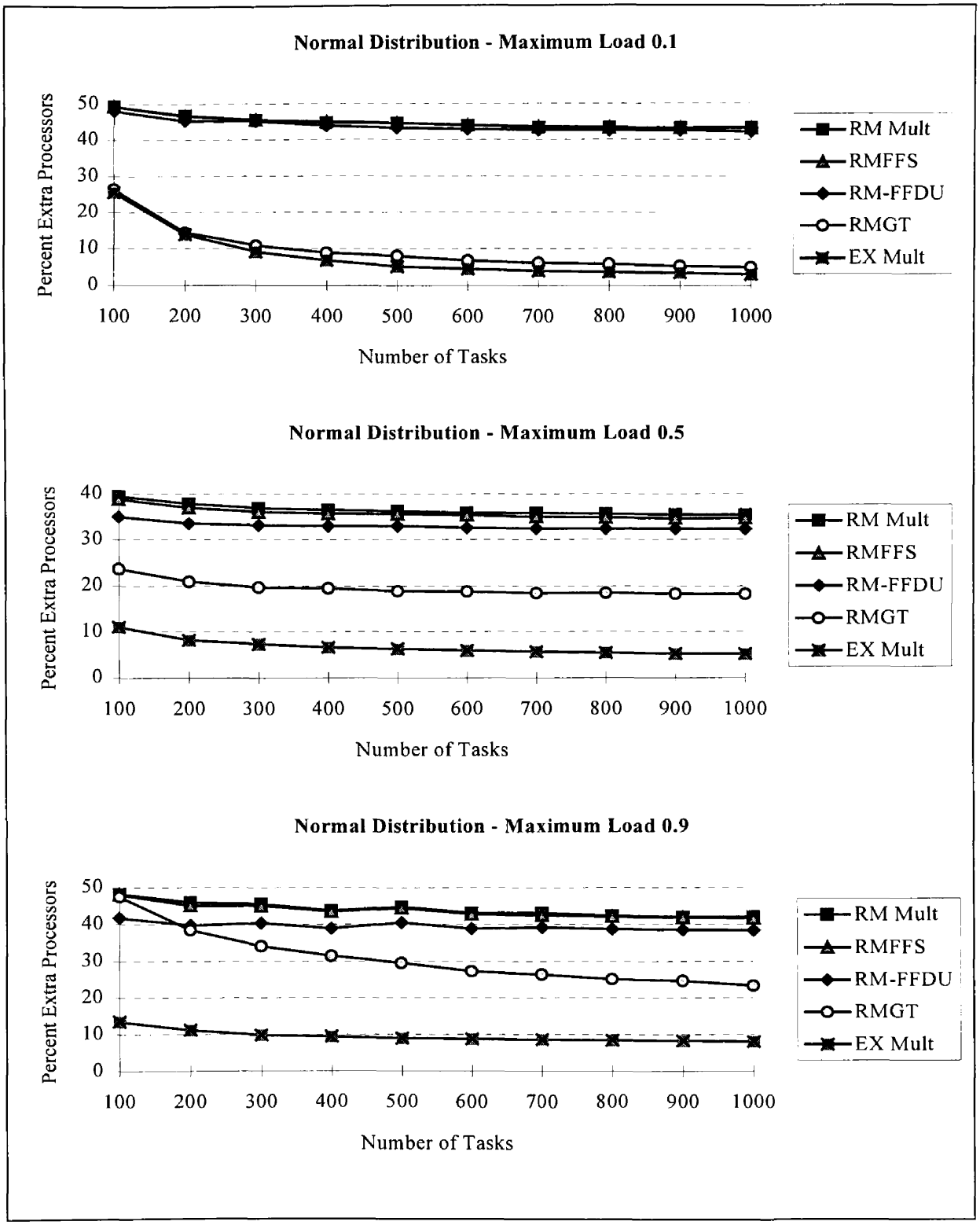


Figure 2 - Normal Distribution Results

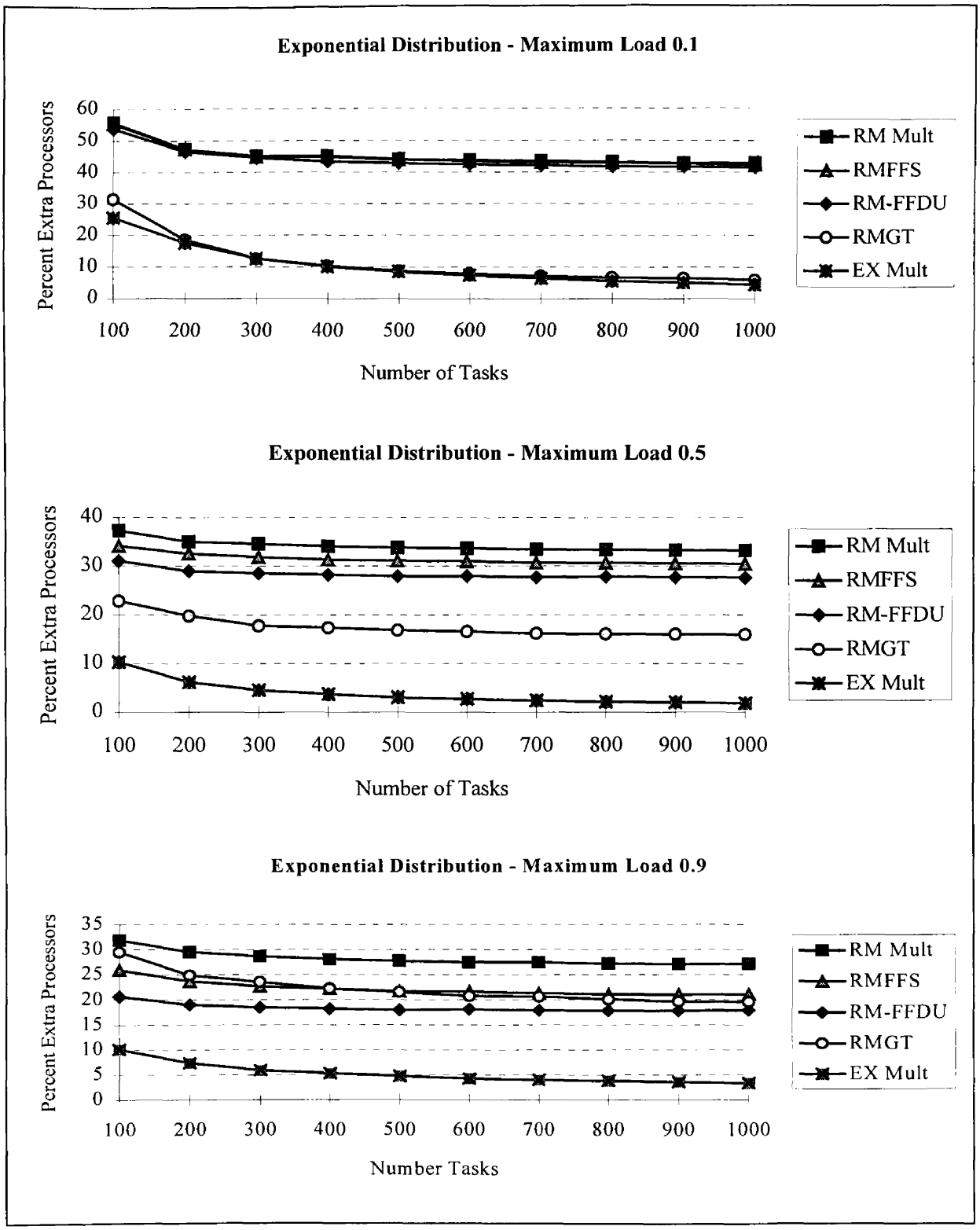


Figure 3 - Exponential Distribution Results

As shown in Figure 1, Figure 2, and Figure 3 at low and medium utilization values RMGT and EX Mult outperform the other heuristics over all three distributions. It is also shown that both RMGT and EX Mult achieve less than 10% extra processors when the workload consists of large number of tasks. At high task loads RM-FFDU generally performs better than RMGT.

With the exception of the RMGT algorithm, increasing task loads didn't have any dramatic effect on the results. Generally, as the loads increased the relative ordering of the heuristics was consistently: RM Mult (worst), RMFFS, RM-FFDU, then EX Mult (best). Only RMGT displayed any change in behavior related to a change in load. Examining Figure 1, Figure 2, and Figure 3, we can see the sensitivity of RMGT to the increasing load from 0.1 to 0.9. This response is understandable because the main benefit RMGT adds to multiprocessor allocation problem is the enhanced schedulability of low utilization task sets by the RMST heuristic. As task loads increase, RMST schedules less and less tasks.

Again, viewing Figure 1, Figure 2, and Figure 3 we see that for low task loads, EX and RMGT are the algorithms of choice. But when the task loads become very large, the utilization oriented nature of RM-FFDU enables it to utilize fewer processors than RMGT. RM Mult consistently requires more processors than the other heuristics while EX Mult always provides the best results. This verifies our assumption of binding the problem within the worst and best heuristics.

The only correlation between data distributions and performance was for high utilization task sets and RMGT. All other heuristics demonstrated the same basic response without regard to data distribution or task load. The RM-FFDU algorithm, based on the UO criteria, required less processors than RMGT for uniform and exponential distributions. See Figure 1 and Figure 3, Maximum Load 0.9. However, for high load normal distributions RMGT needed less processors then RM-FFDU. Figure 2, Maximum Load 0.9. This exception is a direct result of the nature of the normally distributed task sets and the RMGT algorithm. In order to produce a random number with a normal deviate, the generator basically returns the same number twice. First as a positive difference from the target mean, then on the next call the negative difference is returned. Furthermore, knowing the outline of the standard bell curve with a target utilization of 0.5 (Experiment 5), we can conclude most of the utilization values should be roughly within the range of 0.35 to 0.65. This appears to be the target utilization range of the second heuristic inside RMGT. Since the G_2 portion of RMGT only attempts to place two tasks on each processor, the range of 0.35 to 0.65 is prime for achieving high utilization rates on a single processor.

In addition to examining how each algorithm reacts to varying loads and different distributions, it is also good to look at the average utilization per processor resulting from the schedules. Figure 4 represents the average processor utilization, by task load, for the three different data distributions.

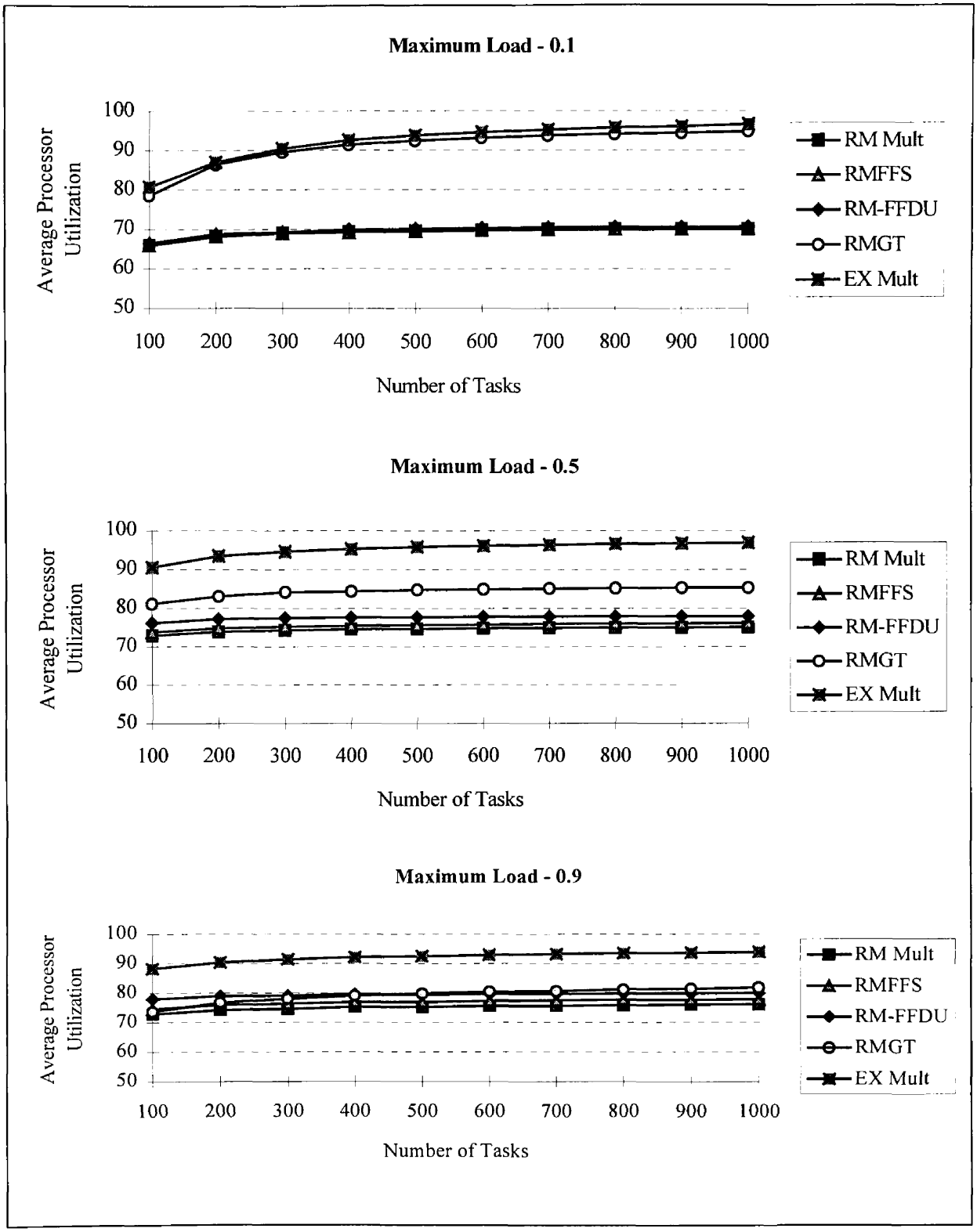


Figure 4 - Average Processor Utilization

These graphs show EX Mult and RMGT achieving greater than 90% utilization for large task sets. The remainder of the heuristics group themselves around 70% - 80% APU. In general, the EX allocation scheme achieves the highest Average Processor Utilization and RM Mult the lowest. RMGT's average utilization would fluctuate between the mid 90% range and low 80% range inversely with the Percent of Extra Processors required.

Quality of experiments

The last statistic we shall present is the Standard Deviation on the number processors required for each algorithm. Since each test contained multiple samples, the Standard Deviation is actually the average standard deviation from all test runs. We shall only present results from one test that typifies the entire experiment.

	RM Mult		RMFFS		RM-FFDU		RMGT		EX Mult	
Tasks	Avg. Procs	Std Dev	Avg. Procs	Std Dev	Avg. Procs	Std Dev	Avg. Procs	Std Dev	Avg. Procs	Std Dev
100	33	1.91	33	1.90	31	1.73	30	1.85	27	1.60
200	66	2.83	65	2.89	62	2.50	59	2.60	53	2.33
300	99	3.21	98	3.47	94	3.01	89	3.25	79	2.96
400	132	3.08	130	2.99	125	2.84	118	3.08	104	2.64
500	165	4.14	162	4.18	157	3.75	147	4.04	130	3.53
600	197	4.34	194	4.53	188	4.14	177	4.06	155	3.91
700	229	4.01	226	3.90	219	3.77	206	4.06	180	3.49
800	261	6.09	257	6.18	249	5.68	234	6.16	205	5.26
900	297	4.16	292	4.36	283	3.98	266	4.21	233	3.58
1000	328	5.28	323	5.39	314	4.78	295	5.28	257	4.46

Table V Average Number Processors & Standard Deviations

All heuristics produced low Standard Deviation values relative to their average number of processors required. The low deviations reflect the quality of the random numbers and indicate how close the computed means represent the average results throughout the study. Again, this table reflects the Standard Deviation values for the entire experiment.

3.4 Summary

This study investigated how the multiprocessor periodic task allocation heuristics react to task sets with increasing loads and different data distributions. We began by describing the heuristics under study including the adaptation of two single processor scheduling criteria for multiprocessor task allocation. Next, the experimentation conditions were explained. Random numbers are the crux of every simulation study, so an in-depth discussion explained how the task periods and computation times were generated. Lastly, we presented the results of the study. The EX Mult heuristic consistently provided the best results in all experiments. When only considering linear heuristics, RMGT proved the best algorithm for most task sets. If a heuristic is required for very high load task sets, one should use RM-FFDU instead of RMGT. These results mirror the asymptotic bound analysis presented at the end of the previous chapter. Of the heuristics with bounds presented, RMFFS produced the worst average case performance and the highest bound. RMGT performs well for low and medium loads (indicating the performance of RMST), but at high loads it requires more processors than RM-FFDU, also reflected in Table III.

Even though the theoretical time complexity is unbounded with respect to number of tasks, we consider the EX Mult algorithm a viable source for allocating large task sets; up to 1000 tasks. Indeed, the algorithm did take substantially longer to produce output; but not unduly so. Testing was performed on a 60Mhz Pentium processor and the EX Mult algorithm only required approximately five minutes to schedule 50 task sets with 1000 tasks in each set.

4 Task Scheduling in Heterogeneous Environments

An underlying assumption in the current research of applying rate-monotonic scheduling to multiprocessor systems is all processors are homogeneous. We will remove the restriction requiring all processors to be the same and allow systems to have two types of processors. Processors will either be standard or "special." Special processors have characteristics which allow them to execute certain types of code faster than the standard processors. Examples of special processors can include Digital Signal Processors (DSP), Floating Point, Multimedia, I/O, and Network Communication processors. Tasks can execute on either the standard or special purpose processors, but will gain some computational time savings by executing on the special processor. In addition, we propose three allocation heuristics for solving the new problem and test them to determine if one consistently produces better results than the others.

4.1 System Model

The updated system model defines the task set $\Im = \{ \tau_i = (T_i, C_i, \Delta_i) \mid i = 1, ..., n\}$ with n periodic tasks and each task τ_i consisting of a period, T_i, the amount of computation time required for execution, C_i, and a delta value of Δ_i. The delta value represents how much computational savings τ_i will realize by executing on one of the special processors. We denote δ as the computation time of the task on a special processor and δu_i as the

utilization of the task executing on a special processor. For example given a task τ_1 = (20, 6, 2), the task has a period of 20 and a computation requirement of 6. However, if τ_1 executes on one of the special processors it will only require 4 ($\delta = C_i - \Delta_i$) time units to execute. Our updated target machine is defined as: $M = \{\rho_j \mid j = 1, ..., m; \omega_k \mid k = 1, ..., s\}$ of m standard processors and s special purpose processors. Furthermore, for this thesis, the heuristics restrict the number of special processors but allow unlimited standard processors. Along with updating the task set definition and target machine, we must also add a new performance measure.

All of the performance measures stated in the Chapter 1 apply to the system model incorporating heterogeneous processors in the rate-monotonic scheduling problem. Furthermore, we need to add the measure Percent Saved Utilization. This measure will help us differentiate the results of two algorithms when they both come up with schedules containing the same number of processors. Percent Saved Utilization is defined as the difference between the utilization of the task set as it would cost if executing on all standard processors and the task set utilization as scheduled by the heuristics. Formally:

$$\text{Task Set Utilization} = U_c = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

$$\text{Saved Utilization} = U_\Delta = \sum_{i=1}^{n} \frac{C_i - \Delta_i}{T_i}$$

$$\text{Percent Saved Utilization} = \frac{U_\Delta - U_c}{U_c} \times -100$$

We now have a new system model for relaxing the restriction mandating homogeneous processors in the rate-monotonic scheduling problem. For this thesis, we introduce three heuristic task assignment schemes for solving the extended problem. Then we conduct a number of experiments to study the performance of these algorithms. We compare the performance of the three heuristic algorithms with each other and with optimal solutions

4.2 Heuristic Algorithms

We introduce three assignment strategies. The first heuristic, Largest Number of Tasks (LT), works on the assumption that we will use the smallest number of processors by scheduling as many tasks as possible on the special processor and still guarantee deadlines. Heuristic LT uses the IP schedulability criteria to determine if a set of tasks is feasible on a processor. The algorithm simply checks each task, in sorted order, against the current special processor. If the task can be feasible scheduled on the processor, it is assigned to it. Otherwise, the next special processor is checked. This continues until all special processors are full. Heuristic LT is shown below.

Heuristic LT

```
--  Schedule the special processors first
Sort the task set by increasing δu_i
i = 1
j = 1

WHILE done = False
      δu = Σ of all δu_i on ρ_j
      k = number of tasks on ρ_j
      IF δ u_i ≤ (1 + δu/k)^-k THEN
            add τ_i to ρ_j
            i = i + 1
      ELSE
            IF  j < s THEN
                  j = j + 1
            ELSE
                  done = True
            END IF
      END IF
END WHILE

-- Schedule the remainder of the tasks
IF some tasks still need to be scheduled THEN
      use RMFFS to schedule the rest of the task set on standard
      processors
END IF
```

The second heuristic algorithm places the emphasis on the gain a task realizes by executing on a special processors. Heuristic Greatest Benefit (GB) orders the tasks from largest benefit gained to smallest benefit gained. Similar to heuristic LT, GB uses the IP schedulability criteria to determine if the task set is feasible on a processor. On the other hand, where LT assigns tasks to the first available processor until the special processors are full, GB will use a form of first-fit bin-packing to allocate tasks to the special processors. It does this by testing every task on each of the special processors to determine if it can be feasibly scheduled on that processor. If a task can be feasibly

scheduled on a special processor it is allocated to the special processor. If not, it is tested on the next special processor. Heuristic GB is shown below.

Heuristic GB

```
--  Schedule the special processors first
{δ_i = u_i - δu_i | i = 1,  2, ... n}
Sort the task set from largest δ_i to smallest

FOR i = 1 to n DO
      j = 1
      done = False
      DO
            δu = Σ of all δu_i on ρ_j
            k = number of tasks on ρ_j
            IF δ u_i ≤ (1 + δu/k)^-k  THEN
                  add t_i to ρ_j
                  i = i + 1
            ELSE
                  IF  j < s THEN
                        j = j + 1
                  ELSE
                        done = True
                  END IF
            END IF
      WHILE done = False
END FOR

-- Schedule the remainder of the tasks
IF some tasks still need to be scheduled THEN
      use RMFFS to schedule the rest of the task set on standard
      processors
END IF
```

The last algorithm proposed for the thesis makes use of the idea that in a heterogeneous multiprocessor system computation time is the one parameter affecting the utilization of a task that is adjustable. By placing the tasks on the special processor with the largest computation time requirement, we can attempt to minimize the utilization of the tasks and thereby minimize the total utilization. Algorithm CT (Computation Time) does this by testing every task on each of the special processors to determine if it can be feasibly

scheduled on that processor. If a task can be feasibly scheduled on a special processor it is assigned to it. If not, it is tested on the next special processor. Heuristic CT is shown below

---

Heuristic CT

```
--  Schedule the special processors first
Sort the task set from largest C_i to smallest

FOR i = 1 to n DO
      j = 1
      done = False
      DO
            δu = Σ of all δu_i on ρ_j
            k = number of tasks on ρ_j
            IF δ u_i ≤ (1 + δu/k)^-k THEN
                  add τ_i to ρ_j
                  i = i + 1
            ELSE
                  IF  j < s   THEN
                        j = j + 1
                  ELSE
                        done = True
                  END IF
            END IF
      WHILE done = False
END FOR

-- Schedule the remainder of the tasks
IF some tasks still need to be scheduled THEN
      use RMFFS to schedule the rest of the task set on standard
      processors
END IF
```

The time complexity of the three heuristic algorithms is similar to the aforementioned multiprocessor heuristics in Chapter 2. Below, Table VI presents a quick synopsis of the time complexity of the proposed heterogeneous processor allocation heuristics.

Algorithm	Time Complexity
LT	$O(n \log n)$
GB	$O(n \log n)$
CT	$O(n \log n)$

Table VI Heterogeneous Processor Allocation Algorithms

(n = number of tasks)

Even though all three heuristics incorporate the IP schedulability criteria (including RMFFS) as the determining factor when allocating tasks to processors, almost any of the schedulability functions proposed in the literature could be used. Increasing Period function was chosen because it is fast and was the first to appear in literature but as long as all of the proposed heuristics use the same criteria, they can be objectively evaluated against each other. The three proposed algorithms were compared against each other in two simulation studies.

We compared the three proposed algorithms using two different methodologies. First, the heuristics were evaluated against each other with large task sets to determine their average case performance. Then, the heuristics were each evaluated against optimal solutions for small task sets.

4.3 Performance Evaluation of the Heuristics

Testing for the average case behavior consisted of generating a task set, scheduling it using each of the three algorithms, and savings the results. This process was repeated for

each sample in the test run. Goals of this study include determining which heuristic(s) produced consistently better results than the others, evaluating the effect of increasing task loads on the allocation of heterogeneous tasks, and analyzing how increasing the number of special processors changes the scheduling output.

Experiments

Six separate experiments were conducted with each experiment consisting of 10 distinct runs each with an increasing number of tasks in the task set. Every run contained a sample of 50 randomly generated task sets using a uniform distribution. Table VII summarizes the experimentation parameters.

	Minimum Period	Maximum Period	Distribution	Special Processors	Load Ratio	Sample Size
1	1	500	Uniform	4	**0.25**	50
2	1	500	Uniform	4	**0.50**	50
3	1	500	Uniform	4	**0.80**	50
4	1	1000	Uniform	**4**	0.50	50
5	1	1000	Uniform	**8**	0.50	50
6	1	1000	Uniform	**16**	0.50	50
Number of tasks in each test run: 20,40,60,80,100,200,400,600,800,1000						

Table VII Average Case Experimentation Parameters

Task periods and computation times were generated based on the same methodology presented in Chapter 3. Additionally, we must produce a value to represent the saved utilization; δu_i. This was accomplished by specifying a maximum allowed computation

time savings of 25% and generating a random number between the computation time and 25% of the computation time; $0.25\ C_i \leq \delta \leq C_i$. Thus, $\delta u_i = \delta / T_i$. The target computation time, δ, value was generated with the same distribution as the other variables in the experiment as specified above. Twenty five percent was basically an arbitrary choice for the maximum allowed computation time based on the belief a larger or smaller savings wouldn't enhance the evaluation of the heuristics against each other.

Experiment Results

We evaluated the three proposed heuristics to determine if one regularly required fewer processors than the others as measured by the metric Percent Extra Processors. Greatest Benefit fulfills this goal; it consistently required fewer extra processors than the other heuristic algorithms. Figure 5 below presents the results from the first three experiments. Results from the second three experiments were similar and therefore not included.

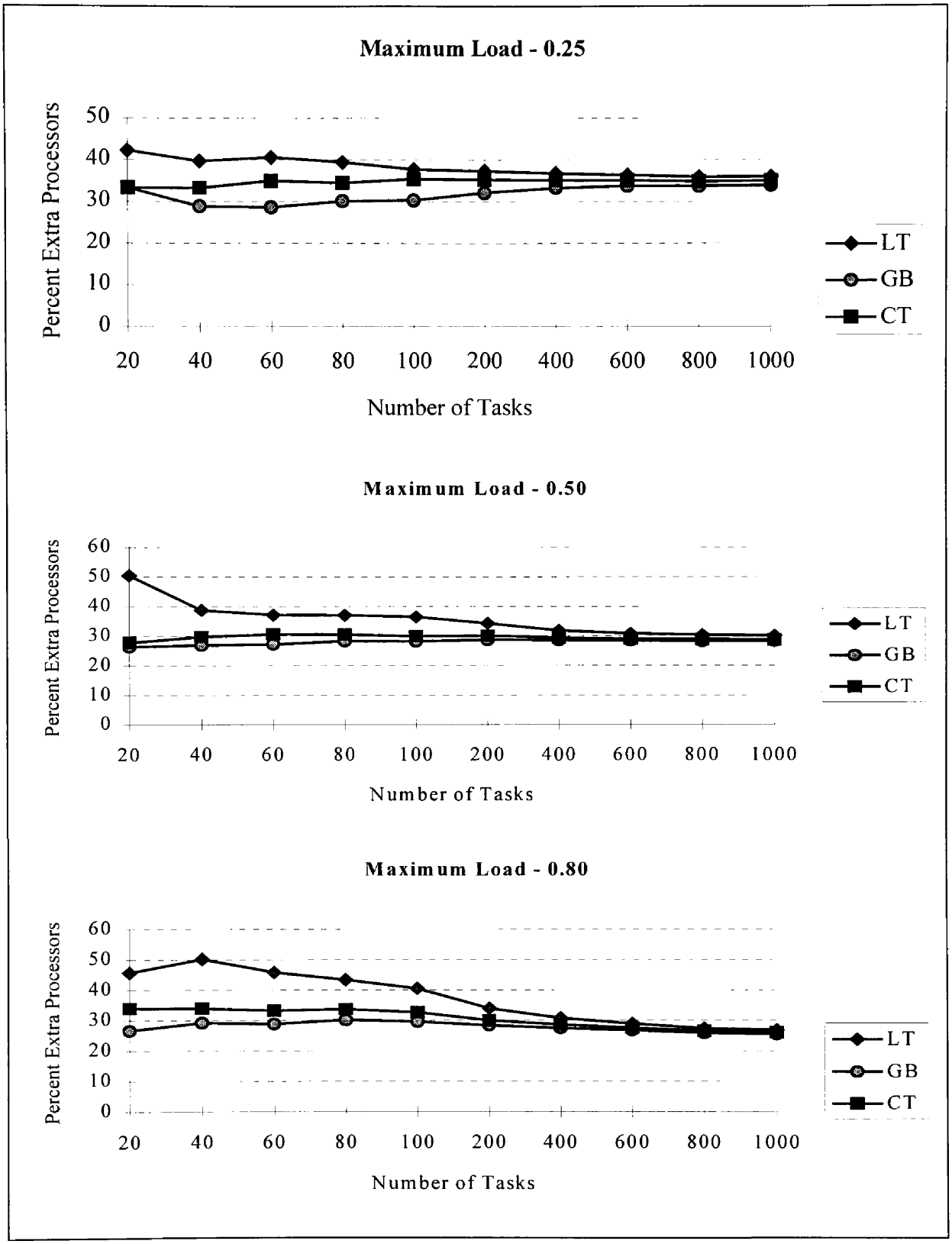


Figure 5 - Percent Extra Processors

The other goals of this study were to investigate how the heuristics react with increasing task loads and increasing number of special processors. In both cases, no correlation was found other than an increase in the total processor requirement when the task set utilization's increase, or the number of special processors decrease. As a matter of fact, both have the same affect on the heuristics. Simply put, decreasing the number of special processors has the same result as increasing the task set utilization.

This set of experiments examined the average case behavior of the proposed heuristic algorithms as measured by an approximation to an optimal value. The next series of tests analyze the performance of the task sets when measured against optimal solutions.

4.4 Comparison Against Optimal Solutions

Our second study characterizes how the heuristic perform when compared against an optimally scheduled task set. The optimal solutions were obtained by exhausting all possible solutions. Following the same philosophy as the first study, the second study contains nine separate experiments with each experiment containing five runs with an increasing number of tasks in each run.

Experiments

Every experiment contained a sample of 20 randomly generated task sets with the distribution varying depending on the test performed. Optimality testing was performed

by generating a random task set, scheduling it with each of the three heuristics, then generating and scheduling all possible permutations of the task set to determine the order creating the fewest number of processors. Unfortunately, due to the enormous number of possible distinct orderings of sets with a large number of tasks, time constraints limit us to allowing sets with 10 or less tasks. In the same respect, the time constraints also limit the number of sample runs available for testing. Tasks were generated in the same manner described in the previous section. Table VIII defines the experimentation parameters.

Experiments 1, 2, and 3 show how an increasing load with uniform distribution affects the optimality of the results. Similarly, experiments 4 - 6 and 7 - 9 show the response of the allocation heuristics to increasing loads with normal and exponential distributions.

In all experiments, the goal is to see how close to optimal the algorithms are under specific test conditions. With such a small number of tasks available, the only way we shall see differences in the output is to allow only one special processor and generate high load task sets by lowering the maximum period from 500 to 250.

	Min. Period	Max. Period	Sample Size	Special Processors	Distribution	Load Ratio
1	1	250	20	1	Uniform	**0.2**
2	1	250	20	1	Uniform	**0.5**
3	1	250	20	1	Uniform	**0.8**
4	1	250	20	1	Normal	**0.2**
5	1	250	20	1	Normal	**0.5**
6	1	250	20	1	Normal	**0.8**
7	1	250	20	1	Exponential	**0.2**
8	1	250	20	1	Exponential	**0.5**
9	1	250	20	1	Exponential	**0.8**
Number of tasks: {2,4,6,8,10}						

Table VIII Optimal Evaluation Parameters

Experiment Results

For this evaluation we need to define a new performance measure, the Optimality Ratio (OR). Specifically, this metric is the ratio of the number of processors required for the heuristic to the number of processors required for an optimal assignment. The number of processors required by the heuristic is labeled $N(\Im)$ and the number of processors required for an optimal assignment is labeled $N(O)$.

The Optimality Ratio is computed as: $\text{OR} = \frac{N(\Im)}{N(O)}$

Table IX Optimality Ratios, shows the results from the study. An entry in the table indicates the percentage of task sets meeting the Optimality Ratio. For example, an entry of 75% in the column under OR ≤ 1.4 indicates 75% of task schedules generated for that run require less than or equal to 1.4 times the number of processors required for optimal

assignments. The average processors column provides an indication of how many processors were required for the task set. It is the average of all samples rounded up to the next integer.

Largest Number Tasks							
Tasks	OR = 1.0	≤ 1.2	≤ 1.4	≤ 1.6	≤ 1.8	≤ 2.0	Avg. Procs.
2	100%	100%	100%	100%	100%	100%	1
4	75%	75%	75%	100%	100%	100%	3
6	60%	80%	80%	100%	100%	100%	3
8	40%	45%	95%	100%	100%	100%	5
10	25%	60%	100%	100%	100%	100%	6
Greatest Benefit							
Tasks	OR = 1.0	≤ 1.2	≤ 1.4	≤ 1.6	≤ 1.8	≤ 2.0	Avg. Procs.
2	100%	100%	100%	100%	100%	100%	1
4	95%	95%	95%	100%	100%	100%	2
6	100%	100%	100%	100%	100%	100%	3
8	65%	70%	100%	100%	100%	100%	5
10	60%	75%	100%	100%	100%	100%	6
Largest Computation Time							
Tasks	OR = 1.0	≤ 1.2	≤ 1.4	≤ 1.6	≤ 1.8	≤ 2.0	Avg. Procs.
2	100%	100%	100%	100%	100%	100%	1
4	85%	85%	85%	100%	100%	100%	2
6	80%	80%	90%	100%	100%	100%	3
8	65%	70%	100%	100%	100%	100%	5
10	55%	70%	100%	100%	100%	100%	6

Table IX Optimality Ratios

From this table we can see all three heuristics produce task allocations within 1.4 of the optimal. Furthermore, GB again showed the best results in the highest task number row by generating optimal schedules 60% of the time. These findings correspond with the data from the average case behavior study.

Only the results from experiment three are presented. Results from the other high load experiments (6 and 9) are similar to those shown. The results from the low and medium utilization experiments provided no significant data. Utilization rates were so low the schedules often required only one or two processors, and that makes it "easy" to generate an optimal schedule. Likewise, testing with different data distributions yielded no meaningful results.

4.5 Summary

In this chapter we relaxed the restriction requiring similar processors in the multiprocessor periodic task scheduling problem. We introduced three heuristic allocation strategies to solve the extended problem. Through average case and optimality testing, we show the Greatest Benefit heuristic algorithm consistently provides the best results. GB provides optimal schedules 60% of the time for task sets containing 10 tasks and no worse than 1.4 times optimal in general. Furthermore, since GB is founded on the Increasing Period schedulability criteria, higher utilization rates and improved optimality results may be possible by choosing a more advanced multiprocessor allocation heuristic.

5 Scheduling with Communication

In the previous chapter we relaxed the restriction requiring homogeneous processors in a multiprocessor environment and provided three allocation strategies for solving the new problem. In this chapter, we expand the original problem definition introduced by Liu and Layland in [1] by allowing communication among tasks. Distributed system communication is addressed by Sha and Sathaye in [22] and [23]. They provide a framework, but no allocation algorithm, for assigning tasks with communication requirements. Essentially, they create real-time aperiodic tasks to handle communication across networks and adjust task priorities for communication across busses. For this thesis, we consider communication in the "classical" scheduling sense with the added restriction of periodic tasks and deadlines.

5.1 System Model

In order to work with communication, we must re-examine our system model. Previously, since communication was not considered, no multiprocessor interconnection topology was specified. Here, the processors are considered fully connected and tasks can execute on any processor. As in the previous chapter, we allow an unlimited number of processors.

The modified task set model is $\Im = \{\tau_i = (C_i, T_i, (K_i, \tau_j)) \mid i = 1, ..., n\}$ of n tasks. K_i represents the time required to perform communication between tasks τ_i and τ_j. We also assume the following:

1. A task communicates by sending a message to <u>one</u> other task
2. A task communicates in one direction only; $\tau_i \rightarrow \tau_j$ (not $\tau_i \leftrightarrow \tau_j$)
3. A task can only send data to one other task and receive data from one other task
4. Communicating tasks have the same period; $T_i = T_j$
5. Communication cost is less than the computation; $K_i \leq C_i$
6. If the two tasks execute on the same processor, $K_i = 0$
7. Task communication precedence can be represented as a graph:
    a. No cycles
    b. Zero or 1 task points in
    c. Zero or 1 task points out

In addition to task periods, message costs, and precedence, we also must consider message deadlines. Because a task must be ready to run at the start of its period and can execute any time throughout its period, we allow one full period for the transfer of the message. This will allow the data to arrive at the other processor before the receiving task is ready to execute. Consider this example provided with Figure 6:

$\tau_1 = (3, 10, (3, \tau_2))$ - τ_1 executes on ρ_1 for 3 time units anywhere between time 0 and time 10; completing by time 10. Additionally, it must send a message of 3 units length to τ_2.

$\tau_2 = (2, 10, (0, 0))$ - τ_2 executes on ρ_2 for 2 time units anywhere between time 0 and time 10; completing by time 10.

For our example, τ_1 will run from time 7 to 10 (assume higher priority tasks use the time between 0 and 7). Before it completes processing, it sends a message to τ_2 on ρ_2. The message will take 3 time units to transfer.

τ_2 executes periodically from time 0 - 2, 10 - 12, 20 - 22, ...

Since τ_1 executes at the end of its period and τ_2 at the beginning, there is no way a message of 3 units can be transferred from ρ_1 to ρ_2 before τ_2 starts its 2nd instance of execution.

After a two period startup time, τ_1 will be 1 instance ahead of τ_2, and τ_2 will process the output of τ_1 while τ_1 is preparing the next set of data.

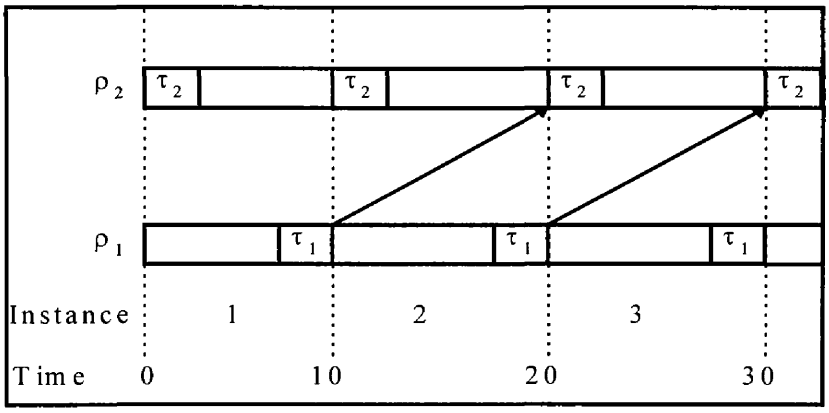


Figure 6 - Communication Example

Though the requirements appear rather strict, they do allow the addition of communication into the periodic task scheduling problem.

Before looking at any allocation strategies let's discuss performance goals. First and foremost, all deadlines must be met. After that, we seek to minimize the total number of processors and minimize the total amount of communication between all tasks.

Achieving these goals is possible through the incorporation of existing rate-monotonic schedulability functions and new allocation schemes.

5.2 Heuristics

We introduce two heuristics for incorporating real-time communication while guaranteeing task deadlines. The first heuristic is not new, just a different view of an existing multiprocessor scheduling algorithm. The second heuristic is designed to minimize the amount of communication while attempting to achieve high utilization rates on the processor.

For the first heuristic we shall use the RMFFS algorithm from Chapter 2. The use of this algorithm is possible by the characteristics of the task set preparation policy--sorting by increasing period values. Since two tasks can communicate only if they have the same periods, the increasing period sort should provide some lowering of intertask communication.

Where the RMFFS should naturally lower the communication costs, the second heuristic is deliberately designed to lower data transfer costs. Heuristic Minimize Communication (MC) seeks to decrease the amount of communication by scheduling all communicating *chains* on the same processor. In addition, we make use of a special property of the RM

schedulability criteria that allows utilization rates of up to 100% when tasks have the same periods.

A sequence of communicating tasks is referred to as a chain. By placing the chains on the same processor in precedence order, lower data transfer costs are achievable. Additionally, because all the tasks on a processor have the same periods, 100% utilization is attainable while still guaranteeing deadlines [12].

Before presenting an evaluation of the performance of the proposed heuristics, Algorithm MC is described below.

Minimize Communication

```
-- Schedule the special processors first
Partition the tasks into G sets with each set containing a
different communication chain in precedence order.

Set H will contain all the remaining tasks.

Order the G sets from largest number of tasks in each set to
smallest.

j = 1

FOR  each set F in G
       i = 0
       WHILE | F | > 0 LOOP

             u = Σ of all u_i on ρ_j
             WHILE (u + u_i ≤ 1.0) LOOP
                   F = F - τ_i
                   add τ_i to ρ_j
                   i = i + 1
             END LOOP
             j = j + 1

             IF | F |   = 1 THEN
                   H = H ∪ F
                   F = ∅
             END IF

       END WHILE

END FOR

-- Schedule the remainder of the tasks
IF  | H | > 0 THEN
      use RMFFS to schedule H on standard processors
END IF
```

Both allocation heuristics for solving the communication problem have linear time complexity as shown in the table below.

Algorithm	Time Complexity
RMFFS	$O(n \log n)$
MC	$O(n \log n)$

Table X Complexity of Communication Algorithms

(n = number of tasks)

Similar to the experimentation performed in the previous chapter, this study also makes use of average case behavior and optimality evaluation.

5.3 Performance Evaluation of the Heuristics

For the average case study, evaluation consists of testing the algorithms against large numbers of randomly generated task sets based on different variables. The first three experiments look to determine the effect of increasing numbers of communicating tasks on the communication requirements. The second three experiments examine how the heuristics react to increasing communication loads. In the last three experiments, both the amount of data transferred and the number of communicating tasks increases with each experiment.

Experimentation Parameters

Other than Dispersion, Average C/E, and Links Ratio, the test parameters for this study remain as defined in previous chapters. New parameters are in order to evaluate the characteristics of the communication heuristics. The input parameter Average C/E

reflects the amount of data transferred from one task to another as a ratio of the execution time. More precisely, define Avg. C/E as:

$$\text{Average C / E} = \frac{\text{Average Communication Time}}{\text{Average Computation Time}}$$

Given a task computation time, a selected Avg. C/E, and a dispersion range, one can randomly generate communication costs under very tight control. Therefore, given the Avg. C/E and Dispersion parameters, the amount of data transferred from one task to another is generated as follows:

1. Target_Comm = C_i * Avg. C/E
2. Hi_Range = Target_Comm + (Dispersion * Target_Comm)
3. Lo_Range = Target_Comm - (Dispersion * Target_Comm)
4. Generate random number between Hi_Range and Lo_Range (Lo_Range $\leq K_i \leq$ Hi_Range)

Related to, but different from the Avg. C/E ratio, the Links Ratio defines a target number of communicating tasks. Supplying a high Links Ratio will create a task set with many communicating tasks. Conversely, a low Links Ratio means few tasks communicate. We define the Links Ratio as: $\frac{\text{Number of Links}}{\text{Number of Tasks}}$. In the same manner as the amount of communication, the number of links in a task set is randomly generated within the high and low dispersion values of a target number of links. Table XI defines the evaluation parameters for studying the average case behavior of communication aware periodic task algorithms.

Test	Min. Period	Max. Period	Load Ratio	Sample Size	Distribution	Dispersion	Avg. C/E	Links Ratio
1	1	500	0.5	50	Uniform	± 10 %	0.5	**0.2**
2	1	500	0.5	50	Uniform	± 10 %	0.5	**0.4**
3	1	500	0.5	50	Uniform	± 10 %	0.5	**0.6**
4	1	500	0.5	50	Uniform	± 10 %	**0.3**	0.4
5	1	500	0.5	50	Uniform	± 10 %	**0.6**	0.4
6	1	500	0.5	50	Uniform	± 10 %	**0.9**	0.4
7	1	500	0.5	50	Uniform	± 10 %	**0.3**	**0.2**
8	1	500	0.5	50	Uniform	± 10 %	**0.6**	**0.4**
9	1	500	0.5	50	Uniform	± 10 %	**0.9**	**0.6**
Number of Tasks: {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}								

Table XI Communication Algorithms Testing Parameters

Experiment Results

We evaluated both heuristics against large samples of data with increasing levels of communication. In all experiments, the results of the MC heuristic are superior to that of RMFFS. However, the result shouldn't be too surprising. MC was developed to achieve lower communication requirements whereas RMFFS superficially looks to lower data transfer amounts. The results were measured by two statistics: Average Communication, Average Processors. These two metrics are simply the means of the amount of communication, and number of processors required for the run.

The results from the experiments 7, 8, and 9 are shown below in Figure 7 and Figure 8. Results from the first six experiments are basically the same and are not shows in this thesis.

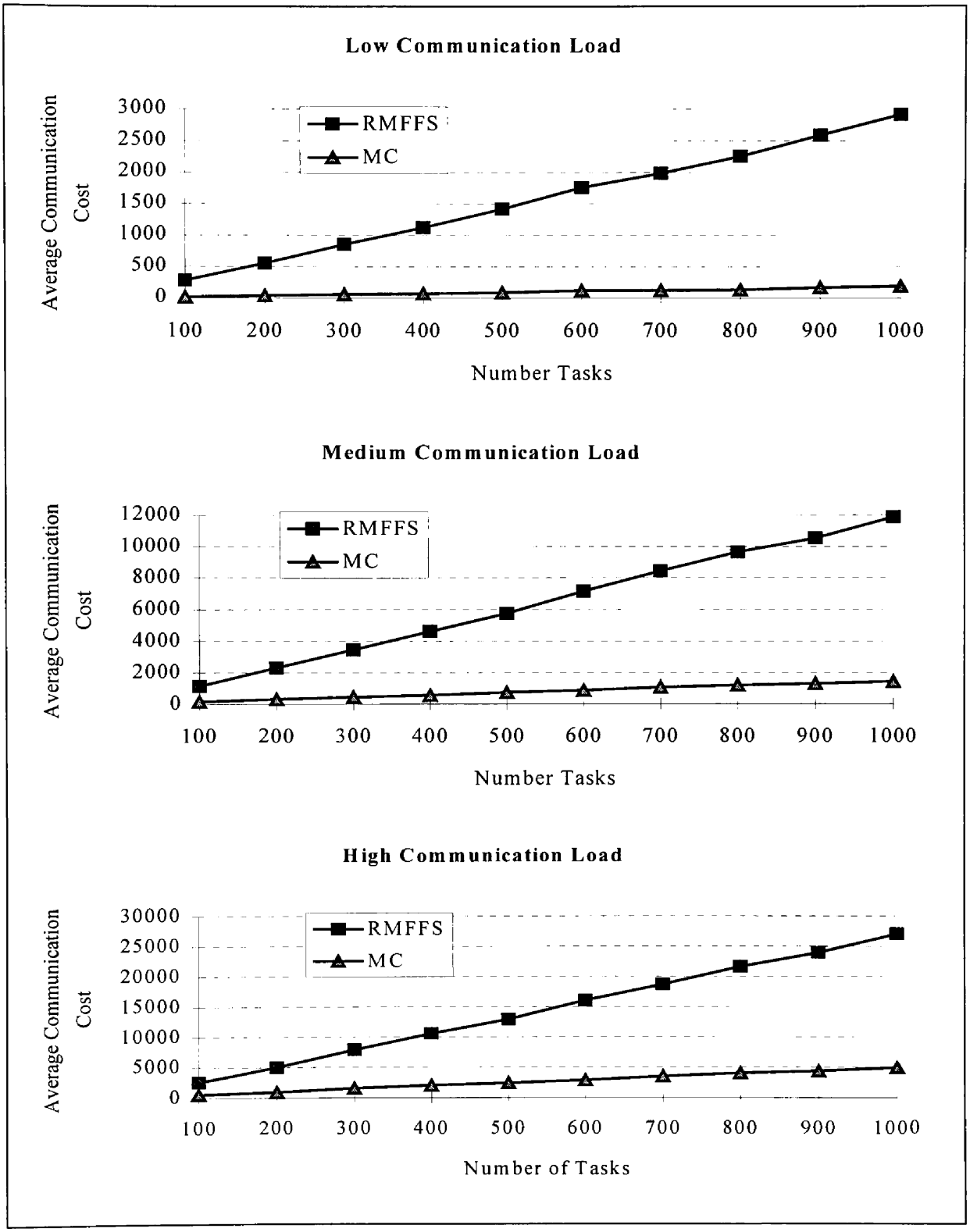


Figure 7 - Average Communication Cost

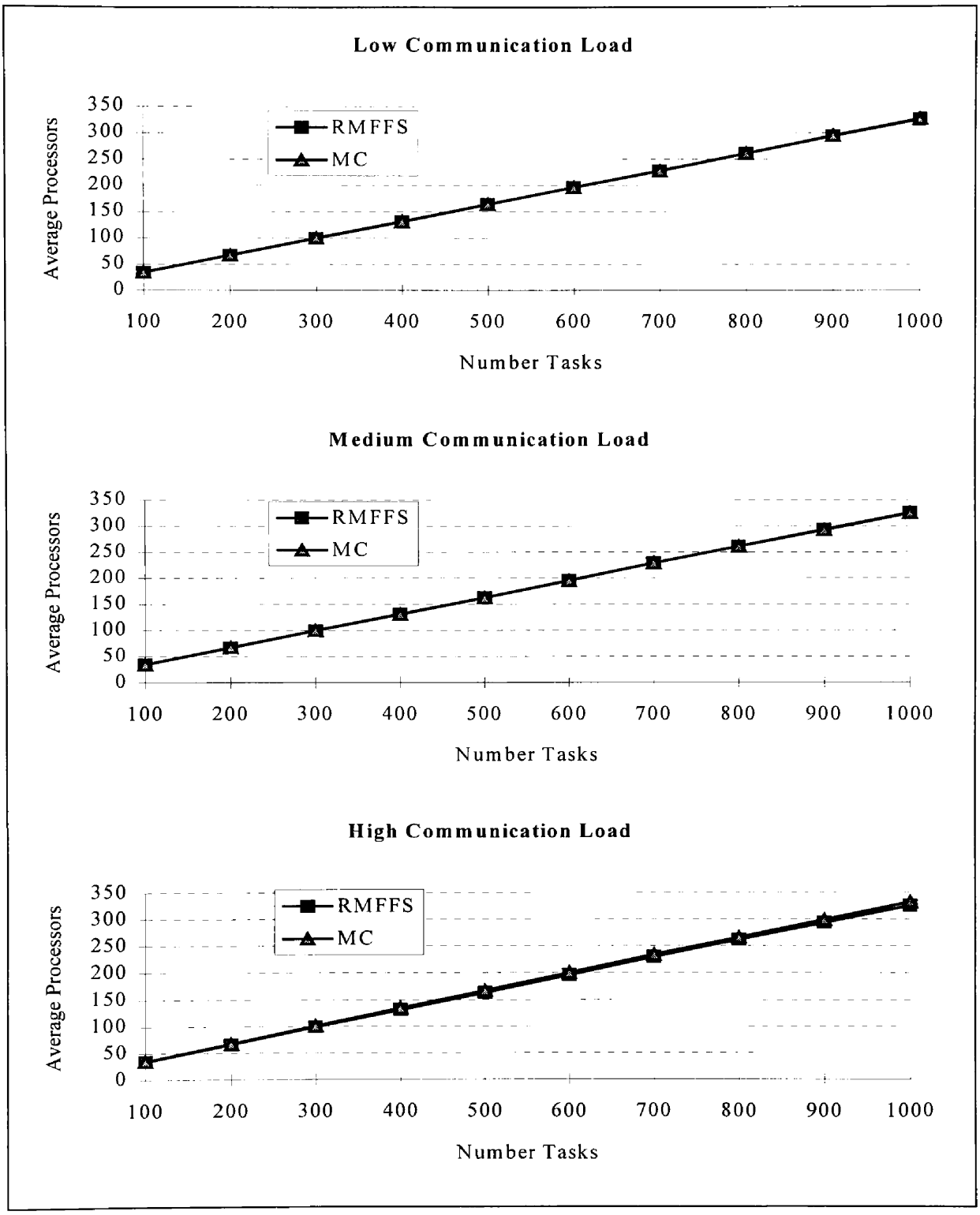


Figure 8 - Average Number of Processors

In all experiments performed, the Average Communication requirements for the MC heuristic were significantly lower than RMFFS (Figure 7). Moreover, the increase in communication costs for the RMFFS are almost linearly related to an increase in the number of tasks. MC, on the other hand, showed low sloping curves as the number of tasks increased to 1000 (see Figure 7 - Average Communication Cost). When studying the effect of increasing communication costs, both algorithms produced the same responses for all levels of increasing communication. Aside from increased Average Communication values, all graphs showed the same basic structure.

Another interesting observation from this series of experiments is the Average Processors statistic. We incorporated the function allowing up to 100% processor utilization for tasks with the same periods in attempt to minimize the total number of processors required. This function may not have come into play. Looking at Figure 8, the average number of processors required for the high communication load experiment was slightly more than RMFFS. We correlate this difference to high Links Ratio values. In the lower communication load experiments, MC mirrored the processor requirements of RMFFS. Though the increase is very small it may indicate a problem with the implementation of MC. This behavior is due to the nature of MC which doesn't allow scheduling of the Set *H* tasks on the previously allocated processors, even though idle time may be available. Another possibility is the selection of RMFFS for scheduling the Set *H* tasks. As shown in Chapter 3, RMFFS performed poorly against the other heuristics.

We have shown that the MC allocation strategy provides great communication savings compared to RMFFS while in most cases requiring the same number of processors. But, how does MC stand up to optimality testing?

5.4 Comparison Against Optimal Solutions

In the previous study, we were determining how the heuristics react to increasing communication costs, and looking to see if one heuristic outperforms the other. MC provides superior communication savings while on average utilizing the same number of processors as RMFFS. Now, we seek to evaluate the performance of the algorithms in relation to optimal solutions.

Experiment Parameters

In the same manner as the study of average case performance, optimality testing employs similar evaluation parameters. The main difference is in the sample size and number of tasks in each experiment. Time constraints limit us to small samples with 10 or less tasks. Under these constraints, we need a method of adequately testing the performance of the heuristics. To do this, we created high utilization task sets in attempt to require as many processors as possible for each run. More processors provides more opportunities to place tasks on different processors. Table XII shows the parameters for the optimality study.

	Min. Period	Max. Period	Load Ratio	Sample Size	Distribution	Dispersion	Avg. C/E	Links Ratio
1	1	50	0.9	20	Uniform	± 10 %	0.5	**0.2**
2	1	50	0.9	20	Uniform	± 10 %	0.5	**0.4**
3	1	50	0.9	20	Uniform	± 10 %	0.5	**0.6**
4	1	50	0.9	20	Uniform	± 10 %	**0.3**	0.4
5	1	50	0.9	20	Uniform	± 10 %	**0.6**	0.4
6	1	50	0.9	20	Uniform	± 10 %	**0.9**	0.4
Number of Tasks: {2, 4, 6, 8, 10}								

Table XII Optimality Experimentation Parameters

Under the above input parameters, we evaluated the periodic task communication heuristics to see how close to optimal their allocations are. Similar to the performance evaluation study, one goal of this study is to determine the effect of increasing communication costs.

Experiment Results

Reflecting the results from evaluating the heuristics against large task sets, optimality testing also shows the benefit of MC when compared to RMFFS. From Table XIII below, the larger numbers under the "OR=1.0" column indicate MC provides more optimal solutions than RMFFS. Additionally, when compared to MC, RMFFS produces many more task schedules requiring greater than twice the amount communication than optimal schedules.

Rate-Monotonic-First-Fit-Scheduling							
Tasks	OR = 1.0	≤ 1.2	≤ 1.4	≤ 1.6	≤ 1.8	≤ 2.0	> 2.0
2	100%	100%	100%	100%	100%	100%	0%
4	50%	60%	60%	70%	70%	75%	25%
6	25%	25%	30%	35%	35%	45%	55%
8	15%	40%	55%	75%	75%	75%	25%
10	20%	30%	55%	65%	70%	85%	15%
Minimize Communication							
Tasks	OR = 1.0	≤ 1.2	≤ 1.4	≤ 1.6	≤ 1.8	≤ 2.0	> 2.0
2	100%	100%	100%	100%	100%	100%	0%
4	95%	95%	95%	95%	95%	95%	5%
6	80%	80%	85%	85%	85%	90%	10%
8	65%	75%	80%	80%	80%	90%	10%
10	90%	100%	100%	100%	100%	100%	0%

Table XIII Optimality Testing Results

Data in the table above is from experiment 3 requiring a high number of communication links. Even with this heavy communication load, we can generalize and say MC often produces optimal or close to optimal schedules. This is the worst case. In the lower communication load experiments, many more optimal schedules were found and the non-optimal schedules were closer to optimal than in the table above. Since experiment 3 shows the poorest performance from all experimentation, the other results are not included here.

Before closing this chapter, an explanation of optimality in the sense of multiprocessor periodic tasks and communication is in order. Since the primary objective of this study was to measure the performance of the proposed heuristics in relation to optimal, total communication cost of a schedule determined the optimal solution; not the number of

processors. Optimal communication was found by generating all possible permutations of the task set one at a time, applying the EX multiprocessor criteria with no sort, and measuring the amount of communication. The EX criteria was selected because it requires the fewest processors of all the multiprocessor heuristics. Why is this important? Because MC may use extra processors to lower communication while our "optimality" algorithm always attempts to minimize the number of processors before determining communication. Therefore, at the expense of an extra processor, MC can order the task set such that it requires less communication than EX. During testing, if MC required *less* communication than EX, MC was determined to have produced an optimal schedule.

5.5 Summary

We began this chapter by defining a new system model for incorporating communication into the rate-monotonic scheduling problem. The primary motivation behind the proposed algorithms is to achieve as low communication loads as possible while still guaranteeing task deadlines. Through average case analysis and optimality testing, we determined the proposed heuristic Minimize Communication provides exceptional savings in communication cost while utilizing the same number, or slightly more, processors than RMFFS.

6 Rate-Monotonic Scheduling and Analysis Tool

In this chapter we present the Rate-Monotonic Scheduling and Analysis Tool (RM-SAT). RM-SAT will support researchers in the development of new scheduling algorithms and allow system designers to evaluate their applications against several up-to-date heuristics.

6.1 Benefits

Since heuristics may perform differently for different sets of tasks, the tool is important for designers/developers. Using the tool, they can see how the different heuristics may schedule a given task set. Thus, the best schedule for this particular set of tasks can be adopted or the entire design may be tuned. The tool can be helpful in answering "what if" questions. For a given task set, will different heuristics meet certain performance goals while others won't?

Currently, when researchers are developing new periodic task scheduling algorithms they produce a new heuristic, provide a theoretical evaluation of the performance bounds, then, perform an average case analysis against the prevailing set of heuristics existing in the literature. This tool will aid in the average case analysis experimentation.

Another benefit of this tool is the quick summary of results available at the end of every test run. The results are available as either a table or histogram depending on the target system and type of testing. Providing bar charts soon after testing lets users know the

results right away without having to input data into a spreadsheet and create a graph. Of course, statistics from the test run are savable to a file for later insertion into a spreadsheet. With these benefits in mind, we can now cover the features contributing to the usefulness of the tool.

6.2 System Overview

The basic features of RM-SAT are categorized by where they reside in the scheduling and testing process. We take a three phase approach: initialization, testing, and reviewing statistics. During the initialization phase, the user specifies all of the target system, task set, and algorithm parameters. Several options are available. First, either Single Processor, Multiprocessor, or Special Multiprocessor, is selected from the target system menu. Included with the Special Multiprocessor selection is the option of specifying how many are available to the allocation algorithm. The next menu determines where the test data will come from. Possible choices include generating a random task set, reading the task data from a file, or entering the tasks manually. If the test data comes from randomly generated task sets, a menu will be displayed providing several options to allow tight control of testing parameters. Included within these options are: random number distribution, number of tasks, minimum and maximum period values, task load ratio, and number of task sets. See Figure 9 - Random Task Set Options.

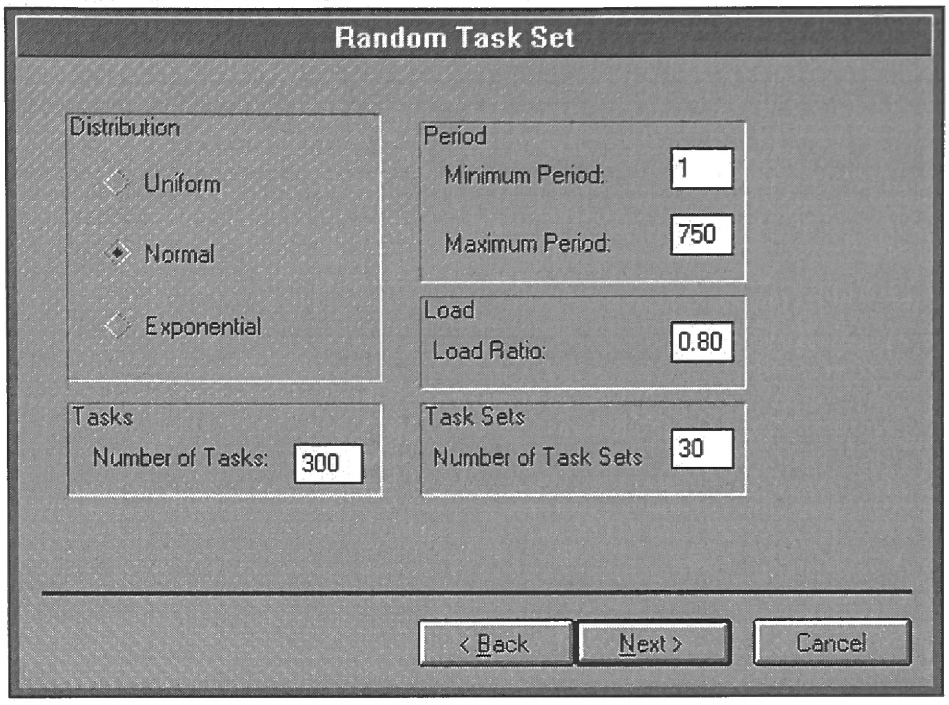


Figure 9 - Random Task Set Options

The last selected option during setup is choosing which algorithm(s) to test. Algorithm options depend on the target system chosen from the initial menu. Figure 10 displays the possible choices for single processor testing.

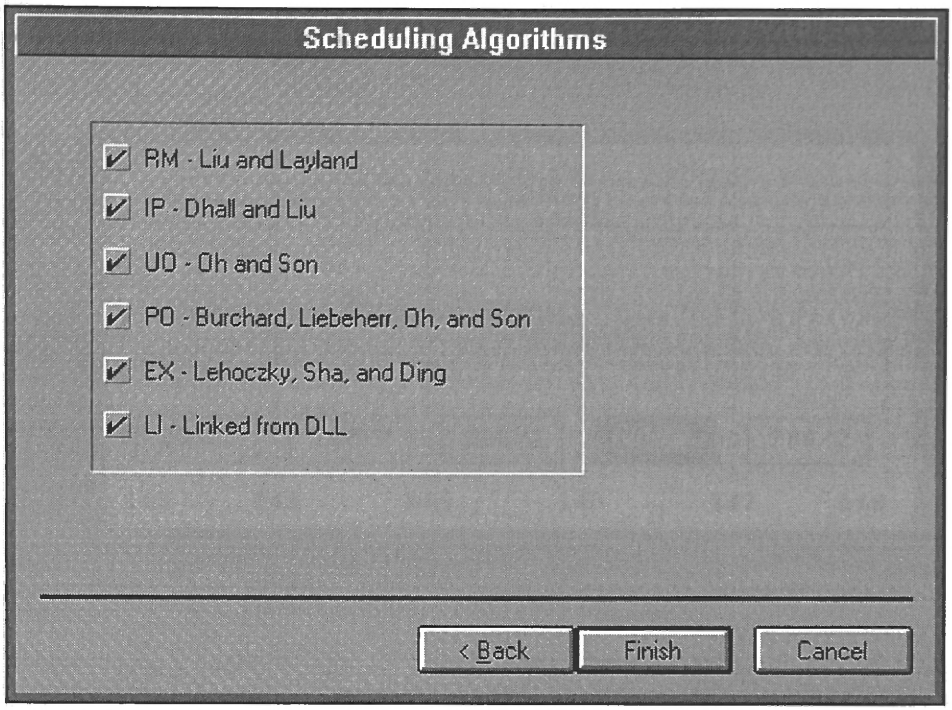


Figure 10 - Single Processor Scheduling Algorithms

When all required options are selected, the main window is redisplayed with the "Test" button enabled for use. At this point, the task set can be saved to disk by selecting "File | Save" from the top menu. Testing is started by selecting the "Test" button. An option menu appears allowing the selection of either Run testing or Step testing. In addition, the check box specifying task deadline verification is allowed for Run testing, whereas Step testing integrates deadline verification into the stepping process. Run testing provides no details of the target system during deadline verification. It's purpose is to allow task set generation and deadline testing for large samples of data. When testing only one task set, stepping through the test provides much more information as to the state of the target system. See Figure 11 - Single Processor Step Display.

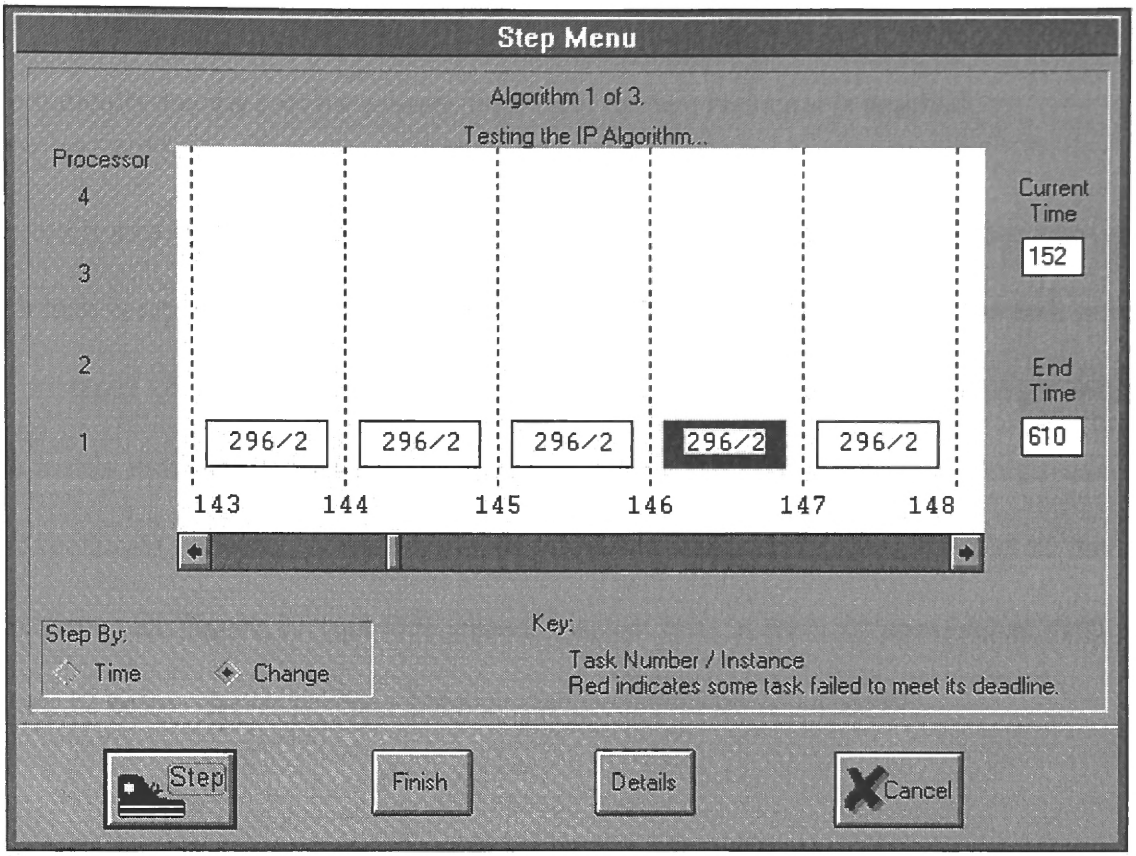


Figure 11 - Single Processor Step Display

Stepping options include specifying the step increment, either by one time unit or a change in processor allocation, or viewing the processor details. By pressing the "Details" button, the ready queue and waiting queue are shown for each processor. The ready queue holds tasks available for execution. Tasks that have completed execution for their current period, and are awaiting the next instance of invocation are placed in the waiting queue until the start of the next period. Pressing the "Step" button causes the system time to be incremented by one and the next set of tasks allocated to processors.

The "Finish" button prevents the tool from waiting for the "Step" button and automatically increments the system time until the test end time is reached.

The main area of the window displays the system time and current processor allocation. Each task is represented by a rectangle. Displayed within the rectangle is the task number and instance of invocation (how many time the task was started). A rectangle with no data inside indicates no tasks were available during the time increment. Additionally, a dark rectangle indicates a task on that processor missed a deadline at the end of the time increment. In Figure 11 task 296 caused another task to miss its deadline at time 147. Figure 12 below shows the Step Display for a multiprocessor schedule

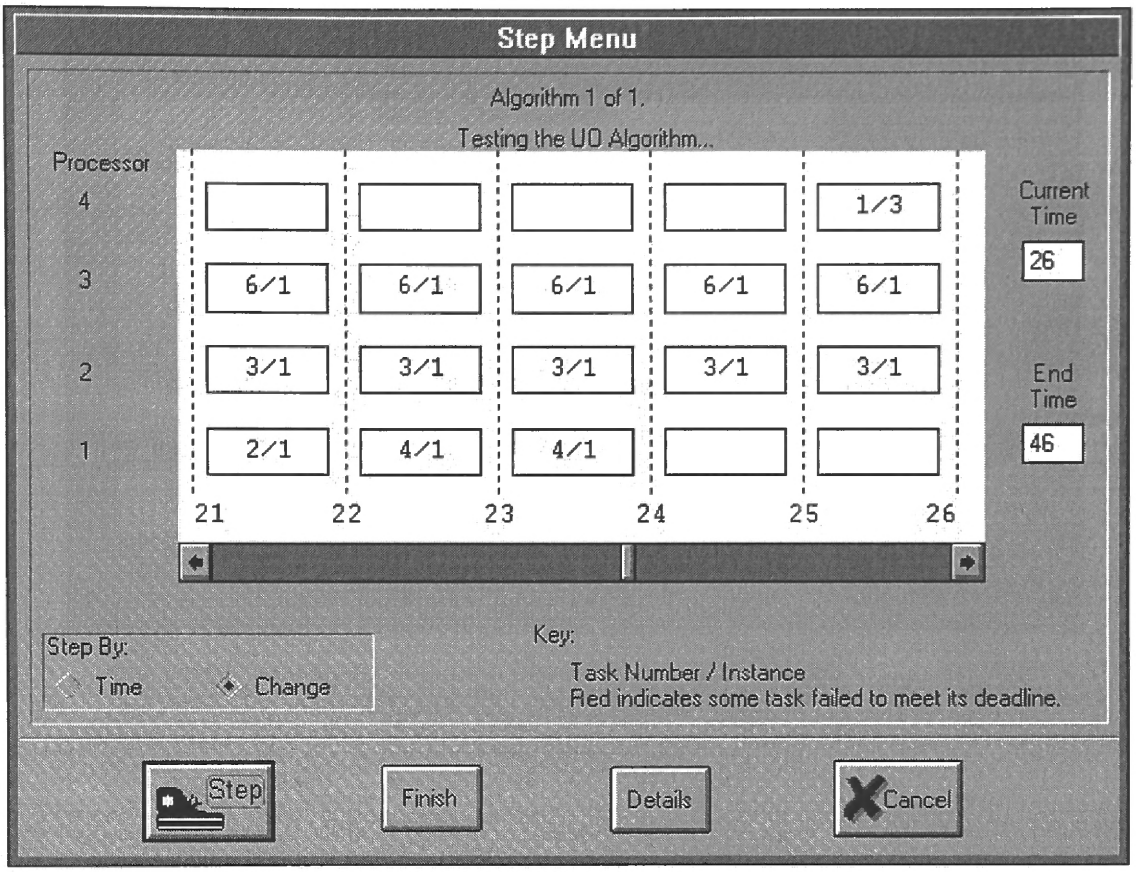


Figure 12 - Multiprocessor Step Display

Statistics are available after the completion of Run or Step testing. Presentation of results is target system and testing dependent with two bar charts available for most types of testing. Multiprocessor testing histograms show Percentage Extra Processors and Average Processor Utilization. Figure 13 provides an example display.

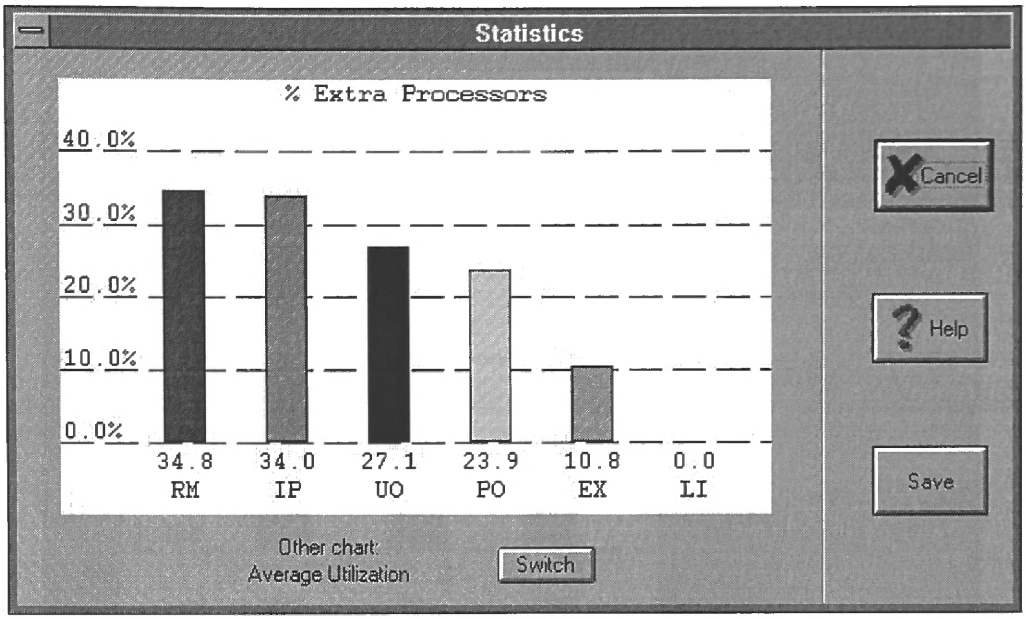


Figure 13 - Statistics Display

Statistics are also available for Heterogeneous Processor experimentation (see Figure 14). These bar charts show Percent Extra Processors and Percent Saved Utilization. While histograms provide a quick summary of the test results, the detailed statistics can be saved to a disk file. More advanced graphs are possible through the built-in capability to keep appending statistics to the same file and the using a spreadsheet to draw the graph.

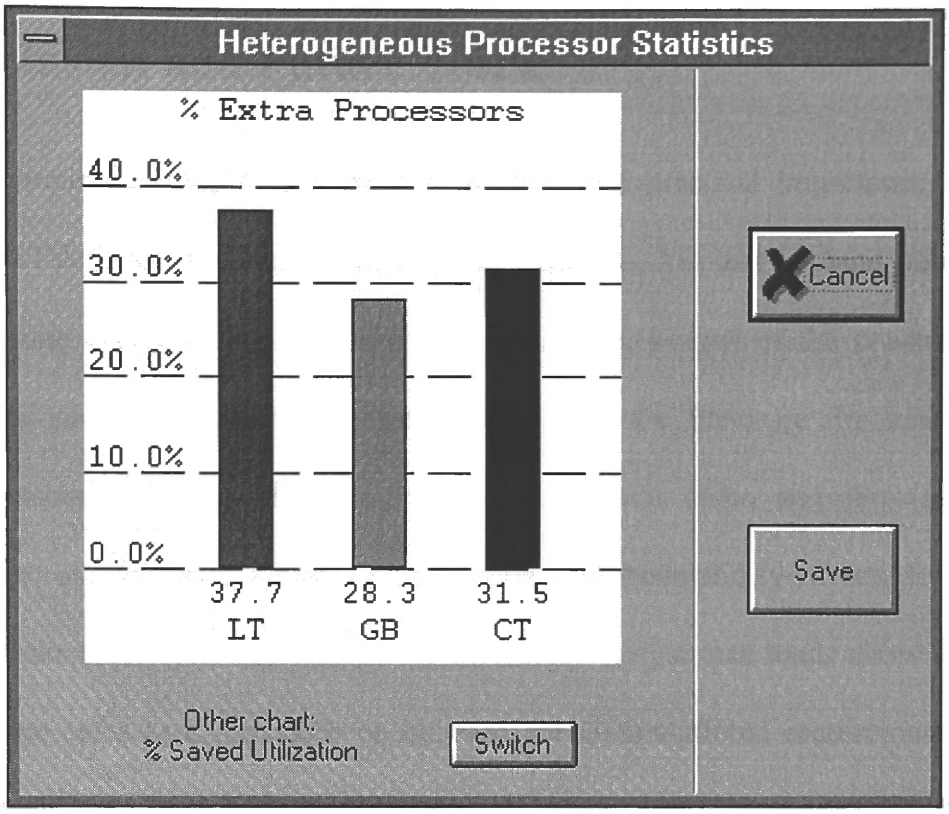


Figure 14 - Heterogeneous Processor Statistics

6.3 Summary

In this chapter we introduced the Rate-Monotonic Scheduling and Analysis Tool. RM-SAT will benefit both the research community and real-time system developers thorough it's ease of use and up-to-date heuristics. The capability to run "what if" scenarios, quickly display test results, and link in proposed algorithms make this tool invaluable. As the testbed for the experimentation performed in Chapters 3 and 4, RM-SAT proves its usefulness.

7 Conclusion and Future Work

Rate-Monotonic scheduling is of both theoretical and practical importance. In 1973 under a well defined set of restrictions and assumptions, Liu and Layland published the paper defining rate-monotonic scheduling. Over time, several of the constraints were relaxed and proposed solutions started appearing in the literature for scheduling on multiprocessors, adding aperiodic tasks with the periodic tasks, and allowing tasks to synchronize. Additionally, researchers enhanced the schedulability criteria defining how large the tasks loads were allowed on the processor. Larger task loads allow developers to add more tasks to a processor or increase the computational requirements without missing deadlines. Along with its theoretical success, rate-monotonic scheduling has seen visibility in industry over the past few years spurred on by its incorporation into the Ada 95 programming language and the POSIX standard.

We add to both the theoretical and practical aspects of rate-monotonic scheduling. The study of current multiprocessor allocation algorithms provides an unbiased and well tested comparison of their performance for various task set parameters. Though not a linear algorithm with respect to the size of the task set, our EX Multiprocessor heuristic provides schedules requiring the fewest number of processors when compared to the algorithms in the literature. Of the linear algorithms, RMGT provides the best all-round performance against medium and low utilization task loads. High task loads require scheduling with RM-FFDU.

We relaxed a restriction set forth by the early pioneers in this field by allowing processors with different computational characteristics. Tasks can execute on either type of processor but will save some computation time by executing on one of the special processors. We present three bin-packing heuristics for allocating tasks to the special processors in addition to the standard processors. Given a fixed number of special processors, the algorithms seek to minimize the total number of processors required for the task set while still meeting task deadlines. By evaluating the algorithms with randomly generated task sets and measuring their performance against each other and the optimal solution, we show the heuristic Greatest Benefit consistently required the fewest number of processors for all test conditions.

Another enhancement provided by this thesis is a new way of viewing communication within the realm of periodic tasks. Our goal is to include the consideration of communication costs during the scheduling process. In addition to introducing an updated system model, though rather restrictive, we provide an algorithm that attempts to minimize the amount of communication in a task schedule and still guarantee deadlines. Heuristic Minimize Communication provides excellent communication savings when compared to an algorithm not taking communication into direct account but based on a similar principle.

Of a little more practical importance, we offer RM-SAT. The rate-monotonic Scheduling and Analysis Tool. RM-SAT provides researchers and developers the functions

necessary to develop and test periodic task scheduling algorithms. Its greatest benefit lies in the ability for analyzing algorithms against large samples of randomly generated task sets. The converse is also true. RM-SAT provides a tool developers can use to find an algorithm for their specific task set. Producing graphs quickly and the capability of linking in new algorithms make RM-SAT an ideal heuristic design tool.

Our thesis offers many results to the field of rate-monotonic scheduling. Much work remains. An examination of the current set of periodic scheduling algorithms against task sets representing actual applications would increase the understanding of the real-time environment and enhance the development of new algorithms. In the case of heterogeneous processor scheduling, would different allocation strategies provide enhanced performance?

With the transfer of large single processor systems to multiprocessor and distributed environments, the importance of communication in scheduling will only increase. Therefore, our communication model requires more research. Can the model be adapted to a more realistic scheduling problem? Furthermore, lowering the processor requirements of the MC allocation strategy would involve instituting a processor reuse scheme and the inclusion of a better multiprocessor scheduling heuristic for the non-communicating tasks.

RM-SAT can always be upgraded to add new features. RM-SAT should include advanced real-time constructs like synchronization and aperiodic tasks in its scheduling and testing functions. New algorithms should be incorporated. Porting to other operating systems. The possibilities are endless.

Scheduling real-time periodic tasks is a critical procedure because task deadlines must be met at all costs. As shown in several areas during this thesis, no one algorithm provides the best results for all task sets. For every algorithm that produces great results in one area, in another area or when measured by a different statistic, it doesn't. Always test the algorithms against many task sets and always test the task sets against many algorithms.

References:

1. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, Jan 1973.

2. S. Dhall and C. Liu, "On a Real-Time Scheduling Problem," *Operations Research*, Vol. 26, No. 1, Jan - Feb 1978.

3. S. Davari and S. Dhall, "On a Periodic Real-Time Task Allocation Problem." In *19th Hawaii International Conference on System Sciences*, 1986.

4. Y. Oh and S. Son, "Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems," *Journal of Real-Time Systems*, Vol. 9, No. 3., Sept 1995.

5. A. Burchard, J. Liebeherr, Y. Oh, and S. Son, "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems," *IEEE Transactions on Computers*, Vol. 44, No. 12, Dec 1995.

6. J. Strosnoder, J. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Transactions on Computers*, Vol. 44, No. 1, Jan 1995.

7. L. Sha, R. Rajkumar, and S. Sathaye, "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," *Proceedings of the IEEE*, Vol. 82, No. 1, Jan 1994.

8. ISO/IEC 8652:1995(E), Ada95 Reference Manual, Annex D, Real-Time Systems.

9. M. Klein, J. Lehoczky, and R. Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing," *Computer*, Vol. 27, No. 1, Jan 1994.

10. J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *Computer*, Vol. 28, No. 6, Jun 1995.

11. K. Ramaritham and J. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proceedings of the IEEE*, Vol. 82, No. 1, Jan 1994.

12. A. Tilborg and G. Koob, "Foundations of Real-Time Computing: Scheduling and Resource Management," Kulwer Academic Publishers, 1991.

13. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, "Numerical Recipes: The Art of Scientific Computing," Cambridge University Press, 1986.

14. T. Tia, J. Liu, and M. Shankar, "Algorithms and Optimality of Scheduling Soft Aperiodic Request in Fixed-Priority Preemptive Systems," to appear in *The Journal of Real-Time Systems*.

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

16. J. Silcock and S. Kutti, "Taxonomy of Real-Time Scheduling," Technical Report, School of Computing and Mathematics, Deakin University, Australia, 1995.

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

18. T. Baker, "Stack-based Scheduling of Real-Time Processes," *Journal of Real-Time Systems*, Vol. 3, No. 1, Mar 1991.

19. M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," Vol. 2, No. 4, Nov 1990.

20. J. Leung and J. Whitehead, "On the Complexity of Fixed-priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, Vol. 2, No. 4, Dec 1982.

21. K. G. Shin and Y.-C. Chang, "A Reservation Based Algorithm for Scheduling Both Periodic and Aperiodic Real-Time Tasks," *IEEE Transactions on Computers*, Vol. 44, No. 12, Dec 1995.

22. L. Sha and S. Sathaye, "Distributed Systems Design Using Generalized Rate Monotonic Theory," Technical Report, Software Engineering Institute, Carnegie Mellon University, Sept 1995.

23. S. Son, "Advances in Real-Time Systems," Prentice Hall, 1995.

24. Y. Oh, "The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems," Ph.D. Dissertation, School of Engineering and Applied Science, University of Virginia, May 1994.

25. S. Kochan, "Programming in ANSI C," Hayden Books, A Division of Prentice Hall Publishing, 1988.

26. T. Swan, "Tom Swan's C++ Primer," Sams Publishing, A Division of Prentice Hall Publishing, 1992.

27. C. Petzold and P. Yao, "Programming Windows 95," Microsoft Press, 1996.

28. J. Richter, "Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface, Microsoft Press, 1994.

29. J. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, Vol. 10, 1975.