1-1-2002

Value-based scheduling in real-time systems

Swaminathan Sivasubramanian
Iowa State University

Value-based scheduling in real-time systems

by

Swaminathan Sivasubramanian

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Manimaran Govindarasu, Major Professor
Arun K. Somani
Soma Chaudhuri

Iowa State University

Ames, Iowa

2002

Graduate College
Iowa State University

This is to certify that the Master's thesis of

Swaminathan Sivasubramanian

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

DEDICATION

I would like to dedicate this thesis to my brother Srinivas and my parents, without whom this Masters and my future Ph.D would have never been possible. I owe all my success in my life till now to them. In academic front, I owe a lot to my advisor Dr. Manimaran, who encouraged me to do research and pursue a Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A real-time system must execute functionally correct computations in a timely manner. Most of the current real-time systems are static in nature. However in recent years, the growing need for building complex real-time applications coupled with advancements in information technology, drives the need for dynamic real-time systems. Dynamic real-time systems need to be designed not only to deal with expected load scenarios, but also to handle overloads by allowing graceful degradation in system performance. Value-based scheduling is a means by which graceful degradation can be achieved by executing critical tasks that offer high values/benefits/rewards to the functioning of the system. This thesis identifies the following two issues in dynamic real-time scheduling: (i) maintaining high system reliability without affecting its schedulability and (ii) providing graceful degradation to the system during overload and maintaining high schedulability during underloads or near full loads. Further, we use value-based scheduling techniques to address these issues.

The first contribution of this thesis is a reliability-aware value-based scheduler capable of maintaining high system reliability and schedulability. We use a *performance index* (PI) based value function for scheduling, which can capture the tradeoff between schedulability and reliability. The proposed scheduler selects a suitable redundancy level for each task so as to increase the performance index of the system. We show through our simulation studies that proposed scheduler maintains a high system value (PI). The second contribution of this thesis is an adaptive value-based scheduler that can change its scheduling behavior from deadline-based scheduling to value-based scheduling based on the system workload, so that it can maintain a high system value with less deadline misses. Further, the scheduler is extended to heterogeneous computing (HC) systems, wherein the computing capabilities of processors/machines are different, and propose two adaptive schedulers (Basic and Integrated) for HC systems. The performance of the proposed scheduling algorithms is studied through extensive simulation studies for both homogeneous and heterogeneous computing systems. We have concluded that the proposed adaptive scheduling scheme maintains a high system value with less deadline misses for all range of workloads. Amongst the schedulers for HC systems, we conclude that the Basic scheduler, which has a lesser run-time complexity, performs better for most of the workloads. The last contribution of this thesis is the design and implementation of the proposed adaptive value-based scheduler for homogeneous computing systems in a real-time Linux operating system, RT-Linux. We compare the performance of the implementation with EDF and Highest Value-Density First (HVDF) schedulers for various ranges of workloads and show that the proposed scheduler performs better in maintaining a high system value with less deadline misses.

CHAPTER 1. Introduction

1.1 Introduction

Real-time systems are being extensively used in wide range of systems and domains, such as flight controllers, autonomous vehicle controllers, nuclear plant controllers, robotics, defense systems and medical systems. The distinct feature of real-time system is the correctness of these computing systems depends not only on the logical result of their computations but also on the time at which the result is produced. Thus, every computational task needs to completed before a deadline and failure to meet the deadline of a task can result in undesirable consequences depending on the criticality of the task.

Real-time systems need to be predictable and reliable. Predictability is a feature that is unique to real-time computing, which requires the ability of the system to determine whether the system will be able to meet the timing requirements of tasks. Ensuring predictability in a real-time system requires scheduler and analytical models that can guarantee tasks, their timing or other service guarantees, apriori or at run time. Reliability is the ability of the real-time system to execute tasks in a reliable manner, subject to certain workload and failure (such as hardware/software faults) assumptions, maximizing the probability of successful execution of tasks. Being fast is not a necessary condition in real-time system, but can be useful, as the system's primary objective is to meet deadlines and scheduling based on average case to improve throughput will harm the first and foremost essential quality, predictability.

1.2 Real-Time Tasks

Since a computational task is time constrained in a real-time system, every task is associated with *deadline*, the time by which a task needs to complete its execution. Traditionally, real-time

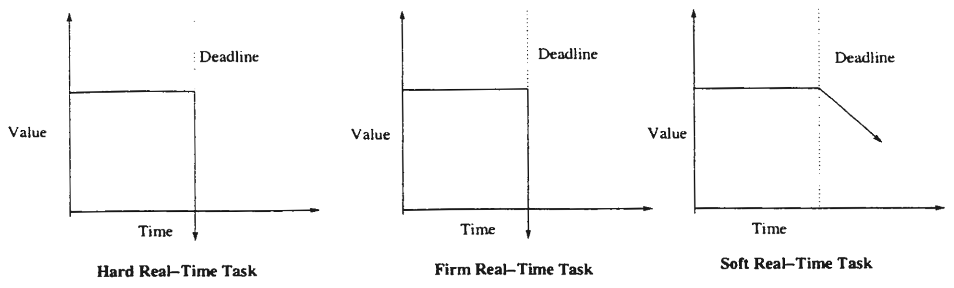

Figure 1.1 Value-Time dependence for real-time tasks

tasks are classified into periodic or aperiodic tasks, based on their activation time. *Periodic tasks* are activated at regular intervals and are characterized by a computation time C and a period (frequency of activation) P, with deadline usually set to the end of period (e.g., sensor acquisition tasks). *Aperiodic tasks* are activated at irregular intervals, due to occurrence of an event (e.g., fault handler tasks) and is characterized by ready time R, computation time C, and deadline D; these parameters are known only upon task's arrival. Real-time tasks are classified into hard, firm or soft [5], based on their criticality and the value they offer to the system upon successful execution. The relation between value and time for these tasks are given in Figure 1.1. Hard tasks are extremely critical tasks, whose deadline misses might lead to catastrophic consequences. Missile controllers, flight controller tasks are examples of hard tasks. Firm tasks are less critical than hard tasks but does not offer value after its deadline (e.g., online transaction processing). Soft tasks are those which offer value (upon execution) to system even after its deadline (e.g., desktop video player).

1.3 Scheduling in Real-Time Systems

The operating systems community has extensively studied scheduling of processes without timing constraints. However, traditional general purpose schemes such as First In First Out, shortest job next or round robin are not appropriate for real-time systems. These scheduling policies aim to reduce the average response time and do not deal with timing constraints. The problem of real-time scheduling is to allocate processors (including resources) and time to tasks in such a way that certain performance guarantees, such as timing requirements and reliability requirements, are met [1]. The process of real-time scheduling involves two steps: (i) *Schedulability Check* - that involves checking to see if the tasks' timing, reliability and resource requirements can be met and (ii) *Schedule Construction* - that involves actual construction of task schedule in such a way that task deadlines are met, in addition to meeting other requirements.

Real-time scheduling algorithms can be classified into four types [1] based on (i) whether the schedulability checking is done offline/online or not at all and (ii) whether schedule construction is done offline/online. Based on this, real-time scheduling algorithms are classified as shown in Figure 1.2. A brief description of each class of algorithm is given as follows:

- *Static table-driven schedulers:* These schedulers perform offline schedulability check and construct the schedule also offline and stores the result in the form of a table to be used by the dispatcher at run-time. This is a highly predictable approach but has the problem of repeating the schedulability check and schedule construction even for a small change in task set.

- *Priority-driven schedulers:* These schedulers perform schedulability check offline but construct the schedule at run-time. Based on the nature of task's priority used by the scheduler, it is classified as follows:

    1. *Static priority-driven schedulers* use static priorities for dispatching tasks e.g., Rate Monotonic Scheduler (RMS) [4]. RMS uses period as the priority metric (lower the period higher the priority), which is determined upon arrival of a task.

    2. *Dynamic priority-driven schedulers* use dynamic priorities for task dispatching such as deadline or laxity. Earliest Deadline First (EDF) [4] is an optimal scheduling algorithm for uniprocessor systems under dynamic priority assignment and uses deadline as priority metric (earlier the deadline, higher the priority). Least Laxity First (LLF) is another optimal scheduling algorithm for uniprocessor systems which uses laxity (defined as the amount of time till which task can wait and still meet its deadline) as the priority metric (lower the laxity, higher the priority).

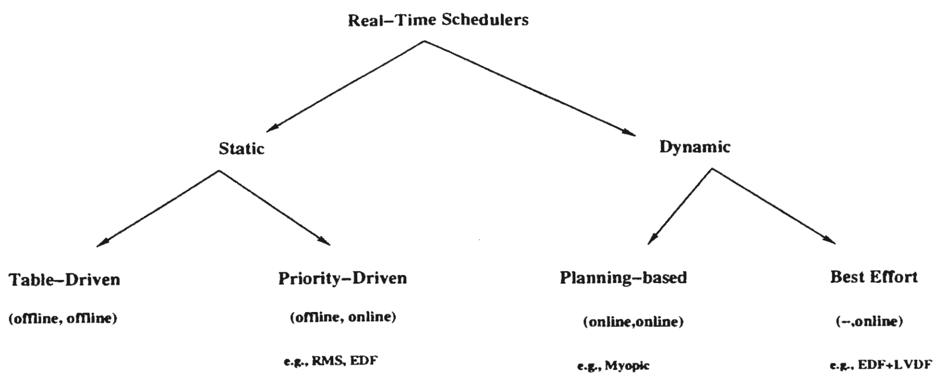

Figure 1.2 Classification of Real-Time Scheduling Paradigms

- *Dynamic Planning-based schedulers:* These schedulers perform schedulability check at run-time and construct the schedule also at run-time, i.e., a dynamically arriving task will be accepted if it is found to be schedulable. These schedulers are used in system wherein the tasks' characteristics are not known apriori. Myopic Scheduler [2] used in Spring Kernel, is an example of dynamic planning-based scheduler.

- *Dynamic best-effort schedulers:* These schedulers do not perform schedulability check and these systems try to do their best in meeting deadlines of the tasks. Any task may be aborted during its execution, as the system gives no guarantee for its completion.

1.3.1 Dynamic Planning-based Schedulers

The problem of dynamic scheduling in a multiprocessor system is to determine when and on which processor a given task is to be executed, with no apriori knowledge of task characteristics. There exists no optimal dynamic scheduler that can schedule tasks with no apriori knowledge [5]. Hence, many heuristics are proposed for this scheduling problem [2, 3]. In dynamic planning-based schedulers, the task characteristics are known only at run-time and the scheduler's job is to ensure predictability in such an environment (using online schedulability check), to see if the task can be guaranteed to meet its deadline. Dynamic planning-based scheduling consists of three main activities: schedulability checking, schedule construction and dispatching (task execution). Planning-based schedulers usually use non-preemptive schedules thereby making schedule construction and dispatching independent. A popular dynamic planning-based scheduler, Myopic [2] is described below.

Myopic Scheduler

Myopic scheduler uses a heuristic search algorithm for scheduling tasks with resource constraints in a multiprocessor system. The algorithm uses a branch and bound search technique wherein a vertex in the search represents a partial schedule and extends the search tree by extending the vertex by selecting one task at a time among a set of tasks, the set is called *feasibility check window.* The schedule from a vertex is extended only if the vertex is strongly feasible. A vertex is strongly feasible only if a feasible schedule can be generated by extending the current partial schedule with each task in the feasibility check window. The larger the size of the window, the higher is the look-ahead nature and the more is the scheduling cost. The algorithms starts with an empty schedule as root node of the search tree and tries to find a fully feasible schedule, leaf node of search tree. The algorithm computes a heuristic function (H) for each task in the feasibility check window, based on the deadline and earliest start time of the task. It then extends the partial schedule by extending the vertex with best (smallest) heuristic value. Otherwise, it backtracks to the previous vertex and then the schedule is extended from there using a task which has the next best heuristic value.

1.4 Motivation: Issues in Dynamic Scheduling

In dynamic real-time scheduling, since the task's characteristics are known only at run-time, ensuring predictability and providing various guarantees, such as timing and reliability guarantees become a challenging problem. Different scheduling approaches are used to address this problem and figure 1.3 outlines these scheduling approaches, their attributes and mechanisms. The first popular scheduling approach is *deadline-based scheduling*, wherein the scheduler schedules tasks based on deadline. This involves schedulability check and schedule construction. These schedulers aim at improving the overall schedulability of the system and perform well during normal workloads. When the system's workload is unpredictable, schedul-

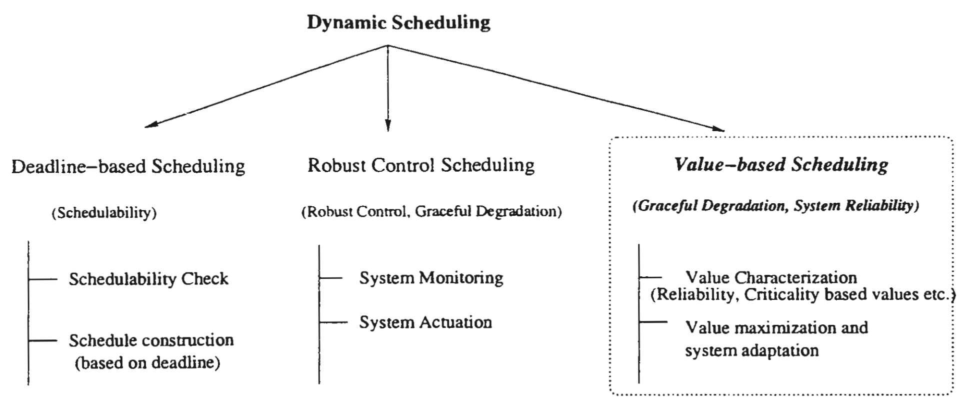
Dynamic Scheduling
Deadline–based Scheduling
(Schedulability)
Schedulability Check
Schedule construction (based on deadline)
Robust Control Scheduling
(Robust Control, Graceful Degradation)
System Monitoring
System Actuation
Value–based Scheduling
(Graceful Degradation, System Reliability)
Value Characterization (Reliability, Criticality based values etc.)
Value maximization and system adaptation

Figure 1.3 Dynamic Scheduling Paradigms

ing approaches such as *Robust control scheduling* [6] and *value-based scheduling* [7] are used to provide graceful degradation and robust performance.

Robust control schedulers are used to achieve robust system performance and graceful degradation amidst unpredictable workload. These schedulers monitor the system performance periodically, and use system level actuators (such as task execution time controllers, admission controllers) to control and achieve required system performance.

Value-based Schedulers are used to provide graceful degradation to the system during system overloads or to achieve high system reliability. These schedulers schedule tasks with an objective of maximizing the overall value (utility) of the system. Value-based scheduling has two important steps: The first step is value characterization, i.e., identifying the value of individual tasks, based on their criticality, reliability, or schedulability and the second step is to schedule tasks to maximize the overall value of the system. Based on the nature of the value functions used, the value-based scheduler also has different attributes. For example, a value-based scheduler employed with a value function based on criticality, schedules task to offer graceful degradation during overloads and a value-based scheduler that employs reliability-based value function schedules task to maintain high system reliability.

The research work presented in this thesis identifies some issues in dynamic real-time scheduling (highlighted in Figure 1.3) and propose value-based scheduling schemes for the identified problems/issues. The following are the dynamic scheduling issues/problems addressed in this thesis:

- **Reliability-aware dynamic scheduling:** We study the problem of guaranteeing reliability requirements at run-time, with the aim of increasing the overall reliability of the system, without affecting the schedulability of the system. We have observed the tradeoff between schedulability and reliability in a real-time system and propose a dynamic planning-based scheduler that can guarantee reliability requirements at run-time and also maintain a high schedulability (low deadline misses).

- **Dynamic scheduling under various workloads:** We address the problem of dynamic real-time scheduling under varying workloads such as lightly loaded, fully loaded and overloaded systems. We observe the need for predictable scheduling behavior during overloads, but also the need for maintaining high schedulability (maintaining low deadline misses) during underloads or near full loads. Further, we study this problem in the context of heterogeneous real-time systems, wherein the processors have heterogeneous computing capabilities, and propose two dynamic real-time schedulers for these kind of systems.

1.5 Contribution of this thesis

The work presented in this thesis is a continuation to the research efforts in the area of value-based scheduling in real-time systems. The first contribution of this thesis is a reliability-aware value-based scheduler for multiprocessor real-time systems, which aims at maintaining a high system reliability and schedulability at the same time. The second contribution of the thesis are new adaptive value-based scheduling schemes for homogeneous and heterogeneous multiprocessor real-time systems that can maintain a high system value with minimal deadline misses during overloads and maintain high success ratio during underloads and near full loads. The third contribution of this thesis is the design and implementation of the proposed adaptive value-based scheduling scheme in a real-time Linux operating system, RT-Linux [23] and evaluating the effectiveness of the proposed scheduler through experimental evaluations.

1.6 Organization of the thesis

Chapter 2 describes in detail the issues in dynamic real-time scheduling, such as reliability-aware scheduling, robust scheduling in various workloads and motivates the need for proposed scheduling approaches. Chapter 3 presents the reliability-aware value-based scheduler, chapter 4 presents the proposed adaptive value-based dynamic scheduling algorithms for homogeneous and heterogeneous computing systems. Chapter 5 presents the implementation details of the proposed adaptive value-based scheme in RT-Linux and the performance results of the scheduler in comparison with two other existing schedulers. Chapter 6 presents the conclusion drawn from the experiment results, and suggests possible future works.

CHAPTER 2. Fault Tolerant Scheduling and Overload Handling

2.1 Introduction

Multiprocessors and multicomputers based systems have emerged as a powerful computing means for real-time applications such as avionic control and nuclear plant control, because of their capability for high performance and reliability. The use of real-time systems in systems handling dynamic events, such as missile control systems, autonomous vehicle controllers, radar controller systems and distributed sensor networks [24] demands efficient dynamic scheduling algorithms. The traditional myopic [2] or parmyopic [3] scheduling algorithms fail to capture various issues in dynamic scheduling. In this chapter, we present some issues that needs to be addressed in the context of dynamic scheduling and motivate the need for value-based scheduling techniques for the identified problems.

The rest of the chapter is organized as follows: In Section 2.2, we discuss the need for fault-tolerance in real-time systems, in section 2.3 we discuss the issues and traditional approaches in tolerating physical faults in a real-time system, identify the tradeoff between reliability and schedulability and need for a dynamic scheduler that captures this tradeoff. In section 2.4, we discuss the issues that arise due to overloads in dynamic real-time systems and motivate the need for a scheduler to exhibit adaptive scheduler.

2.2 Fault-tolerance in real-time systems

As mentioned earlier, tasks in real-time systems are critical in nature. In critical applications, such as space shuttle controllers and nuclear power plant controllers, it is important that tasks meet their deadlines under all circumstances. These circumstances includes faults generated by various factors such as power fluctuations, processor failure, software bugs, etc.

Fault-tolerance is defined as the ability of the system to deliver the expected service in the presence of faults. Though the issue of fault-tolerance is addressed in other fields of computing, the solutions cannot be directly applied in real-time systems as the dependability requirements [25] cannot be addressed independent of timing requirements. In other words a real-time system may fail to function correctly either because of faults in its hardware and/or software faults or because of not responding in time due to overloads (timing faults). Hence, to avoid the catastrophic consequences of missing deadlines, it is essential that real-time tasks meet their deadlines even in the presence of faults and/or overload conditions.

2.3 Tolerating system faults

The need for fault-tolerance in real-time systems is recognized by both real-time and fault-tolerance research communities. In [26], authors identify the need for fault tolerance in real-time systems by saying that "real-time systems must be sufficiently fault-tolerant to withstand losing large portions of hardware or software and still perform critical functions". Thus, research in fault-tolerant systems has led to the development of *responsive systems* [27], MARS system [28] and HARTS [29].

Faults are tolerated in real-time systems using redundancy [14], which are of three kinds: hardware redundancy, software redundancy and time redundancy. Hardware redundancy is the addition of extra hardware to the system, such as spare processors that are used in case of failure of running processor. Software redundancy is the use of extra software modules to verify the results, or to use multiple versions of a program. Time redundancy is providing additional time to task to re-execute in case of failure.

2.3.1 Real-time fault tolerant scheduling algorithms

The goal of a fault-tolerant scheduling algorithm is to guarantee the recovery of real-time task in case of failure. As stated earlier, the faults are usually tolerated by executing multiple versions of the task either in the same processor or in multiple processors. More than one version of a task can be scheduled on a single processor, if faults expected are only transient (temporary malfunction) in nature. If the faults are permanent in nature, such as failure of processor, then multiple versions of a task needs to be scheduled in different processors. Two major techniques evolved for fault-tolerant scheduling of tasks and are given as follows:

- **N-Version Programming [30]:** Here multiple versions of tasks are executed concurrently in different processors and the results produced by these processors are voted on. The voter, which is assumed to be reliable, compares the output and selects the majority vote. This approach is based on the principle of design diversity, wherein multiple versions of the same task are created by employing different algorithms, different languages, and/or programmers.

- **Recovery Blocks [31]:** This technique uses multiple alternates to perform same function. The version that is executed first is called the primary, and the others secondary. At the end of execution of primary, an acceptance test determines whether the output is acceptable or not. If not, the secondary versions are executed until an acceptable output is obtained or the deadline is missed.

2.3.2 Reliability vs. Schedulability

As stated above, fault-tolerance in real-time systems is realized by redundancy, using techniques such as NVP. In NVP, the number of versions that are concurrently executed determines the reliability of execution of a task. Higher the number of versions executed concurrently, higher is the reliability of a task. For example, in a 10-processor system, with each processor having a reliability of 0.9, the reliability of a task executing with one version is 0.9, with two versions is 0.99 and with 3 versions is 0.999. Thus, it can be seen that with increase in number of versions executed, the task reliability increases. However, it must be noted that increasing number of versions of one task can lead to missing of deadlines of other tasks in the system. Thus, it can be seen that there is a tradeoff between reliability and schedulability in a real-time system and it is the responsibility of the scheduler to capture this tradeoff.

2.3.3 Related Work

The primary issue in fault-tolerant scheduling is to find the correct redundancy level for each task so that the system's reliability is maintained with high schedulability. Various scheduling algorithms were proposed to capture this tradeoff between reliability and schedulability. Some of them are described as follows:

Spare-capacity scheduling: In [15], an algorithm was proposed with fault-detection and location capabilities for real-time systems based on the myopic algorithm. The objective of the algorithm was to improve the schedulability and to use the spare capacity of idle processors for fault-detection and location. Thus, the algorithm's emphasis is on schedulability of the system than reliability. Further, this algorithm does not consider the value parameters into account for making a scheduling decision. Such an algorithm is inadequate when tasks have different rewards (or penalties) for their successful executions (or failures).

Performance index scheduling: In this context, an approach is proposed in [12] to determine the redundancy level for a given set of tasks so as to maximize the total performance index which is a performance-related reliability measure. This approach is used in dynamic planning based scheduling where decisions are taken as tasks arrive. In such a system with m processors, n tasks arrive at a particular point in time. The approach tries to find the best redundancy level for each task such that the overall performance index is maximized. Once a task redundancy level is determined, a task is said to be guaranteed if the given number of copies of the task are all scheduled to complete before the task's deadline. Suppose a task T_i provides a reward V_i, if it completes successfully once it is guaranteed, a penalty P_i if it fails after being guaranteed, and penalty Q_i if it is not guaranteed. Let R_i be the reliability of a task and F_i be its failure probability, where $R_i = 1 - F_i$. The redundancy level of a task T_i and the failure model of the processors affect R_i. The performance index PI_i for task T_i is defined as

$$PI_i = \begin{cases} V_i R_i - P_i F_i & \text{, if } T_i \text{ is guaranteed} \\ -Q_i & \text{, if } T_i \text{ is not guaranteed} \end{cases}$$

Then, the performance index for a task set containing n tasks is defined as:

$$PI = \sum_{i=1}^{n} PI_i$$

In [16], another dynamic fault-tolerant scheduling algorithm was proposed, where the myopic algorithm was modified to schedule tasks aiming to maximize the overall PI of the system. However, the algorithm by its nature selects the maximum redundancy level (an input parameter) as the chosen redundancy level for each task. Such an approach of considering only one task at a time for determining the redundancy level for the task set can lead to poor overall performance index, which is the motivation for the first work of the thesis.

2.4 Overload Handling in Real-Time Systems

As mentioned earlier, dynamic real-time scheduling algorithms need to ensure predictability without having apriori knowledge of task characteristics. Dynamic algorithms such as [2, 3] behave in a predictable manner during underloads when the system is not utilized to its fullest capacity. However, these algorithms do not behave in a predictable manner during overload conditions and can lead to instability of the system due to missing of deadlines of tasks that are critical to the functioning of the system. Hence, a new scheduling paradigm called *value-based scheduling* is employed, which schedules tasks such that the overall value (i.e., utility) of the system is maximized and allows the system to degrade gracefully during overloads. It is assumed that each task offers certain “value” to the system, if it meets its deadline. Thus, value-based scheduling is a decision problem involving the choice of tasks to execute so that the overall system value is maximized [7]. However, this problem is also intractable and several heuristics have been proposed [8, 9].

Such a scheduling is of immense use in a flexible real-time system, wherein the system is expected to take decisions at run-time for efficient resource usage and also for system stability and any design phase decision might lead to pessimistic usage of resources. An example of a flexible real-time system is an autonomous vehicle controller [7], wherein the system needs to exhibit intelligent and adaptive behavior in order to function in a highly dynamic and non-deterministic environment characterized by unpredictable nature of other vehicles, route information, weather and road conditions. Such a real-time system has two conflicting objectives: (1) guaranteeing safety and mission critical tasks to provide results of acceptable quality and (2) to increase the system utilization (and schedulability) determined by frequency, timeliness and precision, by scheduling as much tasks as possible. A typical example for need of adaptive behavior in such a system can be understood by the following scenario: In the autonomous vehicle controller system, under normal driving scenarios, a service such as headlight controller that determines the amount of light to be flooded based on the weather conditions must be scheduled in time for obtaining proper lighting performance. However, such a service is of least importance when the car is in verge of collision and services such as collision control and brake control needs to be run with the highest priority. Hence, the system needs to adapt itself to the environment and conditions accordingly.

In [11], the tradeoff between *value vs. deadline* scheduling has been studied in detail. The paper concludes that there is a need for different scheduling behaviors under different load scenarios. Consider the following simple example with two tasks in a uniprocessor system, where a task T_i is characterized by $< v_i, c_i, d_i >$ (v_i - value offered by T_i, c_i - computation time of T_i, d_i - deadline of T_i) : $T_1 : < 10, 20, 30 >$, $T_2 : < 100, 50, 80 >$. For scheduling these two tasks in a real-time system at time 0, scheduling based on deadline will meet the deadline of both the tasks (with T_2 scheduled after T_1), if the system is lightly loaded and execute both the tasks before their deadlines. However, if the system is highly loaded, then a deadline scheduling scheme might result in missing the deadline of a higher valued task. Hence, during overloads, value-based scheduling scheme is preferred (with T_2 scheduled before T_1). Thus, it can be clearly seen from this simple example that there is a need for different scheduling behavior such as deadline based scheduling under light loads and value-based scheduling during overloads. The first part of the second work of this thesis is to develop an adaptive value-based scheduler for homogeneous computing systems that adapts its scheduling behavior based on the current workload.

2.4.1 Real-time scheduling in Heterogeneous Computing Systems

Most of the multiprocessor scheduling algorithms in the literature [2, 3] assume that the processors have homogeneous computing capabilities. However, an emerging trend in computing is to use distributed heterogeneous computing (HC) systems [17, 18], wherein the machines/processors differ in their computing capabilities. Though significant amount of research has been done on scheduling algorithms for heterogeneous systems [19, 20], these algorithms are studied in the context of non real-time systems. For example, several algorithms were proposed in [20] for HC systems aiming at minimizing the schedule length, which is not a suitable metric for scheduling in real-time systems. In [21], a reliability driven real-time scheduling algorithm is proposed for HC systems, where scheduling is done based on the reliability of a task on a processor to improve the reliability. However, even the issue of dynamic real-time scheduling in a HC system itself is yet to be clearly addressed. As a part of the second component of this thesis, we propose a dynamic scheduling algorithm for HC systems that maintain a high system value with minimal deadline misses.

In this thesis, we propose value-based scheduling techniques for the above two identified problems: (i) reliability-aware scheduling and (ii) dynamic value-based scheduling (with minimal deadline misses).

CHAPTER 3. Reliability-Aware Value-based Scheduler

3.1 Introduction

In this chapter, we present a reliability-aware value-based scheduler for a multiprocessor real-time system. The objective of the proposed scheduler is to increase the overall reliability of the system with less degradation in schedulability at the same time. In this chapter, we consider the fault-tolerance related value function, PI, that captures the tradeoff between schedulability and reliability, and propose a scheduling algorithm to maximize the overall performance of the system. Further, the proposed dynamic scheduler differs from traditional dynamic real-time scheduling (search) algorithms, as it extends the task schedule with more than one tasks at a time. The reason for such a task scheduling is as follows: Since PI of a task increases with increase in redundancy levels, just considering only one task at a time for extension of schedule will always lead to selection of the highest value task with highest redundancy level of execution. Hence, it would be beneficial to extend the schedule with more than one task at a time in terms of yielding a better PI. Thus, the proposed reliability-aware value-based scheduler uses a dynamic branch and bound search algorithm similar to myopic that extends the task schedule possibly with more than one task at at time. The rest of this chapter is organized as follows: In section 3.2 we present the system model and terminology used in the chapter and present the proposed scheduler in section 3.3. We evaluate the performance of the proposed algorithm in section 3.4.

3.2 System Model and Terminology

- The system model is shown in Figure 3.1. As shown in the figure, system consists of task queue in which the real-time tasks enter the system, which are scheduled by the scheduler and placed in the appropriate dispatch queues of processors.

- The system consists of m processors having identical reliability.
- Each task T_i is characterized by ready time (r_i), worst-case computation time (c_i), deadline (d_i), reward value (V_i) and penalty values (P_i and Q_i). Further tasks are assumed to be aperiodic and non-preemptable.
- Let p be the reliability of a task (with one version), T_i, executing on a single processor. Then, the reliability (R_i) of the task with r versions is given by $R_i = 1 - (1-p)^r$.
- A task T_i is said to feasible only if $EST(T_{i1}) + c_i \leq d_i$, where $EST(T_{i1})$ denotes the earliest start time for task T_i with one version.
- Let A be the set of tasks in the feasibility check window, of size K, then the partial schedule obtained is *strongly feasible* if all the schedules obtained by executing the current schedule to the unscheduled task set is also feasible. i.e., $\forall i, T_i \epsilon A, EST(T_{i1}) + C_i \leq d_i$
- The performance of a task T_i for a redundancy level j is defined as :

$$PI_{ij} = \begin{cases} V_i R_i - P_i F_i, if j > 0 \\ -Q_i, if j = 0 \end{cases} \tag{3.1}$$

where, $F_i = (1-p)^j$ and $R_i = 1 - F_i$.

3.3 Proposed Reliability-aware Value-Based Dynamic Scheduler

In this section, we present the reliability-aware dynamic scheduling algorithm that aims at maximizing the overall value (performance index) of the system by finding the right redundancy level for each task. The proposed algorithm is a variant of myopic algorithm. The myopic algorithm works on extending on a partial schedule P, by one task at a time. Initially it considers all the K tasks (in the feasibility check window) and checks if they are feasible. If it is strongly feasible then it extends the schedule by task T_i that offers the smallest heuristic

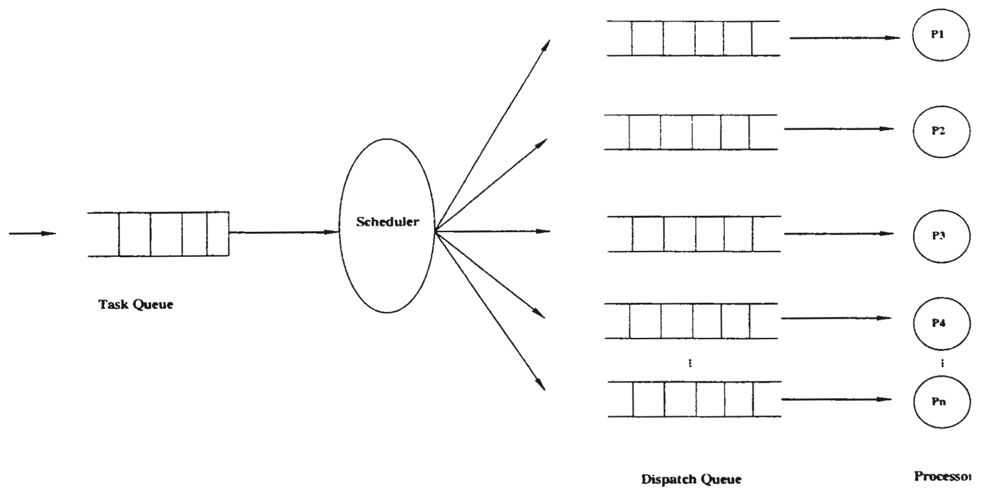

Figure 3.1 System Model

value $H(.)$, out of the all the tasks in its feasibility check window. If the current schedule is not strongly feasible, the algorithm backtracks and selects the task that offers the next best heuristic value.

The proposed scheduling algorithm, as given in Figure 3.3, starts of similar to that of the myopic scheduling algorithm, by considering K tasks in the feasibility check window, A. If the current schedule is strongly feasible, then the algorithm selects the right combination of task with their appropriate redundancy level by the *Combination Selection Algorithm*, such that the overall performance index of the task set in A is maximized and the total number of tasks (including the redundant tasks) is less than or equal to L, the maximum schedule extension size. Then, the scheduling order of the tasks is determined by the *Order Selection Algorithm*, and the schedule is extended with specified order of the selected combination. If the selected order for a combination is not feasible to schedule, then task schedule is extended with different order until a feasible order is obtained. If all the orders are exhausted, then the task schedule is extended with next best combination that offers the next best PI value with the order being selected for the new combination. This process is repeated until a feasible schedule is obtained or until all combinations are exhausted. The key difference between the proposed scheduler and the conventional myopic scheduler is that in the former, the schedule is extended by L tasks at a time, while it is extended by only one task at a time at the latter.

Here, the value of L can be less than or equal to or more than the number of processors in the system. The value of L affects the effectiveness of the algorithm, with increased value of L reducing the total number of scheduling decisions. Similarly, the value of K also affects the effectiveness of the algorithm, as K represents the look-ahead nature of the scheduler. Higher the value of K, higher is the number of tasks considered in the search and hence the chance of the scheduler to come up with a task set giving a better value is increased.

3.3.1 Combination Selection Algorithm

The objective of the combination selection algorithm is to come up with right combination of tasks with their appropriate redundancy levels such that overall performance index of the system PI is maximized. Thus, the combination selection algorithm takes the task in the feasibility check window as input and extends the schedule upto L tasks, such that the selected combination gives the maximum PI. The combination selection algorithm can be considered as a search algorithm, which searches for the correct combination of redundancy levels of tasks that leads to the maximum PI, which are feasible to schedule. Hence the issue in this algorithm is the depth of search to be carried for every task-set scheduling to attain a good combination yielding the best or near-best PI. For example, if the number of tasks within the feasibility check window A is K and the maximum redundancy level of a task is r, then the combination selection at the maximum must search all combinations (of order $O(r^K)$) to guarantee an optimal solution. We propose two combination selection algorithms, *Exhaustive Search* and *Reduced Search*, in this section that can come up with an optimal or near-optimal solution with the first algorithm incurring high computation cost for optimal solution whereas the second algorithm incurs less computation cost for near-optimal solution.

3.3.1.1 Exhaustive Search Algorithm

The exhaustive search algorithm comes up with the best combination of tasks after an exhaustive search of all task combinations (within the feasibility check window). The algorithm is as follows:

```
Value-Based Scheduler()
Input: Task Set to be scheduled.
Output: Feasible Schedule (maximizing the $PI$) or failure.

 1. Tasks (in the task queue) are ordered in non-decreasing order of deadline.
 2. Start with an empty partial schedule.
 3. Check if the set of tasks (set A, $|A|$=$K$), constituting the feasibility check window is strongly
    feasible.
 4. If ( strongly feasible)
    Repeat
      • Combination Selection Algorithm: $\forall T_i \epsilon A$, find $j$ (=$r_i$) for each task such that:
                $PI = \sum_{i=1,j=1}^{K,r} PI_{ij}$ is maximized.
      • Repeat
        Order Selection Algorithm:
          – $\forall T_i \epsilon A$, compute a heuristic value H(.) for the selected redundancy level $r_i$
          – sort the tasks based on their H(.).
          – The sorted order represents the dispatching order of the task set.
        Until (Termination Condition 1)
    Until (Termination Condition 2)
 5. if (a feasible combination is obtained)
      • $\forall T_i \epsilon A$, extend the schedule by $T_i$ with $r_i$ versions
      • Move the feasibility check window $A$, to next $K$ tasks.
    else
      • Backtrack to the previous partial schedule.
      • Extend the schedule with the next best combination.

Termination Condition 1: Feasible order is obtained or the maximum number of orders have been
searched.
Termination Condition 2: Feasible combination is obtained or the maximum number of combinations
have been searched.
```

Figure 3.2 Reliability-Aware Value-Based Scheduling Algorithm

1. Let the number of the tasks in the feasibility check window be K and the maximum redundancy level to be considered be r.

2. Calculate the performance index for all combinations of task sets, such as n_1 version of T_1, n_2 version of T_2 ,..., n_K versions of T_K, where $0 \leq n_i \leq r$ and $1 \leq i \leq k$ and $\sum_{i=1}^{k} n_i = L$, where L is the maximum schedule extension size.

3. Select the combination of (i, j) for each task T_i and its redundancy level j that offers the best overall performance index PI, where:

   $PI = \sum_{i=1,j=1}^{K,r} PI_{ij}$

Though the algorithm is always guaranteed to come up with an optimal combination for the set of K tasks at any given time, its time complexity is O(r^K), which makes it very expensive.

3.3.1.2 Reduced Search Algorithm

This algorithm tries to reduce the search space by reducing the number of redundancy levels to be searched for each task, so that the number of combinations searched is less than that done by the *Exhaustive Search Algorithm*. This algorithm reduces the search space by taking smart decisions on the maximum redundancy level to be searched for each task. The algorithm finds the maximum redundancy level for each task, by finding the right redundancy level beyond which the gain in performance index for increasing redundancy is not high, which is determined by Δ (an input parameter to the system). The reduced search algorithm is as follows:

1. Let the number of the tasks in the feasibility check window be K and the maximum redundancy level to be considered be r.

2. For each task T_i, fix the maximum level of redundancy level to be searched as $j(=r_i)$, such that $PI_{i(j+1)}$ - $PI_{ij} \leq V_i * \Delta/100$, where Δ is a fixed value (such as 1).

3. Calculate the performance index for all combinations ($r_1 * r_2 * \ldots * r_K$) of task sets, such as n_1 version of T_1, n_2 version of T_2,... , n_K versions of T_K (where $0 \leq n_i \leq r_i$ and $1 \leq i \leq k$)

To illustrate the working of this algorithm lets consider the following example. Consider a task T_i for $\Delta = 0.02$, with following characteristics: $< v_i, p_i, q_i > = < 10, 100, 5 >$ and with the reliability of the processor $p = 0.9$. For this task set, the PI values for different redundancy levels are: $PI_{i1} = -1, PI_{i2} = 8.9, PI_{i3} = 9.98, PI_{i4} = 9.998$. Since for this task, $PI_{i4} - PI_{i3} = 0.018 \leq (v_i * \Delta = 10 * 0.02)$, the algorithm eliminates all combinations of task T_i with redundancy level more than 3 in its search. The intuition behind pruning search in this manner is that the reward for searching for more than a redundancy level of 3 for the given is not high (as determined by the Δ factor). Though, the reduced search algorithm need not always come up with a best possible PI for every task set, it incurs a less computation cost for every combination selection process for a task set. The amount of search executed and how close the solution is to the best combination offering the best PI, is controlled by the Δ parameter. With increased value of Δ, the number of redundancy levels searched for each task is also increased, thereby increasing the search space and improving the overall PI.

3.3.2 Order Selection Algorithm

The objective of the order selection algorithm is to order the task execution sequence, for a given combination of a task set, such that the selected combination of task set is feasible to schedule. The order selection algorithm is an important component in the proposed scheduler as the feasibility of scheduling a selected combination depends on the effective dispatching of the tasks. This is because in a dynamic scheduling environment the task dispatch order is not a trivial decision as it involves many factors such as the earliest available processor time, task ready time and earliest resource availability time. So the task dispatching order must take into account these constraints to increase the chances of meeting tasks deadlines. A mere dispatching of task with highest PI with their redundancy level is not desirable, as this can introduce "holes" (idle processor time) in the schedule, leading to missing of deadlines of many other tasks. In this section, we propose two order selection schemes, *Deadline Ordering* and *Heuristic Ordering*, which orders the selected task set combination based on deadline and heuristic functions respectively.

3.3.2.1 Deadline Ordering Scheme

The deadline ordering scheme determines the order of task execution, based on how close the tasks are to their deadline, for all the tasks in a given task combination. So, tasks having smaller deadlines will be scheduled first. The disadvantage of this scheme is that it does not take into account the Earliest Start Time (EST) of a task, which can introduce a lot of holes, thus resulting in poor schedulability.

3.3.2.2 Heuristic Ordering Scheme

The heuristic ordering scheme overcomes the disadvantage of the deadline ordering scheme by taking into account the EST of a task for finding the appropriate task dispatch order. This scheme uses a heuristic function which captures the importance of deadline and also EST of a task, which is given by:

$H(T_i) = d_i + W * EST(T_i)$

The W factor determines the weight given to the EST of a task. So, lower the value, means lesser the importance given to the EST. So, the tasks are ordered based on their heuristic value $H(.)$ and dispatched in the sorted order. As explained in the *FT-Myopic algorithm*, if a selected order for a task set is not feasible, then a different order can be obtained by varying the W parameter. A special case of this scheme for $W = 0$ is the deadline ordering scheme.

3.3.3 Illustration for the proposed value-based scheduler

Consider a real-time system with four processors. Assume that five tasks $\{T_1, T_2, T_3, T_4, T_5\}$ are to be scheduled with all tasks having the same computation time and deadline but different reward values, as given in Table 1. Assume the reliability of the processor to be 0.9, $K = 3$, L=4 and r=3.

Table 3.1 Example Task Set

task	r_i c_i d_i	V_i P_i Q_i
T_1	0 5 10	10 100 1
T_2	2 5 10	20 150 2
T_3	0 5 10	10 80 0.8
T_4	3 5 20	30 200 3
T_5	0 5 20	8 50 0.5

Then the algorithm behaves as follows: First the combination selection selects algorithm selects T_2 and T_3 out of the first 3 tasks (T_1, T_2, T_3). Then the ordering algorithm (Heuristic Ordering) comes up with schedule of T_{21} on P_1, T_{22} on P_2, T_{31} on P_3, T_{32} on P_4. Then, in the next schedule, out of 3 tasks (T_1, T_4 and T_5), T_1 and T_4 are selected with a redundancy level of 2 each. The order of dispatch is of T_1 first (if EST or Deadline ordering scheme is used) and T_4 next and then T_5 is scheduled with redundancy level of 4, as no other task can be scheduled.

3.4 Performance Evaluation

In this section, we first discuss the method adopted for task generation and simulation and then present the simulation results.

3.4.1 Task Set Generation for Value-based Scheduling

The objective of the proposed scheduling algorithm in this paper is to obtain a feasible schedule for a set of tasks, if such a schedule exists, such that the overall value (performance index) of the system is maximized. Heuristic algorithms cannot be guaranteed to achieve this everytime but one heuristic algorithm can be considered better than another, if it obtains a better value and a feasible schedule, given a schedulable task set. This is the basis of our simulation study. Hence, in our simulation study we generate a tightly schedulable task set for m processor system, with appropriate redundancy level determined during the task generation phase, based on its value. Thus, the task generator generates tasks to guarantee almost maximum utilization of the processors and also determines the redundancy level of each

Table 3.2 Simulation Parameters

Task Parameter	Value Range
C_i	$10 \cdots 40$
V_i	$50 \cdots 200$
P_i	$(2 \cdots 4) * V_i$
Q_i	$V_i/10$

processor based on its value. So, higher the value of a task, higher its redundancy level. The inputs to the task generator are given in Table 2.

The schedule generated by the task is in the form of a matrix M which has S (given by schedule length) columns and r rows. Each row represents a processor and column represents the time unit. The task generator starts with an empty matrix, then the value parameters and computation time of a task are chosen randomly using a uniform distribution between the values shown in Table 2. Then, it determines the redundancy level for each task based on its value, such that it is proportional to the value of the system and fills up the appropriate columns and rows of the matrix. Thus, the task generator generates task until the utilization left is less than the minimum computation time of a task. Here, the task's deadlines are chosen to be their finish time of the generated task, which gives very little leeway for the scheduler. We believe that the task set generated by this task generator can evaluate various heuristics in a rigorous manner.

3.4.2 Simulation Method

In our simulation, N schedulable task sets are generated and the values obtained for the schedule during task generation is recorded. Performance of the the proposed algorithms, ES-D (Exhaustive Search and Deadline Ordering), ES-EST (Exhaustive Search and Earliest Start Time Ordering), RS-D (Reduced Search and Deadline Ordering) and RS-EST (Reduced Search and Earliest Start Time ordering) was evaluated based on the following metric:

- $ValueRatio$: It is defined as the ratio of the value obtained by the scheduler to the value obtained during task generation. It must be noted this value can vary from 0 to

1 (and also above 1 some cases, because the value of the schedule obtained during task generation need not be optimal).

$$ValueRatio = \frac{Value_{sched}}{Value_{generator}} \qquad (3.2)$$

where, $Value_{sched}$ = Value obtained by the proposed scheduler, $Value_{generator}$ = Value obtained during task generation, this is used as the baseline value for comparison.

Thus, all the four algorithms are evaluated for varying values of maximum schedule extension size (say L), i.e., the maximum number of tasks with which a schedule is extended for every scheduling decision, the feasibility check size window (K), Δ (for the Reduced Search Algorithm). The simulation was run for 10 sessions of feasible task sets, each consisting of 50 tasks and each simulation was run 5 times and the averages are plotted.

3.4.3 Effect of Schedule Extension Size (L)

In this experiment, the efficiency of all the schedulers were tested for varying values of Schedule Extension Size, L. Recall that one of the motivations for our work is to extend the task schedule by more than one task at a time in order to improve the overall PI. The results of the experiment conducted for a 4-processor system and 5-processor system, with $K = 4$ is given in Figures 3.3 and 3.4.

As it can be seen from the figures, increase in L gives a better performance with respect to the value ratio for both 4-processor and 5-processor systems. This is due to the fact that the scheduler can make a more effective scheduling decision, when it is allowed to extend its schedule with more tasks (as the number of redundancy levels that can be searched for each task will also be high, thereby leading a higher value). However, it must be noted that higher values of L requires more computation for making a scheduling decision as the scheduler's search space increases. Comparing the performance of the four algorithms it can be seen that the Exhaustive Search algorithms performing better than the reduced search counterparts. However, the decrease in value ratio due to the reduced search is not that high, hence there is a good reward for going for reduced search as it obtain schedules that also have

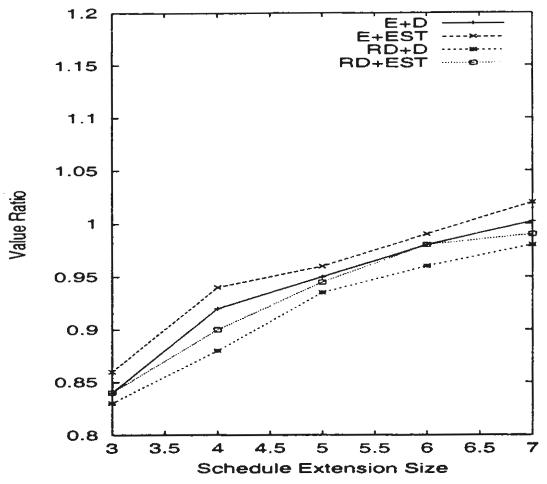

Figure 3.3 Effect of L in a 4-processor system

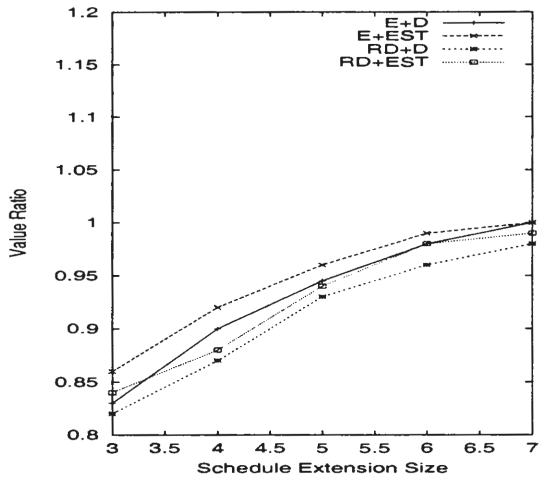

Figure 3.4 Effect of L in a 5-processor system

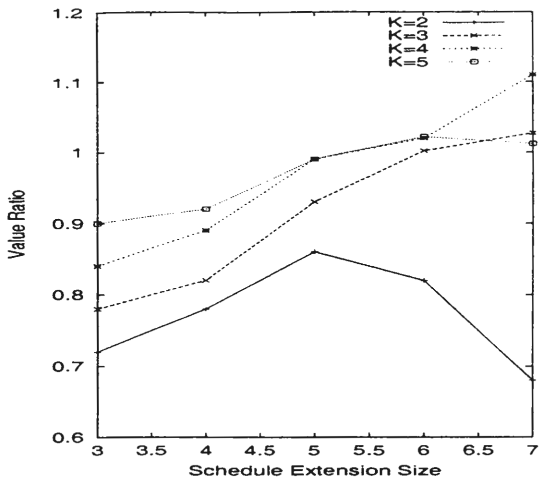

Figure 3.5 Effect of K on 4-processor system for ES+EST Scheduler

comparable value ratio. It must be noted that in this simulation, the resource constraints have not been simulated, which would establish the superiority of EST ordering schemes over Deadline ordering schemes. However, still it can be observed that in the current simulation setup, the EST ordering algorithms do better than deadline ordering schemes as the amount of holes (idle processor time) created in the schedule is decreased.

3.4.4 Effect of Feasibility Check Window Size (K)

In this experiment, the efficiency of the ES+EST scheduler was tested for different values of K and for different values of L. The results of the experiment conducted for 4-processor and 5-processor system are given in Figure 3.5 and Figure 3.6. The effect of K was studied only for ES+EST scheduler as it is the one that gives best performance among all the schedulers and can measure the effect of K on $ValueRatio$ effectively. The K is studied for varying L parameters as both of these parameters affect the scheduling performance together, as explained below.

As it can be seen from the figures, the increase in K increases the value ratio of the schedulers. This is due to the fact that the K represents the look-ahead nature of the scheduler and higher its value, the more effective is its scheduling decision, leading to higher value of the system. However, the performance gain obtained by increasing K from 4 to 5 is not really huge, hence a more complex search is not that rewarding for a small performance increase in

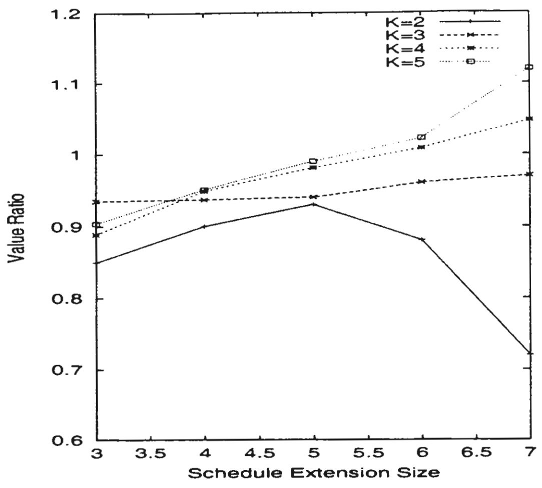

Figure 3.6 Effect of K on 5-processor system for ES+EST Scheduler

this setup. Further, it can be seen from the figures that as for a given K (say 2 in the figures), the $ValueRatio$ obtained decreases with increase in L beyond a value. For example, in the figures it can be seen that for K of 2, the $ValueRatio$ obtained decreases beyond L=5. This is due to the fact that extending a task schedule with smaller number of tasks (determined by K), each having more versions (determined by L) yield less value. Thus, it is necessary that L parameter should be high if the K is also high to get a better performance. It must be noted that this problem of scheduling more versions of tasks due to small sized feasibility check window is eliminated by the reduced search algorithms as they determine the maximum redundancy level of a task and does not extend the schedule more than the determined levels, irrespective of the value of L.

3.4.5 Effect of Δ

In this experiment, the efficiency of the RS+EST scheduler was tested for different values of Δ for 4 and 5 processor systems. The results of the experiments are shown in Figure 3.7 and 3.8 for L=7.

As it can be seen from the figure that with decreasing values of Δ, the $ValueRatio$ obtained increases for both the systems. This is due to the fact that the search space increases with increase in Δ, thereby leading to a better performance. However, the performance gain

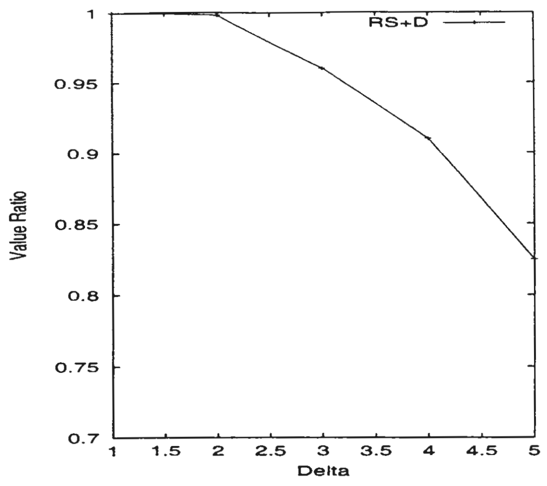

Figure 3.7 Effect of Δ on 4-processor system

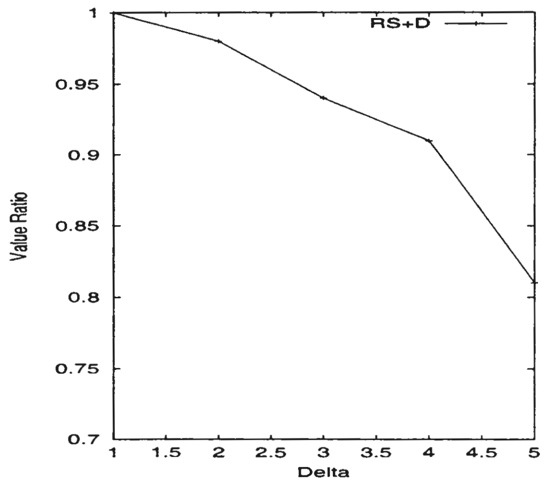

Figure 3.8 Effect of Δ on 5-processor system

obtained from $\Delta = 3$ to $\Delta = 1$ is not high. Hence, it can be clearly seen that unnecessary searching is reduced by the algorithm.

CHAPTER 4. Adaptive Value-Based Scheduling Algorithms for Homogeneous and Heterogeneous Computing Systems

4.1 Introduction

In this chapter, we propose adaptive value-based scheduling algorithms for homogeneous and heterogeneous computing systems that can maintain high system value with minimal deadline misses for all ranges of workloads. First, we propose an adaptive value-based scheduling scheme for homogeneous computing systems and extend the proposed adaptive value-based scheduling scheme to HC systems to take into account the heterogeneity in computing. Then, we evaluate the performance of proposed scheduling schemes using extensive simulation studies and compare their performance with traditional deadline scheduling schemes [2]. The rest of the chapter is organized as follows: Section 4.2 present the system and task model, section 4.3 presents our proposed adaptive value-based scheduling algorithms and their performance evaluation results are presented in section 4.4.

4.2 System Model

- The system consists of m processors, with non-homogeneous computing capabilities for a HC system. The reliability of processors are assumed to be equal. The system model is the same scheduler-dispatcher model used in the previous chapter.

- The total system utilization, U is given by $U = \sum_{i=1}^{m} U_i/m$, where U_i gives the utilization of processor P_i.

- The tasks are assumed to be aperiodic and have no resource or precedence constraints.

- Each task T_i entering the system is characterized by $< a_i, \widehat{C_i}, d_i, v_i >$, where a_i represents the arrival time of the task, d_i represents the deadline and v_i represents the value offered by the task to the system. $\widehat{C_i}$ is the computation time vector of T_i that contains the computation times of the task for m processors ($\widehat{C_i} = < C_{i1}, C_{i2}, ..., C_{im} >$). For example, C_{ij} denotes the computation time of task T_i on processor P_j. In case of homogeneous system, the task is just characterized by C_i, as the task will take the same time for computation in each processor.

- A task T_i is said to be feasible to schedule in a processor P_j, if $EST_{ij} + C_{ij} \leq d_i$, where EST_{ij} is the earliest start time of T_i on P_j = max $\{P_j$'s available time, $a_i\}$.

- The Minimum Completion Time (MCT) of a task T_i on a processor P_j, MCT_{ij}, is defined as the earliest time by which T_i will complete its execution on P_j.

4.3 Proposed Value-based Schedulers

In this section, we present adaptive value-based schedulers for both homogeneous and heterogeneous computing systems. The optimal scheduling of real-time tasks, maximizing the overall value of the system subject to meeting of their deadlines in a multiprocessor system is NP-complete [10]. In this section, we propose a branch and bound search algorithm, similar to myopic scheduling algorithm, but which can adapt its scheduling behavior based on the system load conditions.

The adaptive nature of scheduling is introduced in the branch function, which is used to extend the schedule (search tree). The principle idea of this scheduling is to monitor the performance of the system periodically, for each scheduling of a set of tasks (we call this as a scheduling epoch), in terms of (i) the value of tasks that were rejected as compared to the value of tasks that were accepted and (ii) total processor utilization. Thus, the system monitors and computes the *Value-Indicator* parameter, v, for every scheduling epoch as follows:

$$v = \frac{maximum\ value\ of\ task\ among\ all\ the\ tasks\ rejected}{minimum\ value\ of\ task\ among\ all\ the\ tasks\ accepted} \tag{4.1}$$

Then, we define a value weight function $F(s)$ for every scheduling epoch s as follows:

$$F(s) = \begin{cases} 1, \; v \geq 1 \\ v, \; v \leq 1 \; and \; U \leq 1 \\ F(s-1), \; otherwise \end{cases} \tag{4.2}$$

The above defined value weight function, $F(s)$, is calculated for every scheduling epoch and determines the weight that must be given to the *value-based scheduling*, while $1 - F(s)$ determines the weight that must be given to *deadline scheduling*. Hence, this function is used as the branch function in the scheduler and is applied to tasks under consideration and the branch corresponding to the minimum value of the H function is selected for scheduling. The function is applied for a task T_i as follows:

$$H(T_i) = d_i * (1 - F(s)) + (\kappa / v_i) * F(s) \tag{4.3}$$

where κ is a constant that normalizes value to the same scale of deadline time.

The above branch function is dynamic in nature, wherein the first part of the equation represents the deadline scheduling behavior, while the second part determines the value-based scheduling behavior of the scheduler. Thus, change in the $F(s)$ value leads to a change in the scheduling behavior with a value of 1 meaning that the tasks are scheduled just based on value, while a value of 0 meaning that the tasks are scheduled just based on deadline. Since, the value of $F(s)$ changes according to the load, the scheduling behavior also changes accordingly from deadline scheduling to value-based scheduling or vice versa.

4.3.1 Adaptive Value-based Scheduler for Homogeneous Systems

In this subsection, we present an adaptive scheduler for homogeneous systems, which is basically a branch and bound search algorithm using the aforesaid branch function. The objective of proposed scheduler is to search for a feasible task schedule for a given set of tasks, such that overall value of system is maximized with minimal missing of deadlines. The system consists of a task arrival queue, containing tasks that are ready to run and ordered based on

```
Adaptive Value-based Scheduler()
Input: Task set to be scheduled.
Output: Feasible schedule (maximizing the value).
  1. Tasks in the task queue are ordered in non-decreasing order of deadline.
  2. Start with an empty schedule (as initial state of search tree).
  3. Check if the initial $K$ tasks in the feasibility check region $R$ is feasible.
  4. If feasible
       (a) Compute the branch function $H(.)$ (as in equation 3) for all tasks in $R$.
           $H(T_i) = d_i * (1 - F(s)) + (\kappa / v_i) * F(s)$
       (b) Extend the task schedule with lowest branch function value on next available processor.
     else
       (a) Backtrack to the previous schedule (node), if number of backtracks are less than the maximum limit.
       (b) Extend the schedule with next best task.
  5. Repeat steps 3-5 until the task queue is empty or all the tasks are either accepted or rejected.
```

Figure 4.1 Adaptive Value-based Scheduling for homogeneous systems

their deadline. The scheduler examines the first K tasks in queue (which we call feasibility check region, R) and examines to see if the tasks in R are feasible. If feasible, then the scheduler applies the above branch function for each task to find the best task to extend the schedule (search tree). i.e., the task with minimum branch function value is selected and the scheduler repeats the same process of task selection until the task queue is empty. If feasibility check fails, then the scheduler backtracks and extends the schedule with next best task. If the number of backtrack exceeds the maximum backtrack limit then the scheduler drops the task with least value and begins the search again.

The adaptive scheduler for homogeneous multiprocessor system is presented in Figure 4.3.1. The scheduler is of complexity $O(n)$, as the deadline ordering of tasks in step 1 will be linear (as tasks can be inserted into the task queue itself based on deadline in linear time), feasibility checking of tasks in step 3 is of $O(K)$ and task selection in step 4 is also of complexity $O(K)$.

4.3.2 Adaptive Value-based Scheduler for HC systems

The scheduler presented in Figure 4.1 can adapt to different workloads dynamically but is suited only for homogeneous multiprocessor system. A direct application of the scheduler in an heterogeneous multiprocessor or multi computer environment is bound to perform poorly, as the task selected for scheduling need not always run best on the next available processor. For example, a task selected for scheduling, if scheduled on the next available processor might miss its deadline or can lead to poor schedulability (as scheduling this task on a processor with a large MCT might force the future tasks to miss their deadline, thus affecting the overall schedulability of the system).

Thus, real-time scheduling for heterogeneous computing systems involves two important steps: (1) *task selection* - selecting a task that needs to be run first and (2) *processor selection* - selecting a processor in which the selected task needs to be run. Thus, the scheduler not only determines the task to run next but also the best processor for that task to run, such that the task meets its deadline. Hence, the branch function used in the scheduler for HC systems needs to be modified to account for the heterogeneity in computing and also to carry out both task and processor selection. For this scheduling problem, we propose two different schedulers, which primarily differ by their nature of task and processor selection. The first scheduler for HC system, Basic scheduler, employs separate functions for task and processor selection. The second proposed scheduler for HC system, Integrated Scheduler, also employs a branch and bound search algorithm and uses a branch function that integrates both task selection and processor selection.

4.3.2.1 Basic Adaptive Value-based Scheduler for HC systems

In this subsection, we propose a basic adaptive scheduler for HC systems, wherein the task selection and processor selection are done in two different steps using separate functions. The *task selection* is done using the following branch function:

$$H(T_i) = d_i * (1 - F(s)) + (\kappa / v_i) * F(s) \qquad (4.4)$$

A task T_i with minimum function value will be selected for extending the schedule. Then, the scheduler selects the best processor for the selected task T_i (*processor selection*) by finding the processor P_j that offers the least MCT_{ij}. Thus, the scheduler works by extending the task with best task and finding the best available processor for the selected task. The algorithm for this scheduler is given in Figure 4.2. The complexity of this algorithm is $O(nm)$ as the feasibility checking of R in step 3 is of $O(Km)$, even though the task and processor selection is of $O(K+m)$.

```
Basic Adaptive Value-based HC-Scheduler()
Input: Task set to be scheduled.
Output: Feasible schedule (maximizing the value).
  1. Tasks in the task queue are ordered in non-decreasing order of deadline.
  2. Start with an empty schedule (as initial state of search tree).
  3. Check if the initial $K$ tasks in the feasibility check region $R$ is feasible, at least in a single
     processor/machine.
  4. If feasible
       (a) Task Selection : Compute the branch function $H(.)$ (as in equation 5) for all tasks in $R$.
           $H(T_i) = d_i * (1 - F(s)) + (\kappa/v_i) * F(\dot{s})$
       (b) Select the task with the lowest branch function and call it $T_i$.
       (c) Processor Selection : Find the processor $P_j$ that has the least minimum completion
           time value for $T_i$. i.e., let $MCT_{ij} = min_{k=1}^{m} MCT_{ik}$.
       (d) Extend the task scheduler with task $T_i$ on processor $P_j$.
     else
       (a) Backtrack to the previous schedule (node), if number of backtracks are less than the maxi-
           mum limit.
       (b) Extend the schedule with next best task.
  5. Repeat steps 3-5 until the task queue is empty or all the tasks are either accepted or rejected.
```

Figure 4.2 Basic Adaptive Value-based Scheduling for HC systems

4.3.2.2 Integrated Adaptive Value-based Scheduler for HC systems

In this subsection, we propose an integrated scheduler that uses an integrated branch function for selecting both tasks and processors at the same time, unlike the previous scheduler which uses two separate functions for the same. The branch function used for the proposed scheduler that will be computed for a task T_i on a processor P_j is as follows:

$$H(T_{ij}) = (d_i - MCT_{ij}) * (1 - F(s)) + (\kappa/v_i) * F(s) \tag{4.5}$$

Thus, the scheduler extends the task schedule with the task T_i on processor P_j that has the lowest branch function value in the feasibility check region. The integrated adaptive value-based scheduler for HC system is presented in Figure 4.3. The proposed integrated heuristic will select a task that can run on a processor that will meet its deadline at the latest, thus the tasks are not always scheduled in the fastest running processor but the processor that is just good enough to meet the deadline of a task. Since, the task and processor selection has a complexity of $O(Km)$ (like feasibility checking in step 3), the complexity of the proposed scheduler is also of $O(nm)$, but this scheduler incurs an additional run-time complexity as the process of task and processor selection is more complex $(O(Km))$ than the basic scheduler $(O(K+m))$.

Integrated Adaptive Value-based HC Scheduler()
Input: Task set to be scheduled.
Output: Feasible schedule (maximizing the value).

```
1. Tasks in the task queue are ordered in non-decreasing order of deadline.
2. Start with an empty schedule (as initial state of search tree).
3. Check if the initial $K$ tasks in the feasibility check region $R$ is feasible, atleast in a single processor/machine.
4. If feasible
      (a) Compute the branch function $H(.)$ (as in equation 4) for all tasks in $R$ for all processors.
          Compute the following function for each task $T_i$ for each processor $P_j$:
          $H(T_{ij}) = (d_i - MCT_{ij}) * (1 - F(s)) + (\kappa/v_i) * F(s)$
      (b) Find the task-processor combination yielding the lowest heuristic value ($H(T_{ij})$ and extend
          the task schedule (search tree) with task $T_i$ on processor $P_j$.
   else
      (a) Backtrack to the previous schedule (node), if number of backtracks are less than the maximum limit.
      (b) Extend the schedule with next best task.
5. Repeat steps 3-5 until the task queue is empty or all the tasks are either accepted or rejected.
```

Figure 4.3 Integrated Adaptive Value-based Scheduling for HC

4.4 Performance Studies

In this section, we evaluate the performance of the proposed adaptive schedulers in homogeneous and heterogeneous multiprocessor systems for various range of workloads, i.e., from underloads to overloads.

4.4.1 Performance in Homogeneous Computing Systems

In this section, we compare and evaluate the performance of the proposed adaptive value-based scheduler against the traditional deadline scheduler, which schedules the task always based on its deadline. The goal of our simulation study is just to compare the algorithms under different workloads, rather than to provide absolute quantification of their performance under realistic workloads. We have adopted this strategy because of two reasons: (i) there is no well known (publicly available) real-time workload for homogeneous and heterogeneous systems and (ii) experimenting under a single realistic workload has a limited scope on the performance evaluation. Hence, the two schedulers are evaluated using the following experimental setup:

- Each simulation run generates 10000 tasks.
- The task inter-arrival time follows an exponential distribution with mean θ.
- A task T_i's execution time (C_i) is uniformly chosen at random between [10,50].
- The deadline of a task is assigned to be $DFACTOR * C_i$, where $DFACTOR$ is chosen at random from uniform distribution - [1, 4].
- The value of a task is chosen at random uniformly between [50,1000].
- The load of the system is characterized by $L = C/\theta$, where C is the average execution time of a task and θ is the arrival rate of tasks in the system.
- The size of the feasibility check region (K) was chosen to 6, as the schedulers yielded better performance for this value of K. (Further, the scope of performance study is not to evaluate the performance of the algorithm with respect to K but to evaluate the behavior of the algorithms under various workloads.)

- The value of $F(s)$ was recomputed after successful scheduling of 50 tasks, comprising a scheduling epoch.

In the above experimental setup, the two schedulers are evaluated on the following metrics:

- **Success Ratio (SR)**: This is defined as the ratio of the number of tasks that were scheduled to the total number of tasks that arrived in the system.

$$SR = \frac{No.\ of\ tasks\ scheduled}{Total\ No.\ of\ tasks\ arrived} \tag{4.6}$$

- **Value Ratio (VR)**: This is defined as the ratio of the value obtained by the scheduler (i.e., the sum of value of scheduled tasks) to the the total value of all the tasks.

$$VR = \frac{Sum\ of\ value\ of\ scheduled\ tasks}{Total\ value\ of\ tasks\ arrived} \tag{4.7}$$

In the experiment setup described as above, the two schedulers, the adaptive value-based scheduler and the deadline-based scheduler were evaluated for 2 and 4 (homogeneous) processor systems for loads from 0.5 to 3. The results of the experiments evaluated based on SR and VR is given in Figures 4.4 and 4.5. As it can be seen from the figures, the proposed adaptive value-based scheduler performs well in terms of value-ratio during overloads and maintains a high value-ratio than its deadline-based counterpart, which in spite of having a better SR has a poor VR. It is to be noted that during overloads the scheduler must maintain a high VR as scheduling tasks with higher value allows the system to degrade gracefully. Thus, the proposed adaptive scheduler performs better during overloads allowing the system to gracefully degrade by maintaining high system value. Further, the performance of the two schedulers during underloads and near full loads are also comparable with respect to SR and both maintain a comparable VR. This validates our claim that the proposed adaptive scheduler performs well not only during overloads by maintaining a high system value but also minimizes deadline misses and maintains a high SR during underloads. This is due to the fact that the proposed adaptive scheduler, adapts to the workload and switches to deadline-based scheduling during underloads thereby maintaining a high scheduling success ratio and does value-based scheduling during overloads yielding an high system value.

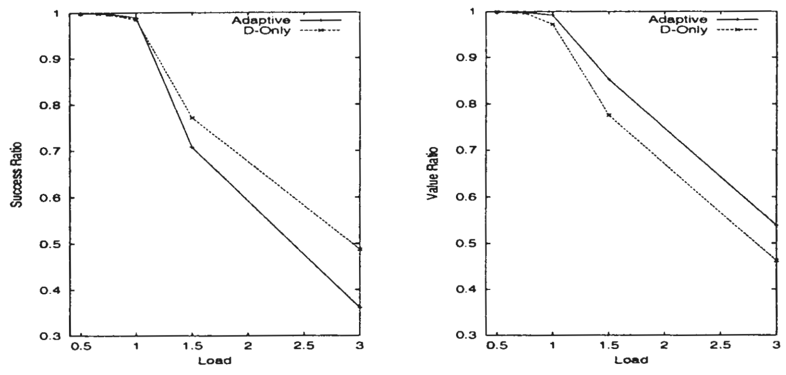

Figure 4.4 Performance of the schedulers in a 2-processor system

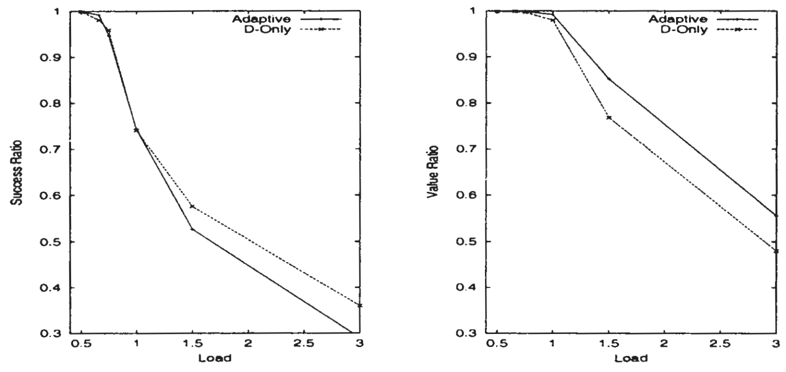

Figure 4.5 Performance of the schedulers in a 4-processor system

4.4.2 Performance in Heterogeneous Computing Systems

In this subsection, we study the performance of the proposed two adaptive schedulers for heterogeneous computing systems, which are variants of the scheduler evaluated in the previous subsection. The simulation of heterogeneous computing environment is not a straightforward process and involves simulation of heterogeneity in task and machines.

Task Generation: In this experiment, the task generation was done in a different manner wherein the task generation was done by computing an ETC (Expected Time to Compute) matrix, which consists of the computation times of tasks for all tasks on different processors. In an ETC matrix, the numbers along the row indicate the execution times of the corresponding task on different machines. The average variation along the rows is defined by task heterogeneity. Similarly, the average variation along the columns is defined by machine heterogeneity. One classification of heterogeneity is to divide into high and low heterogeneity. Based on the above idea, four categories of ETC matrix are possible: (1) high task heterogeneity and high machine heterogeneity (HiHi), (2) high task heterogeneity and low machine heterogeneity (HiLow), (3) low task heterogeneity and high machine heterogeneity (LowHi) and (4) low task heterogeneity and low machine heterogeneity (LowLow). Thus for the simulation random ETC matrices (for T rows and M columns are simulated in the following manner:

1. Let Δ_T be an arbitrary constant defining task heterogeneity, smaller denoting low task heterogeneity. Let N_t be a number picked from the uniform random distribution (1,Δ_T).
2. Let Δ_m be an arbitrary constant defining machine heterogeneity, smaller denoting low machine heterogeneity. Let N_m be a number picked from the uniform random distribution (1,Δ_m).
3. Sample N_t T times to get a vector q [0 ... [T-1]].
4. Generate the ETC matrix, e[(0...T-1),(0..M-1)] as follows:

```
(a) for $T_i$ from 0 to $T-1$
(b)         for $m_j$ from 0 to $M-1$
(c)                 pick a new value for $N_m$
(d)                 $e[i,j] = q[i] * N_m$
(e)         endfor
(f) endfor
```

Experiment Setup: In the experiments described here, the values for Δ_T for low and high task heterogeneities are 100 and 300, while the values for Δ_m for low and high machine heterogeneities are chosen to be 10 and 100. These values are selected based on the heterogeneous environment described in [20]. Further, the experiments were studied for HiHi and LowHi heterogeneous environments as only these environments capture the heterogeneity in computing power. Thus, the values generated in the ETC matrix are used as computation times of the tasks to be scheduled by the schedulers and were tested in the following simulation setup:

- Each simulation run generates 10000 tasks and were run for a 10 processor system.
- The task inter-arrival time follows an exponential distribution with mean θ.
- A task T_i's execution time vector $(\widehat{C_i})$ is obtained by generating a random ETC matrix using the algorithm described above.
- The deadline of a task is assigned to be $DFACTOR * max(C_{i1}, ..., C_{im})$, where $DFACTOR$ is chosen to be 1 to 2 with equal probability.
- The value of a task is chosen at random uniformly between [50,1000].
- The load of the system is characterized by $L = C/\theta$, where C is the average execution time of a task and θ is the arrival rate of tasks in the system.
- The size of the feasibility check region (K) was chosen to 6.

Simulation Results: The proposed two schedulers' performance are evaluated in the above simulation setup for various loads (L) for both HiHi and LowHi heterogeneous environments and the results are presented in Figures 4.6 and 4.7. It can be seen from Figure 4.6 that in a HiHi heterogeneous environment under high loads the basic scheduler performs better in terms of scheduling success ratio (SR) than integrated scheduler (for L=5 to 1.5). This is because the basic scheduler selects the more urgent task using a separate heuristic and then schedules it in a best processor whereas the integrated scheduler schedules the task only in processors that can finish as close to their deadline as possible. Further, it must be noted here that the load L is not an exact depiction of load (as even load $L = 2$ has a good scheduling success ratio), as the deadline selected for each task is twice the maximum of its computation times in all machines. Further, since the environment is highly heterogeneous the computation times can differ by a multiple of 100 times. Hence, even load values greater than 1 yields high success ratio. In such a study, it can be seen that the basic scheduler performs better under heavy loads in terms of SR and VR, whereas under lighter loads (0.5 to 1), integrated scheduler performs a little better than its basic counterpart. This is due to the fact that the integrated scheduler tries to schedule tasks that can just meet its deadline thereby under near full loads, it finds the exact fit for each task, thereby increasing the schedulability whereas the basic scheduler always schedules in the fastest processor.

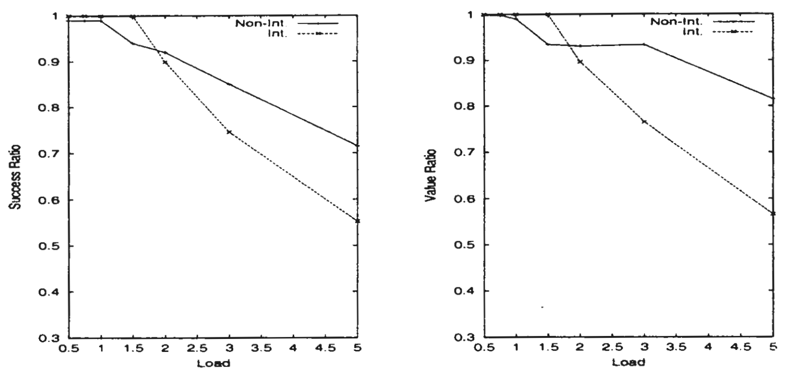

Figure 4.6 Performance in a HiHi heterogeneous computing system

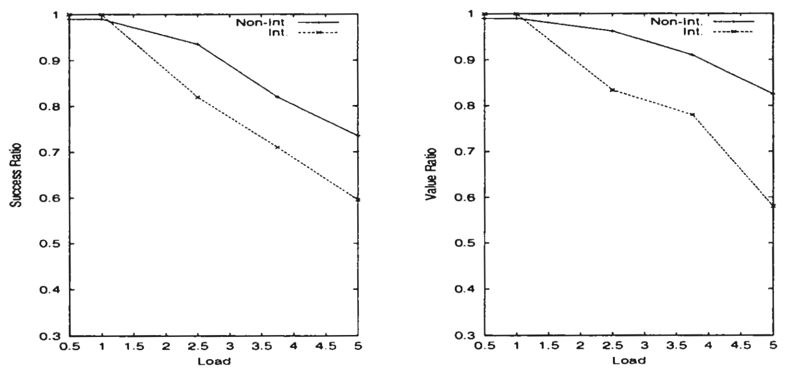

Figure 4.7 Performance in a LowHi heterogeneous computing system

The performance of the schedulers in a LowHi heterogeneous environment is given in Figure 4.7. The relative performance of the two schedulers is the same as in HiHi environment, with the basic scheduler performing better than the integrated scheduler in terms of SR and VR during heavy loads while the latter performs marginally better than the former during light loads. However, it must be noted that the performance of the integrated scheduler is better in a LowHi environment than in a HiHi environment as the heterogeneity in task computation times are less compared to HiHi environment and hence deadlines are more comparable to the least computation time of a task, whereas in a HiHi environment the laxity tends to be always high due to the high deadline estimates used in the simulation.

Thus, among the two proposed heterogeneous schedulers it can be concluded that the basic scheduler is better than the integrated scheduler as it performs better in terms of success ratio and also value ratio, and further has a marginally less run-time complexity.

CHAPTER 5. Implementation of Adaptive Value-based Scheduler in RT-Linux

5.1 Introduction

In this chapter, we present the design and implementation of the proposed adaptive value-based scheduler (for homogeneous computing system) in RT-Linux, a real-time variant of Linux operating systems and present the results of experimental evaluation of the implemented scheduler for various ranges of workloads. A shorter version of this work can be found in [32]. The rest of the chapter is organized as follows: In section 5.2, we present the basic architecture of RT-Linux and discuss its current scheduling behavior. In section 5.3, we discuss the implementation of proposed adaptive value-based scheduler and Highest Value-Density First (HVDF) scheduler in RT-Linux and present the results of experimental evaluations in RT-Linux in section 5.4.

5.2 RT-Linux

RT-Linux is a real-time variant of Linux, which adopts the approach of making Linux run as a low-priority task of a real-time executive. RT-Linux decouples the mechanisms of the real-time kernel from the mechanisms of the general-purpose kernel so that each can be optimized independently and so that the real-time kernel can be kept small and simple. One of the key features of the RT-Linux is that the real-time kernel never has to wait for the Linux side to release any resources. RT-Linux does not request memory, share spin locks, or synchronize any data structure. However, it can interact with non-real-time system using shared memory and device interface. Another key-feature of RT-Linux is that the non-real-time kernel takes

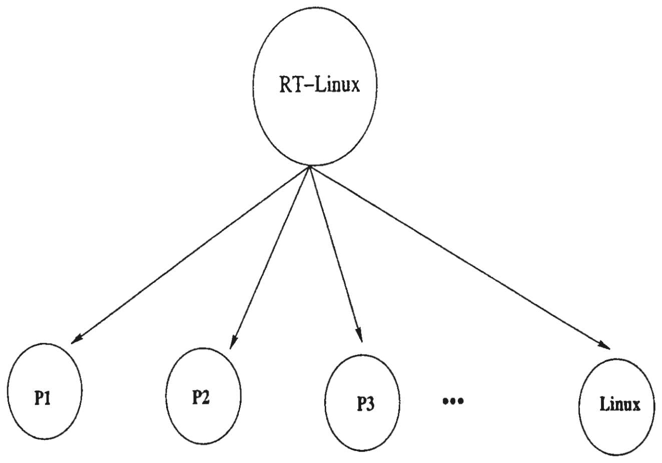

Figure 5.1 RT-Linux Architecture

care of system and device initialization, as there can be no hard real-time constraints while booting. The job of the real-time kernel is to provide the direct access to the raw hardware for real-time tasks so that they can have minimum latency and maximum processing capacity is available to them. The architecture of RT-Linux is given in Figure 5.1.

RT-Linux is module-oriented and relies on the Linux loadable kernel mechanism to install components of the real-time system and to keep the RT-system modular and extensible. The key modules of the system are the scheduler and the one that implements RT-FIFOs. The use of modules ensures the ease of changing policies for scheduling RT tasks, if the deadline requirements are not met. Currently, RT-Linux supports only periodic tasks and has two in-built schedulers - RMS and EDF implemented in it.

5.2.1 Task Creation and Scheduling in RT-Linux

Real-time tasks are created in RT-Linux using real-time thread library *pthreads*. The pthread's (task's) attributes are stored in a task structure, *rtl_thread_struct*. The important variables of the real-time task structure (rtl_thread_struct) are given as follows:

- *period* - Task Period variable - set during invocation by the task module
- *priority* - Task Priority variable - set during invocation

- *current_deadline* - Task Deadline variable - set at run-time after every period
- *resume_time* - Task Resumption Time variable - set by scheduler at run-time usually as multiples of task period

The kernel maintains a list of tasks that are ready to execute in the system in a global linked list, *task_list* and APIs such as *pthread_create(), pthread_delete() and pthread_set_schedparams()* are used for task creation, deletion and setting up of task parameters respectively. Thus, a real-time developer needs to use the above APIs to write a program that creates periodic real-time thread with desired timing characteristics.

Scheduler in RT-Linux follows a run-time system model and constructs schedule at run-time without performing a schedulability check. The scheduler can be invoked due to various reasons: such as creation of a new task, completion of current running task etc. Upon invocation, the scheduler (*rtl_schedule()*) selects the best task in the list among the tasks that are ready (task with resume_time > current_time), based on priority/period/deadline and switches the execution to the selected task. The priority-based scheduler selects task with highest priority, while RMS and EDF schedulers schedule tasks with lowest period and deadline respectively. In case of overload, the deadlines of few tasks will be missed and it can be observed when:

$$resume\_time \;<\; current\_system\_time - period$$

Hence the *resume_time* is increased by multiples of *period* until the above condition is invalid (i.e., resume_time > current_system_time - period). The number of times *period* is added to *resume_time* indicates the number of instances of the task missing the deadline. The complexity of the existing scheduler in RT-Linux is $O(n)$, as the scheduler does a linear search on the task list to select the best task.

5.3 Implementation of Value-based Schedulers

RT-Linux from the original open sources does not employ a notion of value of a task. Hence for the implementation of value-based scheduling schemes in RT-Linux, the real-time task structure (*rtl_thread_struct*) was changed and a new variable, *value*, was added. Two value-based schedulers, proposed adaptive value-based scheme and HVDF, are implemented in RT-Linux. Further as said earlier, RT-Linux does not perform schedulability check and hence the proposed scheduler is implemented without schedulability check component. The implementation details of these two schedulers are presented in the next subsections.

5.3.1 Adaptive Value-based Scheduler

For implementation of the proposed adaptive value-based scheduler (Adaptive-Impl), the primary schedule function (rtl_schedule()) was changed, wherein each task priority is computed with following heuristic function:

$$H(T_i) = d_i * (1 - FS) + (\kappa / v_i) * FS \tag{5.1}$$

The task with the minimum heuristic value is selected for scheduling. Thus, the complexity of the scheduler is still linear, i.e., $O(n)$. Further, the value of tasks that met and missed their deadlines were profiled through newly added kernel APIs (MM_met_deadline() and MM_miss_deadline()). These APIs update the maximum value of rejected tasks, MM_max_rejval variable, and minimum value of accepted tasks, MM_min_acceptval variable. These variables were used in the calculation of value-ratio for every scheduling epoch determined by MM_invocation_interval variable, which is decremented after every call to *rtl_schedule*(). Once it reaches zero, FS is recalculated and $MM\_invocation\_interval$ is reset to original feedback value. Thus, recalculation of FS is done every $MM\_invocation\_interval$ times. It must be noted that the $MM\_invocation\_interval$ variable determines the sensitivity of the scheduler to change in workload. The value of v and FS are calculated as follows:

$$v = \frac{MM\_max\_rejval}{MM\_min\_acceptval} \tag{5.2}$$

$$FS = \begin{cases} 1, \; v \geq 1 \\ FS(s-1), \; otherwise \end{cases} \tag{5.3}$$

5.3.1.1 HVDF

The implementation of this scheduler was done similar to the above scheduler, except that the task with highest value density (v_i/c_i) is selected for scheduling.

5.3.2 Validation of Implementation

The correctness of the implementation of the proposed adaptive value-based scheduler was validated by evaluating its performance for various workloads and comparing it with the simulated scheduler for the same set of workloads in terms of success ratio (SR) and value-ratio (VR). The validation method adopted is as follows: Periodic task sets were generated for different loads and the schedulers' (both implementation and simulation) performance were studied. The results of such a study is presented in Figure 5.2. It can be seen from the figure that the implemented scheduler performs close to the simulated scheduler. The difference in the schedulers performance could be due to the extra workload introduced by the Linux OS and its daemons and overheads due to actual scheduler, context switch and interrupt handling, which were not accounted in the simulation.

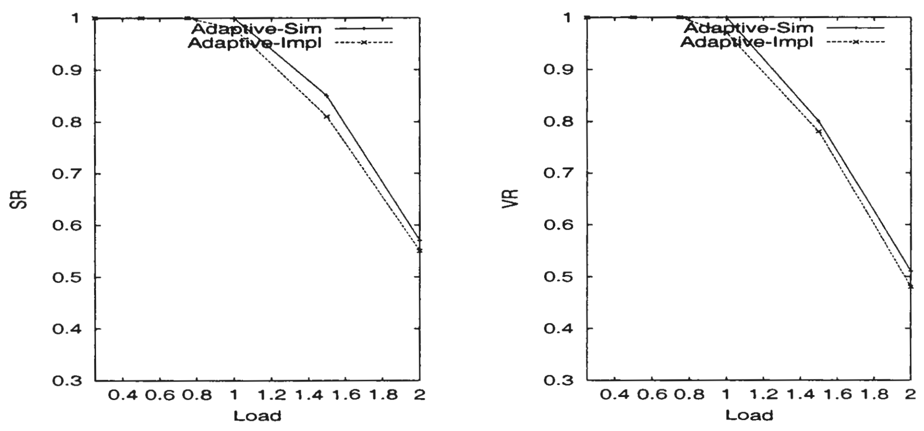

Figure 5.2 Comparison of the implementation vs. simulation (uniprocessor system)

5.4 Performance Evaluation

The performance of the three schedulers were evaluated by generating random task sets for different loads (i.e., task sets with different arrival rates). Further, it must be noted that RT-Linux supports only periodic threads and in our performance evaluation we studied aperiodic threads by creating periodic threads and running them for a single instance and deleting them upon execution. Further, kernel math libraries are not available in Linux, hence random task sets were generated by user level programs, which were used in the kernel modules. The experimental setup used for performance evaluation is as follows:

- Each simulation run generates 1000 threads.
- The task inter-arrival time follows exponential distribution with mean θ.
- A task T_i's execution time (C_i) is uniformly chosen at random between [100,5000] milliseconds.
- A task T_i's deadline is chosen to be four times C_i.
- The value of task is chosen at random between [1000,50000].
- The value of *MM_invocation_interval* was set to 20.
- The load of the system is characterized by $L = C/\theta$, where C is the average execution time of a task and θ is the arrival rate of tasks in the system.
- Each evaluation was run for generation of 1000 tasks and for 20 runs.

The performance evaluation was conducted in a Pentium-II 266 MHz machine with RT-Linux running on it. The performance of three schedulers in terms of success ratio (SR) and value ratio (VR) for various workloads and the results are presented in Figure 5.3. It can be seen from the figures that the proposed scheduler performs better than EDF and HVDF in terms of VR during underloads. Further, it can be seen that during near full loads and full loads, the proposed adaptive scheduler performs better than HVDF in terms of SR as it schedules tasks in terms of their deadline even during near full loads (not based on their deadlines as in HVDF). However, during overloads both HVDF and proposed adaptive schemes maintain the same VR as both switch over to value-based scheduling schemes.

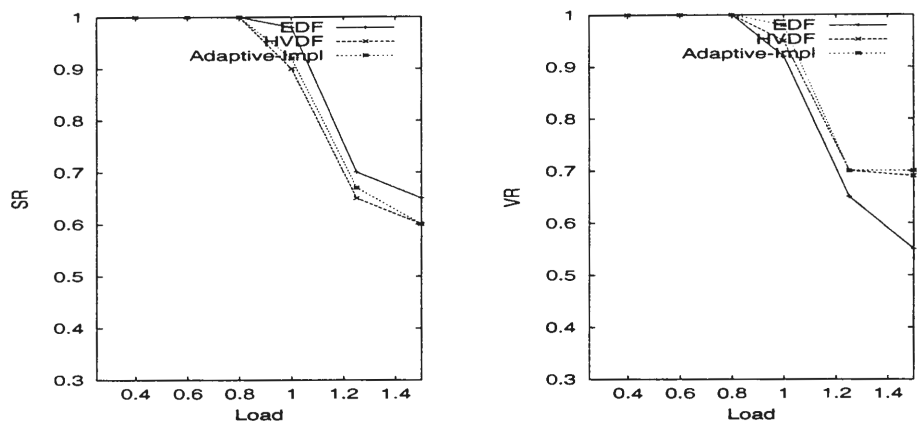

Figure 5.3 Performance of the implemented schedulers (uniprocessor system)

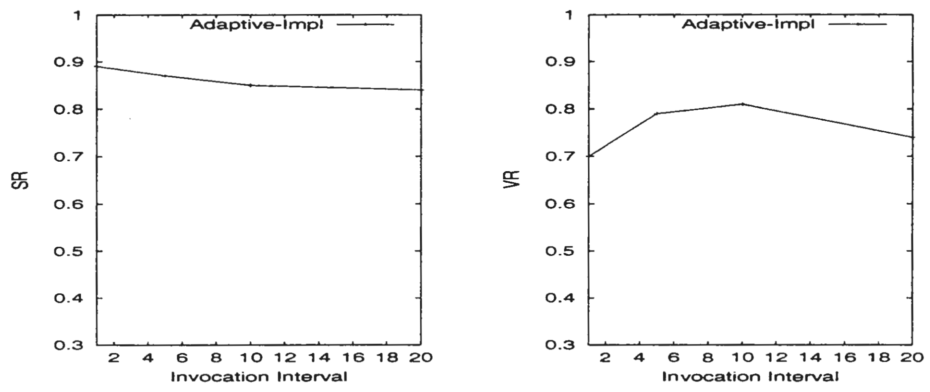

Figure 5.4 Performance of the implemented schedulers (uniprocessor system)

The feedback invocation interval (MM_invocation_interval) will also affect scheduler's performance as a high feedback rate makes the system more sensitive to transient changes in the system, increasing the run-time overheads, while a low feedback rate decreases the system's sensitivity to change in workload, yielding poor performance. The effect of invocation interval on SR and VR was studied for various values of (MM_invocation_interval) for a full (100%) load and the results are given in Figure 5.4. It can be seen from the figure that scheduler yields better result in terms of VR and SR, when FS is calculated in period of scheduling of every 10 tasks for current experiment setup, which does not introduce too much overhead due to feedback and also does not decrease the system's sensitivity to change in workload.

CHAPTER 6. Conclusions

In this thesis, we have identified two issues in dynamic scheduling for multiprocessor real-time systems and used value-based scheduling techniques to address these issues. The first issue addressed was to capture the schedulability-reliability tradeoff in real-time systems. We have proposed a dynamic value-based scheduler for multiprocessor real-time systems, which aims to maximize the overall PI of the system. The proposed scheduler has two components: Combination Selection and Order Selection algorithms. We study the effectiveness of the proposed scheduling algorithm (for various combinations of the components) through simulation studies by comparing the value obtained by scheduling a feasible task set to the value generated during its generation, by varying K and L parameters. We find that a careful selection of K and L parameter pair can yield high performance. We find the reduced search algorithm maintains a good value ratio incurring less search cost, which is important for dynamic schedulers.

The second issue addressed in this thesis is the problem of maintaining high system value with minimum deadline misses for various workloads. For this problem, we have proposed an adaptive value-based scheduler that switches its scheduling behavior from deadline-based scheduling to value-based scheduling and vice versa based on system's workload. The performance of the proposed adaptive value-based scheduler was studied using extensive simulations for various range of loads in a homogeneous computing environment and was found that the proposed scheduler maintained a high system value while maintaining high success ratio (schedulability). Further, we have proposed two adaptive value-based scheduling approaches for heterogeneous computing systems, Basic and Integrated Heterogeneous Schedulers, which differ only in their nature of processor selection. The performance of these schedulers were studied for a wide range of workloads. It was found that the basic scheduler performs better than the integrated schedulers during high loads and performs comparable to the latter during underloads and near full loads. Further, the former involves less run-time complexity than the latter and hence we conclude that the Basic scheduler can perform better as a dynamic value-based scheduler in a heterogeneous environment for wide range of workloads.

We have implemented the proposed adaptive value-based scheme in RT-Linux. We have verified the implementation by comparing its performance with the simulated scheduler for various workloads. Further, the performance of the implemented scheduler was compared with EDF and HVDF scheduler and was found that the proposed scheduler performs better than EDF in terms of value-ratio for all workloads and performs better than HVDF in terms of success ratio for near full loads. Further, the effect of feedback invocation interval was studied and was observed that the system needs to maintain a feedback invocation interval that is not too low or too high for good performance.

Future Work: The future work that can compliment the works presented in thesis can include the following: (1) Investigation of adaptive scheduling schemes that can capture the schedulability-reliability tradeoff in real-time systems, (2) Investigation of multiple task extension scheduling schemes under different real-time scheduling contexts and (3) in the context of heterogeneous computing scheduling, investigation of new integrated schedulers that can perform better than the existing integrated and basic scheduling heuristics can be a good research problem.

Bibliography

[1] K. Ramamrithnam and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. of IEEE*, vol. 82, no.1, pp.55-67, Jan. 1994.

[2] K. Ramamrithnam, J. Stankovic, and P.F. Shiah, "Efficient scheduling algorithms for multi-processor real-time systems", *IEEE Tran. Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184-194, Apr. 1990.

[3] G. Manimaran and C. Siva Ram Murthy, "An efficient and dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 312-319, Mar. 1998.

[4] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 45-61, Jan. 1973.

[5] C. Siva Ram Murthy and G. Manimaran, "Resource Management in Real-Time Systems and Networks," *MIT Press,* April 2001.

[6] Deepak R. Sahoo, S. Swaminathan, R. Al-Omari, Murti V. Salapaka, G. Manimaran, and A. K. Somani, " Feedback theory for real-time scheduling," In *Proc. of American Control Conference,* Anchorage, Alaska, 2002.

[7] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini, "The meaning and role of value in scheduling flexible real-time systems," *Journal of Systems Architecture*, 1998.

[8] S.A. Aldarmi and A. Burns, "Dynamic value-density for scheduling real-time systems," In *Proc. of 11th Euromicro Conference on Real-Time Systems*, pp. 270-277, Jun. 1999.

[9] S. Swaminathan and G. Manimaran, "A reliability-aware value-based scheduler for dynamic multiprocessor real-time systems," *In Proc. of Intl. Workshop on Parallel and Distributed Real-Time Systems*, 2002.

[10] M.R.Garey and D.S. Johnson, "Computer and intractability: A guide to theory of NP-completeness," *W.H.Freeman Company*, 1979.

[11] G. Buttazzo, M. Spuri and F. Sensini, "Value vs. deadline scheduling in overload conditions," In *Proc. of Real-Time Systems Symposium*, pp. 90-99, Dec. 1995.

[12] F.Wang, K. Ramamrithnam, and J. Stankovic, "Determining redundancy levels for fault-tolerant real-time systems," *IEEE Trans. Computers*, vol. 44, no. 2, pp. 292-301, Feb. 1995.

[13] S.Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272-284, Mar. 1997.

[14] A.K. Somani, and N.H.Vaidya, "Understanding fault-tolerance and reliability," *IEEE Computer*, vol. 30, no. 4, pp.45-50, Apr. 1997.

[15] K. Mahesh, G. Manimaran, C. Siva Ram Murthy, and A.K. Somani, "Scheduling algorithm exploiting spare capacity and task laxities for fault detection and location in real-time multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 52, no. 2, pp. 136-150, June 1998.

[16] J. Zhou, G. Manimaran, and A.K. Somani, "A dynamic scheduling algorithm for improving performance index in multiprocessor real-time systems," in *Proc. of ADCOMM*, 1999.

[17] I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Fransisco, CA, 1999.

[18] M. Maheswaran, T.D. Braun and H.J.Siegel, "Heterogeneous distributed computing," in *Encyclopedia of Electrical and Electronics Engineering*, J.G. Webster, ed., John Wiley, New York, NY, 1999.

[19] H. Topcouglu, S. Hariri and Min-You Wu, "Task scheduling algorithms for heterogeneous processors," in *Proc. of 8th Heterogeneous Computing Workshop*, 1999.

[20] M. Maheswaran, S. Ali, H.J.Siegel, D. Hensgen and R.F. Freund, "Dynamic matching and scheduling of class independent tasks onto heterogeneous computing systems," in *Proc. of 8th Heterogeneous Computing Workshop*, 1999.

[21] X. Qin and H. Jiang, "Reliability-driven scheduling for real-time tasks with precedence constraints in heterogeneous systems," in *Proc. of Intl. Conference on Parallel and Distributed Computing and Systems*, Nov. 6-9, 2000.

[22] M.L. Dertouzos and A.K.Mok, "Multiprocessor on-line scheduling of hard real-time tasks", *IEEE Software Engg.*, vol. 15, no. 12, pp. 1497-1506, Dec. 1989.

[23] Victor Yodiken, "The RT-Linux Manifesto," http://www.fsmlabs.com/developers/white_papers/.

[24] S. Swaminathan and G. Manimaran, "FARM: A feedback-based adaptive resource management for autonomous hot-spot convergence system," in *Proc. of Intl. Workshop on Parallel and Distributed Real-Time Systems*, 2002.

[25] K. G. Shin and P. Ramanathan, "Real-time Computing: A new discpline of computer science and engineering," *Proc. of IEEE Computer, vol. 82, no. 1* 6-24, Jan. 1994.

[26] C.M. Krishna, and Y.H.lee,"Guest-editors' introduction: Real-time systems," *IEEE Computer*, vol. 24, no. 5, 10-11, May 1991.

[27] D.S. Fussell, and M. Malek, "Responsive computer systems: Steps towards fault-tolerant real-time systems," *Kluwer Academic Pub.*,1995.

[28] H. Kopetz, A. Damm, C. Koza, and Mulozanni, "Distributed fault-tolerant real-time systems: The MARS approach," *IEEE Micro*, vol. 9, no. 1, 25-40, Feb. 1989.

[29] K.G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer, vol. 24, no. 5*, 25-35, May 1991.

[30] L. Chen, and A. Avizienis, "N-Version programming: A fault-tolerant approach to re-aliability of software operation," In *Proc. of Symposium on Fault Tolerant Computing (FTCS)*, 2-9, June 1978.

[31] B. Randell, "System structure for software fault-tolerance," *IEEE Trans. on Software Engineering,* 220-232, June 1975.

[32] S. Swaminathan and G. Manimaran, "An adaptive value-based scheduler and its RT-Linux implementation," accepted for publication in *High Performance Computing (HiPC)*, 2002.

ACKNOWLEDGEMENTS

I would like to thank my brother, dad and mom without whom doing this Masters would have never been a possibility. Academically, I owe a lot to my advisor, Dr. Manimaran Govindrasu, who patiently taught me how to do research and got me this far. I owe a lot to him for his interest in my research and also in encouraging me to pursue on for a Ph.D. Further, I would like to thank my program of study committee members for their reviews. I need to thank my lab mates Anirban Chakrabarti and Aaron Striegel, who helped me a lot to kick off my initial research ventures. I would like to thank my friends in ISU for making my stay in ISU a memorable experience.