Heuristic Algorithms for Adaptive Resource Management of Periodic Tasks in Soft Real-Time Distributed Systems

By

Ravi K. Devarasetty

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements of the degree of

Master of Science
In
Computer Engineering

APPROVED:

_____
Binoy Ravindran, Chairman

_____ _____
Scott F. Midkiff Pushkin Kachroo

February 6, 2001
Blacksburg, VA

Keywords: Adaptive Resource Management, Dynamic Real-time Systems, Distributed Real-time Systems, Prediction-based algorithms, Heuristic-based algorithms

Heuristic Algorithms for Adaptive Resource Management of Periodic Tasks in Soft Real-Time Distributed Systems

Ravi K. Devarasetty

Abstract

Dynamic real-time distributed systems are characterized by significant run-time uncertainties at the mission and system levels. Typically, processing and communication latencies in such systems do not have known upper bounds and event and task arrivals and failure occurrences are non-deterministically distributed. This thesis proposes adaptive resource management heuristic techniques for periodic tasks in dynamic real-time distributed systems with the (soft real-time) objective of minimizing missed deadline ratios. The proposed resource management techniques continuously monitor the application tasks at run-time for adherence to the desired real-time requirements, detects timing failures or trends for impending failures (due to workload fluctuations), and dynamically allocate resources by replicating subtasks of application tasks for load sharing. We present "predictive" resource allocation algorithms that determine the number of subtask replicas that are required for adapting the application to a given workload situation using statistical regression theory. The algorithms use regression equations that forecast subtask timeliness as a function of external load parameters such as number of sensor reports and internal resource load parameters such as CPU utilization. The regression equations are determined off-line and on-line from application profiles that are collected off-line and on-line, respectively. To evaluate the performance of the predictive algorithms, we consider algorithms that determine the number of subtask replicas using empirically determined functions. The empirical functions compute the number of replicas as a function of the rate of change in the application workload during a "window" of past task periods. We implemented the resource management algorithms as part of a middleware infrastructure and measured the performance of the algorithms using a real-time benchmark. The experimental results indicate that the predictive, regression theory-based algorithms generally produce lower missed deadline ratios than the empirical strategies under the workload conditions that were studied.

Acknowledgements

I sincerely thank my advisor, Dr. Binoy Ravindran and my committee members, Dr. Scott Midkiff and Dr. Pushkin Kachroo. I could not have asked for an advisor better than Dr. Ravindran. His constant support and guidance helped me achieve my goals on time.

I would also like to thank John Harris for helping me on several occasions with valuable advice and support.

Finally, I would like to thank my friends; without their company and support, this would not have been possible.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Real-time computer systems that perform distributed mission management such as collaborative direction within a team of autonomous entities conducting combat must accommodate significant run-time uncertainties in the application environment and system resource state. The computations in the system are predominantly asynchronous (e.g., event driven and aperiodic), data-dependent (e.g., input data arrivals cause significant changes in resource needs based on semantic content of data), and they frequently constitute an overload. Furthermore, hardware and software failures are common. Examples of distributed, asynchronous real-time systems and their applications include the emerging generation of surface combatant systems of the U.S. Navy.

Classical real-time computing research focuses on “hard” real-time computing for synchronous, device level, sampled data monitoring and regulatory control – usually centralized [Bak91, LL73, LRT92, LSS87, LW82, RCF97, RTL93, SB96, SKG91, SLS88, SSL89, TLS96, XP90], but occasionally distributed [CSR86, HS92, Kao95, Kop97, Mok83, RSZ89, Shi91, SR91, SRC85, VWHL96, WSM95]. It is difficult to practically employ or adapt such techniques for systems that are distributed and asynchronous [Jen92, Koob96, Sta96, SK97]. Asynchronous real-time computer systems and their applications are inherently *posteriori* in terms of their workload characteristics and thus require adaptive real-time resource management.

1.1 A Distributed Dynamic Real-Time System

A benchmark application developed previously [SWR99] has been used as a distributed, dynamic real-time system. This benchmark application functionally approximates the Navy’s Anti-Air Warfare (AAW) surface combatant system. The system is explained in the chapters that follow.

1.2 Resource Management Problem

Unpredictable loads at run-time characterize a dynamic, distributed real-time system, and as such the system may miss its deadlines when the loads increase. When such a system misses its deadline, the system's deadline requirements have to be satisfied as soon as possible for which there is no single approach. One approach could be to replicate the bottleneck subtasks that form a part of the system. Another could be to prioritize the message delivery between a pair of subtasks of the system. In this thesis, we look at the former approach. The resource management problem involves predicting the number of replicas of bottleneck subtask programs required so that the end-to-end deadline of the system could be satisfied as soon as possible.

1.3 Existing Solutions

Existing solutions look at the resource allocation problem in a static manner. By static manner we mean that for a given real-time system, offline analysis is made and when the system is up and running, those solutions are applied for resource allocation whenever the system misses its deadline. This method is suitable for static real-time systems where the run-time loads are known before the system is actually up and running. But, this method is not suitable in the case of a dynamic real-time system where the run-time loads are unknown hence an offline analysis is not suitable.

1.4 Potential Solutions

For solving the resource allocation problem of a dynamic real-time system, offline analysis is not suitable. For this, we need to take a dynamic approach. We need to continuously monitor the system for missed deadlines, and when a deadline is missed, we need to perform online analysis, and determine how resources have to be allocated, like how many replicas of the bottleneck subtasks are needed to satisfy the system's deadline requirement. This approach is what we call adaptive resource management, which is explained in the chapters to follow.

1.5 Previous Work

Previous work has been done on real-time scheduling and resource management. We present a simple classification scheme that classifies real-time scheduling and resource management algorithms. The classification scheme is based on a characterization of the size of data streams processed by periodic tasks. The size of the data stream of periodic tasks can be defined as either deterministic or stochastic depending upon whether the data stream size can be characterized completely *a priori* using discrete values or using probability distribution functions, respectively. Data stream size can also be defined as dynamic, if an *a priori* characterization of the stream size is impossible.

We now examine well-known real-time scheduling and resource management algorithms and classify them on the basis of the classification scheme.

Table 1 shows classification of real-time scheduling and resource management algorithms based on the size of data streams processed by periodic tasks. Observe that the size of the data stream refers to the number of data items (or sensor reports) that periodic assessment tasks and transient-periodic guidance tasks of real-time C2 (command and control) systems must process during a single execution cycle. The number of sensor reports that must be processed in a single execution cycle will significantly impact the execution times of the tasks. Thus, we classify the real-time scheduling and resource management algorithms as deterministic, stochastic, and dynamic depending upon how the algorithms model the cycle execution times of periodic and transient-periodic tasks.

Table 1. Classification of Real-Time Scheduling and Resource Management Algorithms Based on Data Stream Sizes

	Deterministic	*Stochastic*	*Dynamic*
Data stream size	[Bak91][Cla90][LL73] [RSZ89][SKG91][Ver95] [XP90][WSM95]	[AB98][KM97][Leh96] [Loc86][SK97] [SL96] [TD+95] [Kao95]	[BN+98] [RLLS97] [HSNL97][RSYJ97]

In a number of real-time scheduling algorithms, the execution time of a "job" is considered to be known completely *a priori*. Typically, execution time is assumed to be an integer "worst-case" execution time, as in [Bak91, Cla90, LL73, RSZ89, SKG91, Ver95, WSM95, XP90]. Therefore, we classify these scheduling algorithms as deterministic, as the algorithms model the number of data items processed by periodic tasks as integer, discrete values that are known *a priori*.

Paradigms that generalize execution times have also been developed. Execution time is modeled as a set of discrete values in [KM97], as an interval in [SL96], and as a probability distribution in [AB98, Leh96, Loc86, SK97, TD+95, Kao95]. Therefore, these algorithms are characterized as stochastic in the classification, since they model data stream sizes using distribution functions.

The resource management models presented in [BN+98, HSNL97, RLLS97, RSYJ97] allow execution time of tasks to have unknown *a priori* characterizations. Hence, they are classified as dynamic. This work also belongs to this category. So we compare and contrast our work with these efforts.

This work fundamentally differs from [BN+98, RLLS97, HSNL97] in the application model. [BN+98, RLLS97, HSNL97] present an application model that consists of multiple applications, where each application can operate at multiple discrete "levels." A level is a strategy for doing the application's work and is characterized by a benefit and resource usage (e.g., CPU utilization). The benefit of an application at a given level is defined as a function of the resource usage of the application at that level and is (explicitly) specified by the user. The authors present resource allocation algorithms that dynamically select or negotiate the levels of applications, or enable the applications to select their levels, such that the total benefit is maximized and each individual application operates without missing its deadline. In contrast, our application model consists of applications, where each application operates at a single level of benefit (i.e., satisfaction of its deadline) and an implicit resource usage corresponding to that benefit. Further, we present resource management algorithms that dynamically allocate resources to an

application when the application exhibits trends for timing failures due to increase in workloads. The resource management algorithms automatically allocate the appropriate resources by determining the "right" programs of the application to replicate or "right" programs to migrate and "right" processor resources for the replica or migrant programs such that the application timing constraint is satisfied.

The approach described in [RSYJ97] is similar to ours in (1) the application model and (2) the way resource management is performed, i.e., QoS monitoring followed by QoS diagnosis and resource allocation by replication or migration to improve QoS. However, there are fundamental differences. The goal of [RSYJ97] is to discover the factors that influence the effectiveness of run-time resource management that achieve the timeliness requirements in dynamic real-time systems. The authors identify several factors such as early detection, overhead of enactment of the reallocation decision, and application state-driven incremental decision heuristics that contribute to the effectiveness of run-time resource management through an experimental study. This work fundamentally differs from [RSYJ97] in the objective. Our goal is to discover resource management techniques—monitoring and detection, diagnosis, and resource allocation strategies—that will achieve the timeliness requirements during high external load situations. Therefore, we are interested in discovering algorithms that answer questions such as (1) how can a low timeliness situation be detected, (2) how can we recover from a low timeliness situation and what recovery actions are needed, (3) what are the resources that must be allocated to recovery actions so that the actions will improve timeliness, and (4) what are the relative merits of algorithms that allocate resources based on availability and those that allocate resources by forecasting application timeliness. This thesis presents algorithms that answer the above questions and, thus, the work differs from [RSYJ97] in the objectives. However, the two objectives are complimentary and provide insight into important considerations for engineering dynamic real-time systems when viewed collectively.

1.6 Organization of the Thesis

Chapter 2 describes a generic real-time system, and how the benchmark used in the experiments for this thesis was evolved from such a system. Chapter 3 explains the adaptive resource management process and the various steps involved in it. Chapter 4 presents the benchmark application and the performance evaluation metrics of the algorithms, which are of prime interest and calculated from the benchmark. Chapter 5 discusses the resource allocation algorithms, mainly divided into Class A, Class B, and Class C algorithms. It also deals with the scalability issue (Class H Algorithms) that deals with resource allocation when several tasks are running simultaneously in the system. Chapter 6 presents the concept of automatic profiling where the non-linear multiple regression equations used for prediction of replicas of subtasks can be computed online. Chapter 7 presents the experimental results and the related observations. Chapter 8 presents the conclusions drawn from the experimental results.

Chapter 2 A Generic Real-Time System

Figure 1 shows a generic real-time system and its environment. The real-time system consists of tasks that perform *assessment* of the environment, *initiation* of actions, and monitoring and *guidance* of the actions to their successful completion. The inter-relationship of the tasks with the environment and the intra-relationship of the tasks among themselves are illustrated in Figure 1.

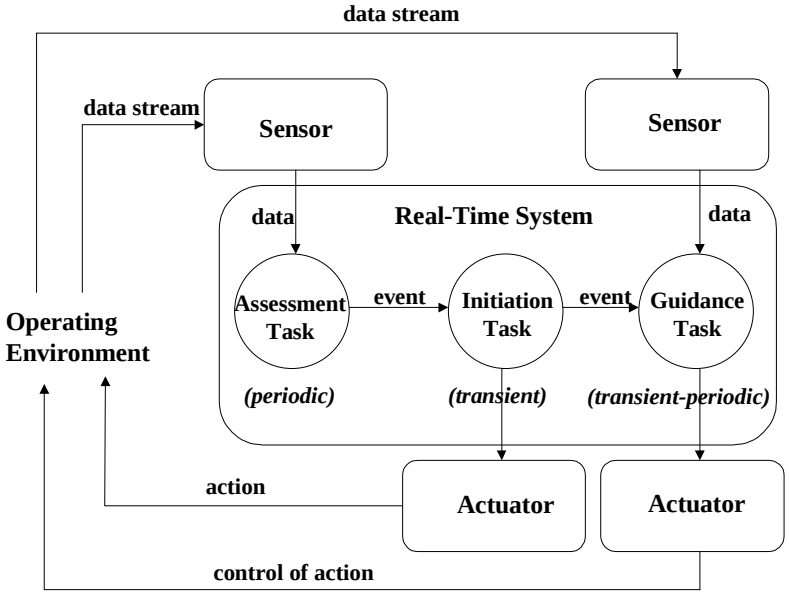

Figure 1. A generic real-time system

The assessment task repeatedly collects data from the environment through hardware sensing devices. The data is filtered, correlated, classified, and then used to determine the necessity of an action by the system. The assessment task has a *periodic* behavior since the data collection and assessment is performed repeatedly throughout the operational life of the system. Typically, there is a temporal bound associated with the completion of each execution cycle of the task, i.e., a bound on the time taken to review each of the elements of the data stream once. When an action is necessary, the assessment task generates an event that activates the initiation task.

The initiation task determines the action that needs to be taken and causes actuators to perform the action. Since the task executes in response to an event that can occur at any time, the initiation task has a *transient* behavior. The initiation task has, typically, a maximum latency requirement on its execution time, i.e., a latency requirement on the execution time of the task in response to a single event. Upon initiation of the action by the actuators, the guidance task is notified.

The guidance task repeatedly uses sensors to collect data, to monitor the actions that were initiated, and to guide the actuators to successful completion of the actions. Since the activation of the guidance task can begin and terminate at any time, the task is transient in behavior. Further, the task behaves like an assessment task once it is active, executing in a periodic manner. Hence, the guidance task has a *transient-periodic* behavior. Observe that each (transient) activation of the task consist of a set of execution cycles. Guidance tasks typically have two temporal bounds: (1) completion time for each cycle and (2) de-activation time (i.e., an activation deadline). Typically, it is more critical to perform the required processing before the activation deadline than it is to meet the completion time for each cycle.

After a careful study of the AAW real-time system, we have observed that the resource needs of the tasks are significantly influenced by the size of the data and the event streams. Size of the data stream refers to the number of data items (sensor reports) that the assessment and guidance tasks have to process during a single execution cycle, and size of the event stream refers to the arrival rate of events that trigger the execution of the initiation and guidance tasks. For systems such as the AAW, data stream sizes (radar tracks) and event (threat) arrivals have neither known upper bounds, nor deterministic distributions. Thus, asynchronous real-time systems are real-time systems that have:

(1) *unknown* upper bounds for the size of data streams processed by periodic and transient-periodic tasks during a single execution cycle and

(2) *non-deterministic* distributions for the arrival rates of events that trigger the execution of transient and transient-periodic tasks.

Observe that the generic model of real-time systems that we have presented here does not capture its distributed nature. This is because the goal of presenting this model is to reason about the load characteristics (both internal and external) of the system and to define what we mean by an asynchronous real-time system. Distribution is an orthogonal issue. The fact that a real-time system is asynchronous need not imply that it is distributed. However, often there exists an application "pull" for the physical distribution of asynchronous real-time systems that is both involuntary and voluntary.

The most common involuntary motivation for distribution is that the assets of the application (e.g., the different processing stages of a manufacturing plant, radars and missile launching devices of a combat system, the ships and aircraft of a battle group) are inherently dispersed [CJR92]. Furthermore, real-time (response time) requirements of individual components of such systems often cannot be met with a centralized computing facility.

A primary voluntary reason for physical distribution is survivability, in the sense of continued availability with a degradation of functionality or performance. Often, it may be cost-effective to distribute a combat system than it is to implement a centralized one that becomes a single point of failure.

Chapter 3 Adaptive Resource Management

3.1 Middleware Architecture

The software architecture of the resource management middleware is shown in Figure 2. This architecture is a simplified version and has been derived from [RWS99]. The core components of the middleware include a *resource manage*r and a *system daemon*. These two components have been developed as a part of this research.

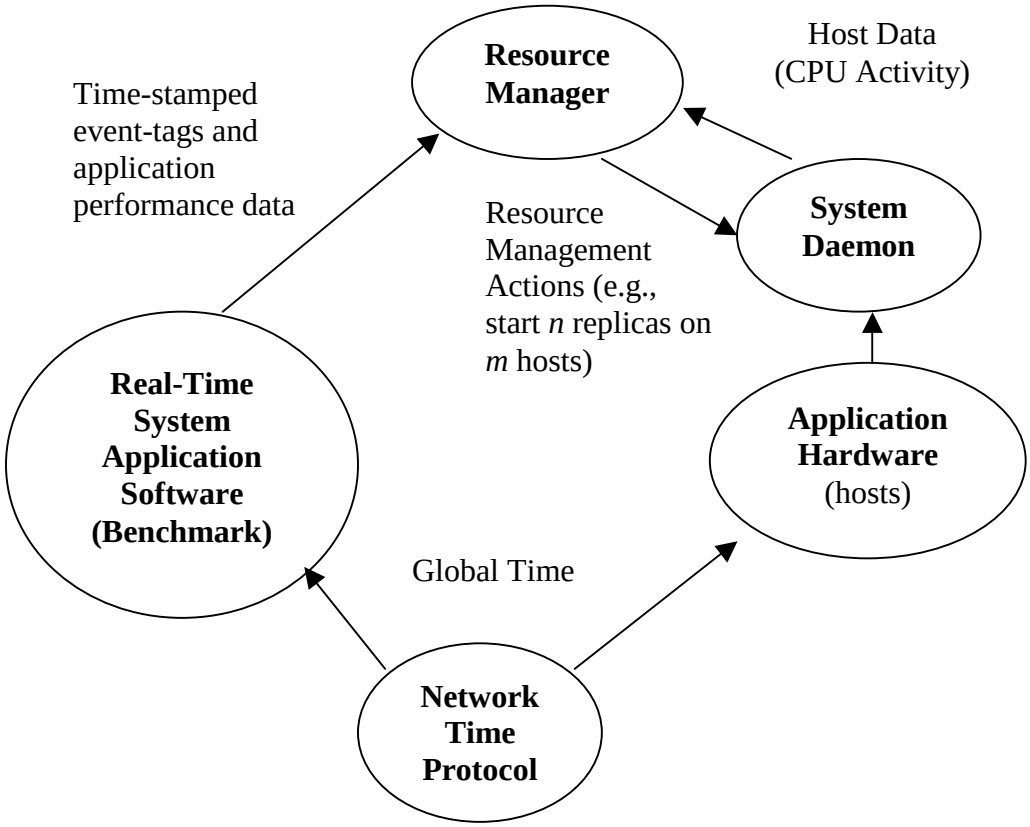

Figure 2. Adaptive Resource Management Middleware Architecture

The resource manager performs several tasks. It is responsible for collecting and maintaining all application information. Dynamically measured performance metrics of the application are collected and maintained by the resource manager. Performance metrics of hardware resources like the CPU and memory utilizations of hosts are

collected and reported to the resource manager by the system daemons. There is one system daemon per host machine. The metrics are transmitted to the resource manager periodically. The system daemon uses the 'vmstat' command of Linux (also available in general flavors of Unix) and calculates the CPU and memory utilization by parsing the data. Similar system calls can be used on other operating systems to evaluate the host metrics.

The resource manager keeps track of the degrading timeliness situations (e.g., when the task latency exceeds its deadline or exceeds an acceptable slack bound on the deadline). It receives time-stamped event tags from application programs, transforms them into task latencies, and compares the latencies against task deadlines for detecting degrading timeliness situations.[1] When a task exhibits low timeliness, the resource manager detects the situation and performs diagnosis to determine components of the task that are exhibiting a low timeliness. An example of a diagnosis is to determine the set of application programs that are experiencing an increased execution or communication latency due to greater contention for resources from other application programs or communication links. Such components may contribute to a low timeliness of the end-to-end task.

When the resource manager detects the low timeliness situation of a task, it performs allocation analysis to identify possible resource allocation actions for the task that will improve the task timeliness. Typical resource allocation actions include *replication* of "bottleneck" application programs of the task that are responsible for the low timeliness of the task. The idea behind replication of application programs is that once an application program is replicated, the replicas of the program can share the data stream that was being processed by the original program. Further, concurrency can be exploited by executing the replicas on different processors and, thereby, the end-to-end latency of the task can be reduced. Thus, replication is allowed as a means to reduce task

[1] The middleware obtains global time through the use of NTP [Mills95].

latencies and satisfy the task timeliness requirements when the data steam size increases and causes task latencies to increase at run-time.

Once the actions are identified through allocation analysis, the resource manager selects resources for the actions. The objective of resource selection is to determine the optimal or sub-optimal resources (computing and communicating resources) for enacting the actions so that the actions will improve the timeliness. Example resource selection includes determining the optimal or sub-optimal host machine for executing the replica of an application program that will improve task timeliness. The resource manager uses load information of resources provided by the system daemons for resource selection. Once the resource allocation actions and resources for the actions are determined, the resource manager notifies the system daemon for enacting the actions.

The system daemons are also responsible for notifying the resource manager when failures of application programs occur. In the event of an application program failure, the system daemon on the host of the failed program alerts the resource manager.

3.2 Steps Involved

The process of adaptive resource management is illustrated in Figure 3.

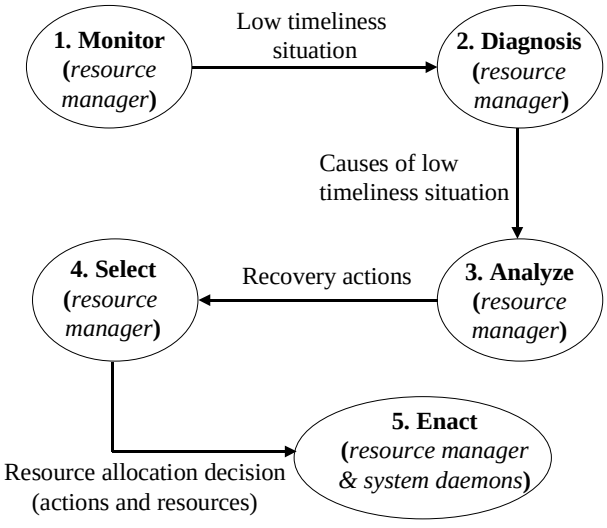

Figure 3. The Adaptive Resource Management Process

The various steps in the adaptive resource management process are explained below.

1. *Monitor:* The resource manager receives time-stamped event tags from application programs, transforms them into task latencies, and compares the latencies against task deadlines for detecting degrading timeliness situations. The resource manager also maintains the current state of the hosts, i.e. their CPU and memory utilizations through the periodic updates sent by the system daemon on each host.

2. *Diagnosis:* When a low timeliness situation arises, the resource manager takes diagnostic action. This action would find the cause of the low timeliness situation. The reason could be the high execution latency of a bottleneck subtask program of the application, or increased communication latency between a pair of subtask programs.

3. *Analyze:* After the cause is determined, say an application subtask is found to have high execution latency and, hence, causing the task to miss its deadline, analysis is performed as to how many replicas of that subtask need to be replicated so that the application task would recover from the low timeliness situation.

4. *Select:* After the number of replicas of the bottleneck subtask programs has been determined, the resource manager selects the host(s) on which those replicas are to be run. The selection of the host could be based on simple criterion such as the host with the least CPU utilization, least memory utilization, etc.

5. *Enact:* Finally, after the decision is made on the subtask programs and the hosts, the resource manager sends commands to the system daemons to start the required number of replica programs on the selected hosts.

The resource manager refreshes its table of hosts on a periodic basis. If it does not receive an update from a system daemon for 10 consecutive periodic cycles, it deletes the entry for that host. If again an update comes from that daemon, the resource manager updates its host table accordingly.

Chapter 4 Performance Evaluation

4.1 A Real-Time Benchmark Application

A real-time benchmark application has been developed that functionally approximates Navy's AAW surface combatant system. The benchmark uses simulated sensors and actuators. However, it employs real algorithms for performing tasks such as assessment, initiation, and guidance. Details of the benchmark can be found in [SWR99]. In this section, we only summarize the structure of the tasks that have timeliness requirements.

The software architecture of the benchmark application is shown in Figure 4. The benchmark consists of the following tasks

1. A periodic assessment task called *Sensing* that consists of the sensor (a simulator program called *Sensor*), a filter manager program (called *FilterManager*), one or more replicas of the filter program (called *Filter*), an evaluate and decide manager (called *EDManager*), and one or more replicas of an evaluate and decide program (called *ED*).

2. A transient task called *Engagement* that consists of three programs: an action manager program (called *ActionManager*), an action program (called *Action*) that is replicated, and an actuator simulator program (called *Actuator*).

3. A transient-periodic monitor and guidance task called *MonitorGuide* that includes all components of the assessment task, a monitor and guide manager program (called *MGManager*), and a monitor and guide program (called *MG*) that is replicated.

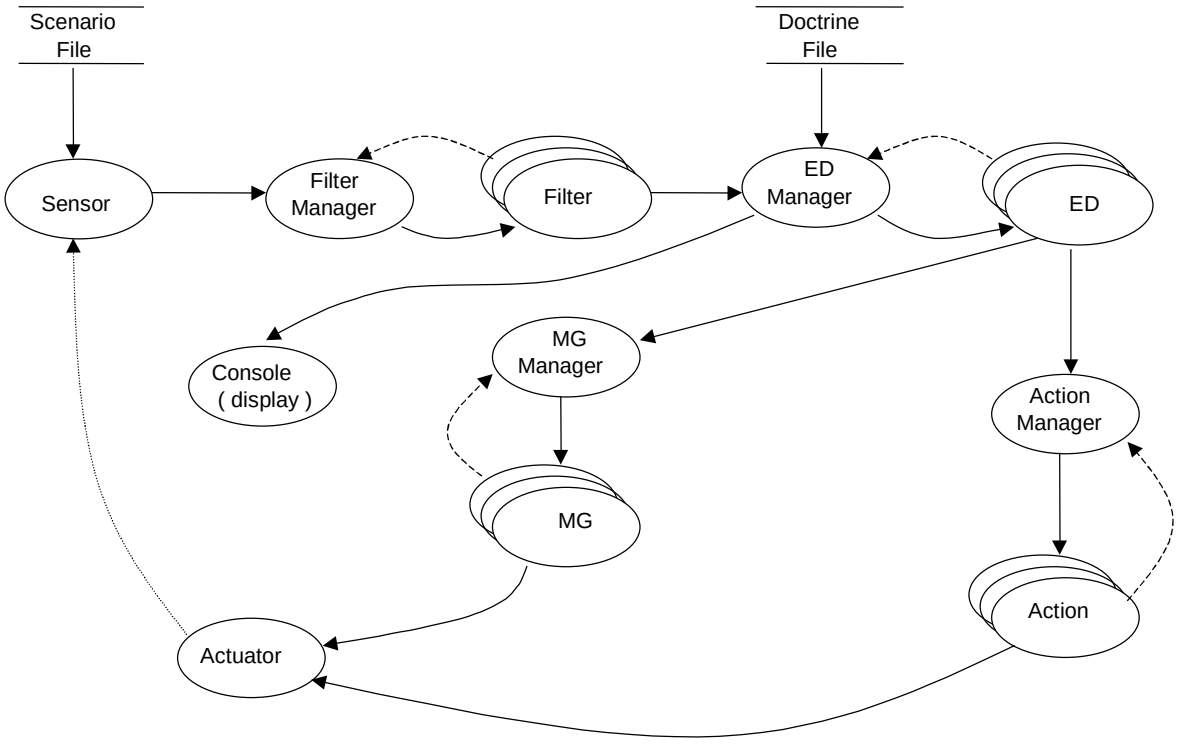

Figure 4. Software architecture of the real-time benchmark application

The timeliness requirements of the benchmark include deadlines for the completion of execution cycles of all three tasks. Further, the transient-periodic monitor and guidance task also has an activation deadline. Here, we focus on only the periodic sensing task. Note that each execution cycle of the sensing task that begins with the generation of data items by the sensor and terminates with the processing of all the data items (generated by the sensor in that cycle) by the ED program has a maximum latency requirement. This latency requirement must be adhered to in all cycles, irrespective of the number of data items generated by the sensor during any cycle.

Since this thesis deals with the resource allocation of only the periodic task or the sensing task, we describe the sensing task in detail. The sensing task further consists of the following sub-tasks

1. Sensor: The Sensor program is an application of benchmark coded as a radar sensor. It sends position samples to filter manager.

2. Filter Manager (FM): The Filter Manager program distributes the workload among its workers depending on the number of workers (Filters). Filter Manager receives the data from sensor and sends the same to filter.

3. Filter: The Filter program acts like a filter that filters the noise in the data received from the filter manager, and recognizes the tracks and their motion equations. After this is done, the track data is sent to Evaluate and Decide Manager.

4. Evaluate and Decide Manager (EDM): The Evaluate and Decide Manager program distributes the workload among its workers depending on the number of workers (Evaluate and Decide). EDM receives the data from Filter and sends the same to Evaluate and Decide. This also provides all its workers with the current evaluation rules.

5. Evaluate and Decide (ED): This program evaluates the present position of tracks by comparing it with the evaluation rules.

The operation of the sensing task is described below.

The sensor program in the benchmark generates a set of tracks per cycle analogous to one sweep of air space by a radar. A track is a packet of data that characterizes an object and goes through the entire set of programs. The corresponding programs perform mathematical processing on the data simulating the objects in a three-dimensional air space. Each cycle of tracks is identified by a cycle number.

We associate a deadline requirement with the latency of the sensing task, which is defined as the end-to-end time taken for tracks of a particular cycle to go from the sensor through the ED Program. To eliminate the impact of random effects and noise, we

differentiate between a latency miss (i.e., during a cycle, end-to-end latency is greater then the deadline) and an end-to-end violation in the following manner.

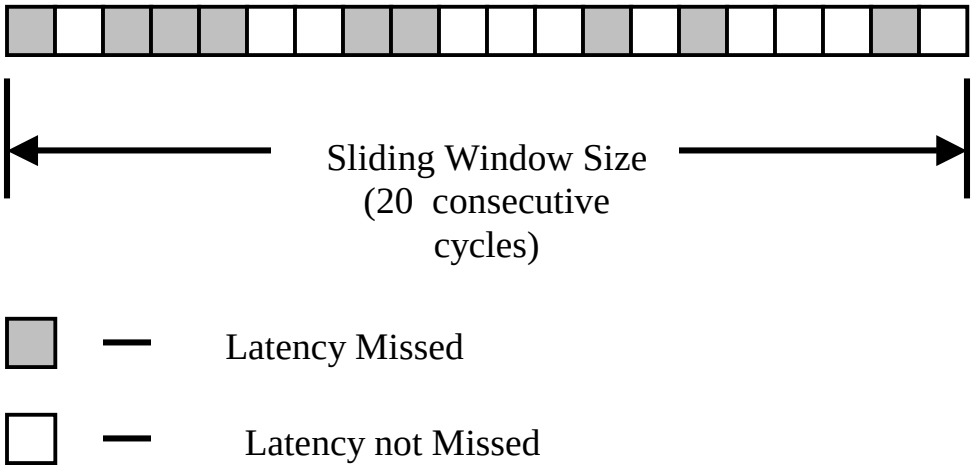

Figure 5. Eliminating the impact of noise and random effects.

The benchmark is designed in such a way that after the additional replicas come up, they automatically find their program endpoints and synchronize with the benchmark within the next few cycles.

4.2 Parameters of Interest

We evaluate the performance of the resource allocation algorithms through a set of experiments. The objective of the experiments is to measure the quality of the prediction techniques in terms of how they affect the timeliness of the tasks during overloaded situations. We measure the quality of the techniques through the following set of metrics

1. Missed Deadline Ratio (MDR): This is the ratio of the deadlines missed to the total number of deadlines that could have been satisfied by resource allocation.

2. Average Resource Utilization: The average CPU or memory utilization of all the hosts of the system throughout the experiment.

3. Steady State Latency Slack (SSLS): The difference between the deadline and the end-to-end latency after the resource manager has made an allocation.

Chapter 5 Resource Allocation Algorithms

Before we consider the resource allocation algorithms that use non-linear multiple regression equations for prediction (i.e., Class A algorithms), we show the iterative technique[2] that was used to calculate the regression equations themselves.

5.1 Derivation of the Regression Equation

Firstly, the average observed execution latency for each subtask of the end-to-end task was noted for different values of the data stream size processed by the task itself. Then, an equation of the form $y = A * dss^2 + B * dss + C * cpu$, where y is the execution latency in milliseconds, *dss* is the data stream size processed by the subtask and *cpu*, the CPU utilization level (in %) of the host that runs this copy of the subtask. Initially, value of zero for the constants A, B and C is assumed. The overall absolute average percentage error between the average observed and the predicted values is computed. This is definitely a large value as the predicted values are still zero. We increase the value of the coefficient A in small positive increments. We observe a decrease in the error. At some point, we observe that the error starts increasing instead of decreasing. We stop changing the value of A at this point, and increment the value of B in small steps. When we observe a reversal in error, we stop incrementing B, and repeat the same procedure for C. Similarly we derive the regression equation for other subtasks. We now present the data for the two subtasks, namely the filter and ED programs.

The regression equation derived for the filter subtask is

$$y = (1 \times 10^{-5}) * dss^2 + (1.1 \times 10^{-2}) * dss + (5 \times 10^{-3}) * (cpu - 10)$$

and that for the ED subtask is

$$y = (9.5 \times 10^{-6}) * dss^2 + (1.2 \times 10^{-3}) * dss + (5 \times 10^{-3}) * (cpu - 10)$$

[2] Handbook of non-linear regression models / David A. Ratkowsky

The following tables compare the observed and predicted values for the subtasks. The observed values were collected from the application for different values of the data stream size processed by the subtask. The overall absolute average percentage error for filter regression equation was 1.71 and for ED it was 2.60.

Table 2. Data for Calculating the Regression Equations for the Filter Subtask

Sample	Observed Latency (ms)	Predicted Latency (ms)	Data Stream Size
1	5.92	6.00	400
2	7.86	8.00	500
3	9.96	10.20	600
4	12.45	12.60	700
5	15.72	15.20	800
6	18.88	18.00	900
7	21.70	21.00	1000
8	24.72	24.20	1100
9	27.95	27.60	1200
10	31.30	31.20	1300
11	36.60	35.00	1400
12	40.30	39.00	1500
13	44.20	43.20	1600
14	48.90	47.60	1700
15	53.75	52.20	1800
16	58.80	57.00	1900
17	63.50	62.00	2000
18	68.30	67.20	2100
19	73.90	72.60	2200
20	79.80	78.20	2300
21	85.20	84.00	2400
22	91.50	90.00	2500
23	97.85	96.20	2600
24	104.00	102.60	2700
25	110.40	109.20	2800
26	117.40	116.00	2900
27	124.70	123.00	3000
28	132.00	130.20	3100
29	139.30	137.60	3200
30	146.40	145.20	3300
31	153.60	153.00	3400
32	161.70	161.00	3500
33	170.10	169.20	3600
34	178.70	177.60	3700
35	186.80	186.20	3800
36	195.00	195.00	3900
37	203.70	204.00	4000

Table 3. Data for Calculating the Regression Equations for the ED Subtask

Sample	Observed Latency (ms)	Predicted Latency (ms)	Data Stream Size
1	2.38	2.00	400
2	3.36	2.98	500
3	4.54	4.14	600
4	5.80	5.50	700
5	7.47	7.04	800
6	9.20	8.78	900
7	11.16	10.70	1000
8	13.30	12.82	1100
9	16.10	15.12	1200
10	18.10	17.62	1300
11	20.80	20.30	1400
12	23.55	23.18	1500
13	26.50	26.24	1600
14	29.80	29.50	1700
15	33.10	32.94	1800
16	36.70	36.58	1900
17	40.50	40.40	2000
18	44.40	44.42	2100
19	48.50	48.62	2200
20	52.80	53.02	2300
21	57.40	57.60	2400
22	61.98	62.38	2500
23	66.77	67.34	2600
24	71.85	72.50	2700
25	77.10	77.84	2800
26	82.50	83.38	2900
27	88.20	89.10	3000
28	93.80	95.02	3100
29	101.10	101.12	3200
30	106.00	107.42	3300
31	112.40	113.90	3400
32	118.70	120.58	3500
33	125.40	127.44	3600
34	132.30	134.50	3700
35	139.40	141.74	3800
36	146.60	149.18	3900
37	154.00	156.80	4000

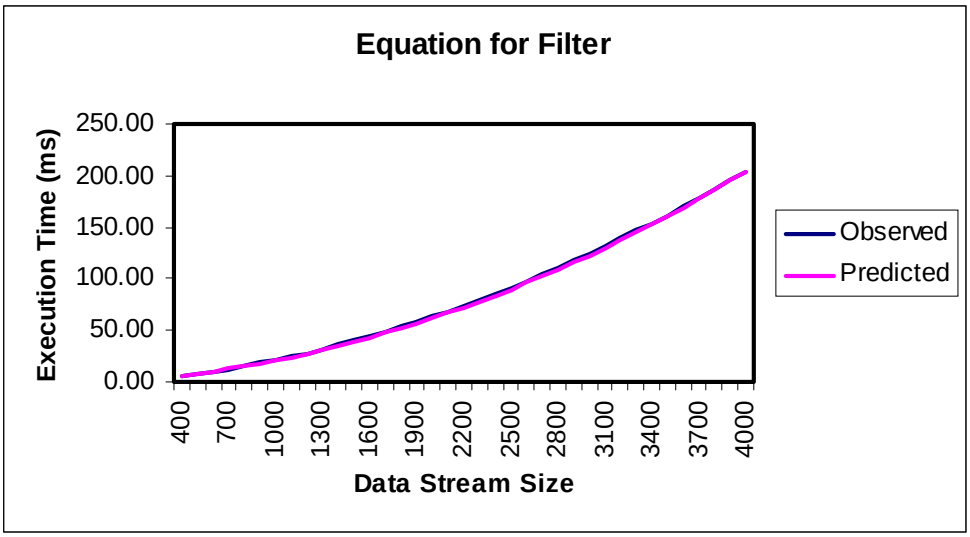

Figure 6. Correlation between predicted and observed values of the execution time for the filter subtask

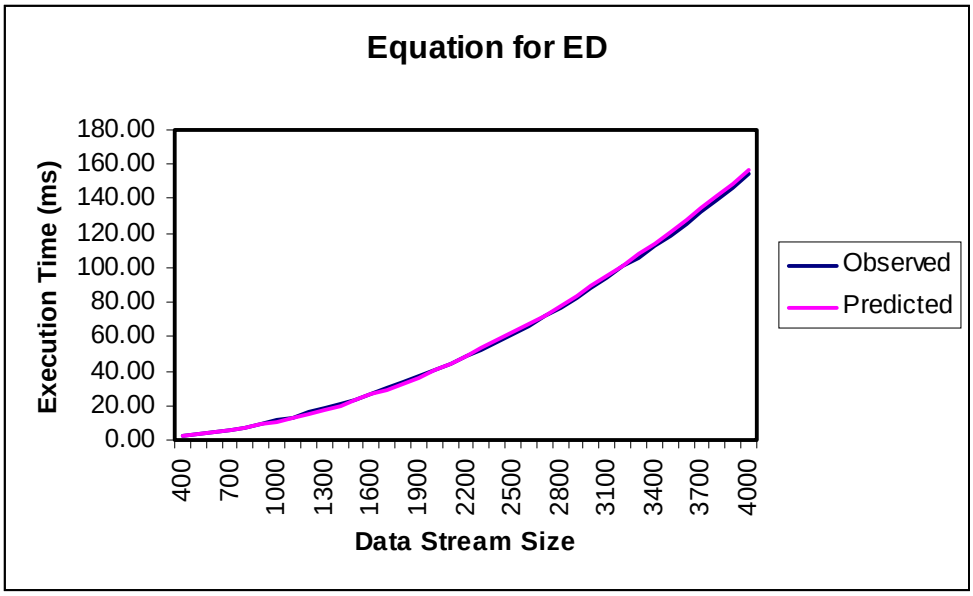

Figure 7. Correlation between predicted and observed values of the execution time for the ED subtask

From Figure 6, we can see that the observed and the predicted values for the filter subtask correlate very well. Similar is the case for the ED subtask (Figure 7).

When an end-to-end violation occurs, we consider replicating the individual sub-tasks, which miss their individual deadlines. The algorithms decide how many replicas of

which subtask programs are required for resource allocation. While the Class A algorithms use an iterative procedure to predict the number of replicas, algorithms belonging to Classes B and C use a simple heuristic to predict the number of replicas.

5.2 Class A Algorithms

Class A algorithms predict the number of replicas of the programs needed for resource allocation on the condition that the deadlines of the individual subtasks would be satisfied. The steps in the adaptive resource allocation, described previously in Section 3.2 proceed as follows.

1. *Monitor:* Define the end-to-end task latency as L and the end-to-end deadline as D and denote subtasks S_1, S_2, …S_n. An end-to-end violation occurs when more than 15 "latency misses" (one "latency miss" is when L > D) occur in a sliding window of 20 consecutive cycles.

2. *Diagnosis:* Using a model based on the EQF (Equal Flexibility) strategy suggested by [Kao97], a deadline is assigned to each individual subtask of the end-to-end task. In the model suggested by [Kao97], each task has the following parameters, arrival time *ar*(X), deadline *dl*(X), slack *sl*(X), real execution time *ex*(X), and predicted execution time *pex*(X). We use a simplified version of this model. The slack of a task is the difference between its deadline and the execution latency. EQF divides the total slack among the subtasks in proportion to their execution times. Effectively, the deadline of a subtask is assigned in proportion to its execution time. This is illustrated by the following expression.

$$dl(T_j) = ar(T_j) + pex(T_j) + [dl(T) - ar(T) - \Sigma\, pex(T_j)] * [\, pex(T_j)/ \Sigma\, pex(T_j)]$$

When an end-to-end violation occurs, at least one of the individual subtasks "misses" its deadline, i.e., its execution latency exceeds this assigned deadline. In this step, we decide which subtask misses its deadline.

3. *Analysis:* Consider the sub-task S_k. Denote the deadline with one copy of the program as D_k. When the end-to-end violation occurs, the latency of the subtask L_k is greater than its deadline D_k. The calculation proceeds as follows. At this point of time, say the program is processing N data items. We first assume one replica of the program. Then we have two copies of the program, each copy processing N/2 data items and the deadline is scaled down accordingly as $D_k/2$. We predict the execution latency of the program on that host using the regression equation already developed. Depending on the choice of the algorithm within Class A itself, either the equation calculated from the subtask profiles or the one calculated online is used. Then, we predict the final execution latency as the maximum of all these execution latencies. Denote this final predicted execution latency as F_k. If $F_k < D_k/2$, we end the analysis and return to start one replica of the program. If F_k is still greater than $D_k/2$, we increase the number of copies assumed by one. Now we have two additional copies, and we perform the same prediction. We then check $F_k < D_k/3$. In general, if we have c copies of the program, then we check F_k (for N/c data items) $< D_k/c$. The total number of copies of replicas predicted is restricted to '*h*', the number of hosts in the system to take advantage of the parallelism of execution by executing each replica on a different host.

4. *Select:* The c number of copies is started on hosts that do not run a copy of the program already to make maximum utilization of the resulting parallelism. The hosts are selected in the ascending order of their CPU utilizations. If we run out of hosts, only then do we "reuse" the hosts.

5. *Enact:* Once the number of replicas of the subtask program and the hosts on which they need to be run is decided, the resource manager sends the commands to the system daemons on those hosts requesting the replication of the calculated copies of the subtask

The Class A algorithms are described in detail.

5.2.1 *Algorithm A1*

This algorithm predicts the number of replicas for a subtask with the highest execution latency, which misses its deadline, and replicates only that program when the end-to-end violation occurs. The maximum number of replicas predicted is restricted to the number of hosts in the system. The coefficients of the non-linear multiple regression equation used are calculated manually by hand using an iterative procedure. Since the algorithm has to look through '*h*' number of hosts for each of the '*h*' maximum number of replicas, the complexity of the algorithm A1 is $O(h^2)$. A short description in the form of pseudo-code is shown below in Figure 8.

```
If an end-to-end violation occurs
        If subtask k misses its deadline
                Sort the host table in descending order
                Copies (of subtask k) = 0
                Until analysis is done loop
                        Increment copies
                        Until copies are exhausted loop
                                Assign copies to hosts not running the subtask
                                Predict Latencies if subtask run on those hosts using
                                the regression equation calculated from the application
                                profiles
                        End loop
                Final Predicted Latency is maximum of all the predicted latencies
                If Final Predicted Latency < subtask deadline for N tracks/(total copies)
                        Copies calculated are sufficient
                        Start copies on those hosts
                        Break
                End if
                End Loop
        End if
End if
```

Figure 8. Pseudo-code of algorithm A1

5.2.2 *Algorithm A2*

This algorithm predicts and replicates all subtasks that miss their deadlines when the end-to-end violation occurs. The same regression equation used by Algorithm A1 is used here. This algorithm performs the same computations as Algorithm A1 for each of

the '*n*' replicable subtasks. If there are '*h*' number of hosts in the system, then the complexity of the algorithm A2 is $O(nh^2)$. The pseudo-code for Algorithm A2 is almost similar to that of Algorithm A1, but the procedure is repeated for all the subtasks that miss their individual deadlines.

```
If an end-to-end violation occurs
        For all subtasks i (1 ≤ i ≤ n)
                If subtask i misses its deadline
                        Sort the host table in descending order
                        Copies (of subtask i) = 0
                        Until analysis is done loop
                                Increment copies
                                Until copies are exhausted loop
                                        Assign copies to hosts not running the subtask
                                        Predict Latencies if subtask run on those hosts
                                        using the regression equation calculated from the
                                        application profiles
                                End loop
                        Final Predicted Latency is maximum of all the predicted latencies
                        If Final Predicted Latency < subtask deadline for N tracks/(total
                        copies)
                                Break
                        End if
                        End Loop
                End if
        End for
End if
```

Figure 9. Pseudo-code of algorithm A2

5.2.3 *Algorithm A3*

This algorithm is a variant of Algorithm A1 but the coefficients of the regression equation are calculated online using the automatic profiling method. The complexity is same as that for Algorithm A1, i.e. $O(h^2)$, where *h* is the number of hosts in the system. The pseudo-code for this algorithm is same as that for Algorithm A1, except that the regression equations used for prediction are calculated online using the automatic profiling method. The automatic profiling method is described in Chapter 6.

5.2.4 *Algorithm A4*

Algorithm A4 is a variant of Algorithm A2, where the coefficients of the regression equation are calculated online using the automatic profiling method. The complexity is same as that for Algorithm A2, i.e. $O(nh^2)$, where n is the number of replicable subtasks, and h is the number of hosts in the system. The pseudo-code for this algorithm is the same as that for Algorithm A2, except that the regression equations used for prediction are calculated online using the automatic profiling method.

5.3 Class B Algorithms

The adaptive resource management process for Class B algorithms is different from that of the Class A algorithms. The steps involved are described below.

1. *Monitor:* This is the same as for Class A algorithms.

2. *Diagnosis:* In this step, the subtask with the highest execution latency is identified as the bottleneck subtask that has to be replicated. ([Kao97] suggests that in real-time database systems, long transactions miss more deadlines compared to the short ones.)

3. *Analysis:* Analysis determines the required number of replicas of the subtask program. Unlike Class A algorithms, these algorithms do not make any prediction, but use a simple heuristic shown in Table 4.

4. *Select:* The execution latency of the subtask processing a certain data stream size is converted into equivalent CPU resource consumption of a host. The equation is again derived from the subtask data stream size – CPU utilization profile. The equation calculated for the filter subtask is

$$y = (1 \times 10^{-6}) * dss^2 + (1.5 \times 10^{-3}) * dss$$

Here, *y* is the equivalent CPU resource (in %) that would be consumed if the filter subtask were to process a data stream size of *dss* on the host.

If the first fit[3] technique is used, then the first host that has the available resources to satisfy the resource consumption requirements of the replicas is chosen.

If the best-fit[4] technique is used, then the host that has the smallest volume of resources remaining and still satisfies the resource consumption requirements of the replicas is chosen.

If the worst fit[5] technique is used, then the host that has the largest volume of resources remaining and still satisfies the resource consumption requirements of the replicas is chosen.

5. *Enact:* This step is the same as described for Class A algorithms.

Table 4. Heuristic used by Class B and Class C Algorithms

% Increase in Data Stream Size	Replicas
Up to 25 %	1
25 % < Increase <= 50 %	2
50 % < Increase <= 75 %	3
75 % < Increase <= 100 %	4
Greater than 100 %	5

[3] First Fit: Select the first host that has sufficient resources to meet the request.

[4] Best Fit: Select the host with the smallest remaining volume that most closely fulfils the request (minimizes leftover).

[5] Worst Fit: Select the host that provides the largest volume of resources that fulfils the request (maximizes leftover).

5.3.1 *Algorithm B1*

This algorithm uses the first fit resource fitting mechanism. The pseudo-code for this algorithm is shown in Figure 10.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of the subtask required using the heuristic
        Choose the first (fit) host that has sufficient resources
        Start the replicas on that host
End if
```

Figure 10. Pseudo-code for algorithm B1

5.3.2 *Algorithm B2*

This algorithm uses the best-fit resource fitting mechanism. The pseudo-code for this algorithm is shown in Figure 11.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of the subtask required using the heuristic
        Choose the 'best fit' (highest loaded) host that has sufficient resources
        Start the replicas on that host
End if
```

Figure 11. Pseudo-code for algorithm B2

5.3.3 *Algorithm B3*

This algorithm uses the worst fit resource fitting mechanism. The pseudo-code is shown in Figure 12.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of the subtask required using the heuristic
        Choose the 'worst fit' (least loaded) host that has sufficient resources
        Start the replicas on that host
End if
```

Figure 12. Pseudo-code for algorithm B3

All the Class B algorithms have to look through '*h*' number of hosts before picking the one which satisfies the condition for each of the '*n*' replicable subtasks. Hence the complexity of all the Class B algorithms is $O(nh)$.

5.4 Class C Algorithms

This class of algorithms is a slight variant of the Class B algorithms in that the host already running a copy of the subtask is not used for running a replica of the same program in order to take the maximum advantage of the parallelism arising from the execution of the copies of the same subtask on different machines. The various steps of the adaptive resource management are as follows.

1. *Monitor:* This is the same as for the Class A or Class B algorithms.

2. *Diagnosis:* The diagnosis is also same as the Class B algorithms.

3. *Analysis:* In determining the required number of replicas of the subtasks required, this class of algorithms also uses the same heuristic as the Class B algorithms.

4. *Select:* If the first fit technique is used, then the first host that has the available resources to satisfy the resource consumption requirements of the replicas and which does not run a copy of the same subtask is chosen.

If the best-fit technique is used, then the host that has the smallest volume of resources remaining and still satisfies the resource consumption requirements of the replicas and which does not run a copy of the same subtask is chosen.

If the worst fit technique is used, then the host that has the largest volume of resources remaining and still satisfies the resource consumption requirements of the replicas and which does not run a copy of the same subtask is chosen.

5. *Enact:* This step is the same as for the Class B algorithms.

The different types of algorithms are described as follows.

5.4.1 *Algorithm C1*

This algorithm similar to Algorithm B1 with a condition that the host that already runs a copy of the subtask is not used for replication of that subtask again. The pseudo-code for this algorithm (Figure 13) is same as that for Algorithm B1 except for the condition stated above.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of subtask required using the heuristic
        Choose the first (fit) host that has sufficient resources and does not run a copy of
        the same subtask
        Start the replicas on that host
End if
```

Figure 13. Pseudo-code for algorithm C1

5.4.2 *Algorithm C2*

This algorithm and its pseudo-code (Figure 14) are the same as that for Algorithm B2 except for the condition stated above.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of the subtask required using the heuristic
        Choose the 'best fit' (highest loaded) host that has sufficient resources and does
        not run a copy of the same subtask
        Start the replicas on that host
End if
```

Figure 14. Pseudo-code for algorithm C2

5.4.3 *Algorithm C3*

This algorithm and its pseudo-code (Figure 15) are the same as that for algorithm B3 except for the condition stated above.

```
If end-to-end violation occurs
        Calculate % increase in data items
        Calculate number of copies of the subtask required using the heuristic
        Choose the 'worst fit' (least loaded) host that has sufficient resources and does
        not run a copy of the same subtask
        Start the replicas on that host
End if
```

Figure 15. Pseudo-code for algorithm C3

All the Class C algorithms have to look through '*h*' number of hosts before picking the one which satisfies the condition (and which does not run a copy of the replicable subtask previously) for each of the '*n*' replicable subtasks. Hence the complexity of all the Class C algorithms is $O(nh)$.

5.5 Scalability Issue

The resource allocation algorithms designed above solve the issue of missed deadlines for a single task during a mission period. But, in a complex real-time system, usually we have to deal with several tasks running simultaneously in the system. In such a case, resource allocation decisions are difficult to make because, while allocating the resources for a task, we should take care that this decision does not affect the other tasks adversely.

In this regard, a scalable set of algorithms (Class H) has been designed that makes resource allocation decisions for each task based on the "health" of the other tasks in the system. By "health" of a task, we take into account the slack of each subtask of that task on each host of the system. The slack of a subtask is the difference between its deadline and its execution latency. A parameter called as the Host Fitness Index is calculated for each host of the system and on a per-task basis, as explained below. The host fitness index is a measure of the preparedness of a host for running a replica of a subtask (of a task with a triggered violation) as a result of a resource allocation decision made by the resource manager. The goal is to minimize the overall Missed Deadline Ratio, which is characterized by the ratio of the total deadlines missed by each task in the system to the total number of deadlines that could have been satisfied.

Host Fitness Index (for a host *H* and task *T*) =

W1 * [Σslack(subtasks of *T* on *H*) + Σslack(all other subtasks on that host)]
+ *W2* * slack(task *T*)
+ *W3* * [Σslack(all subtasks of *T* except subtasks of *T* on *H*)]

The host fitness index is calculated at runtime when a violation is detected for a particular task. If the index is more than the predetermined threshold value, then that host is considered suitable for replication. After a host has been selected, care is taken not to replicate the subtask of the task missing its deadline on that host if the host already runs a copy of that program. Instead, another subtask of the same task is considered for replication on that host.

The constants W1, W2, and W3 in the above equation have been chosen on the basis of the following considerations. The weight W2 assigned to the slack of the end-to-end task has the highest value. Then the other two unequal weights, namely, W1, and W3 can be in any particular order. The following two cases have been considered. The first algorithm, H1, uses the weights in the order, W2 > W1 > W3. Another variant of the algorithm, H2, uses weights in the manner W2 > W3 > W1.

If there are ‘*t*’ number of tasks in the system, ‘*n*’ number of replicable subtasks per task, and ‘*h*’ number of hosts in the system, then the host fitness index is calculated on a per-host, per-task basis, and for ‘*n*’ number of replicable subtasks of that task and the complexity of either algorithms is $O(hnt)$.

Chapter 6 Automatic Profiling

Previously, we have seen that the predictions regarding number of replicas of subtasks needed were made using the non-linear multiple regression equation whose coefficients were calculated by hand using an iterative procedure. In this section, we present a method by which the coefficients of the regression equation can be computed online even while the system is running. This is particularly useful in the case of profiling the system on different platforms. Suppose we port the system from one platform to another, the execution latency profile of the programs may vary by a significant amount, and the equations designed earlier may no longer be accurate. Also, it was found that the automatic profiler tracks the applications' execution latencies well. In fact, it would be shown later in the experimental results section that one of the algorithms based on the automatic profiling technique offers overall superior performance.

The procedure for the computation of the coefficients of the regression equation is presented below. The system starts out with the assumption of the equation of the form of $y = A * dss^2 + B * dss + C * cpu$. The initial values of the coefficients, namely A, B, and C are assumed to be zero. After the system starts up, the automatic profiler program starts collecting the data for each subtask, like the execution latency, CPU utilization level of the host, the data stream size, etc. At periodic intervals, the coefficients are calculated. Say, we have N samples of data. The algorithm proceeds in the following manner.

1. Assume a small fractional value of A. These fractional values are assigned after observing the order of the coefficients. Compute the predicted latency for an individual sample using the above equation. Take the absolute error between the predicted and the observed latency of that sample. As the experiment proceeds, the samples increase to a very large extent. Add the error for all the samples and divide the error with the number of samples. This yields the average cumulative error for that particular subtask. If it is less than a preset threshold error, then the computation is done for that subtask.

2. If the average cumulative error is still greater than the threshold error, try to "fine-tune" the coefficient. Backup the value of the coefficient and increment the value of the coefficient by the same fractional value. Again calculate the average cumulative error. If this value is less than the threshold, return. If the error is better than the previous error and more than the threshold error, continue this process until the error is less than the threshold. At any point of time if the error is less than the threshold, analysis ends for that subtask. If the number of iterations exceeds a specific timeout value, we stop the analysis. This is to prevent infinite number of iterations.

3. If the error is worse than the previous error, and still more than the threshold, we move on to the next coefficient namely B, and perform the same analysis for that coefficient. Next, we move on to the last coefficient namely, C.

4. If however, in the first iteration itself we find that the error becomes worse, then we go on decrementing the value of the coefficients in a fractional manner instead of incrementing them. This small condition works very well in determining the coefficients.

After this algorithm is implemented, it was observed that though the coefficients fluctuate in the beginning of the experiment, they stabilize very soon, and there is very little difference between the coefficients resulting from the later iterations. The algorithm may be summarized in a flowchart shown in Figure 16.

The regression equations calculated for the subtasks Filter and ED were respectively

$$y = (1.37 \times 10^{-5}) * dss^2 + (3 \times 10^{-3}) * dss + (3 \times 10^{-2}) * cpu$$

$$y = (1.14 \times 10^{-5}) * dss^2 + (1 \times 10^{-2}) * cpu$$

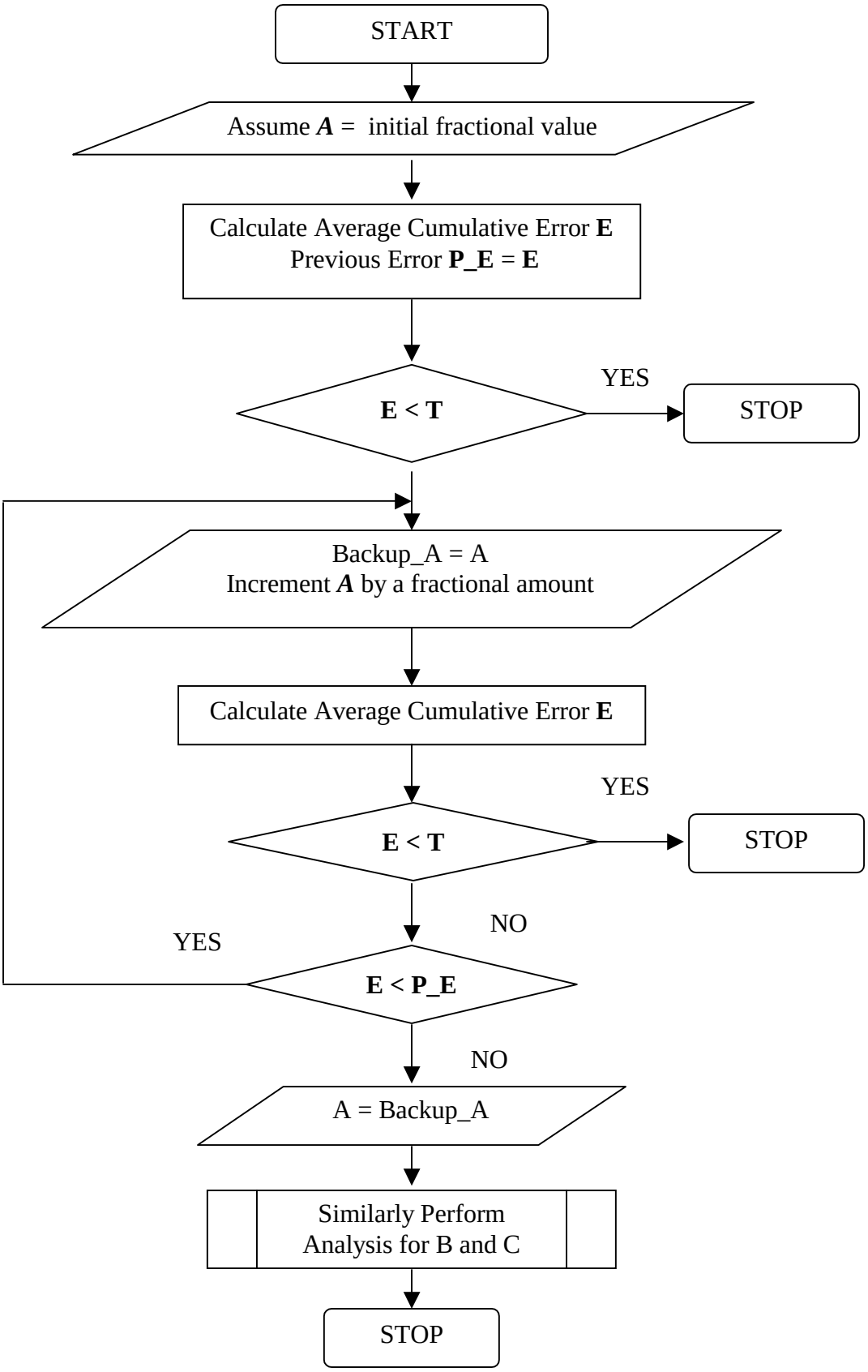

Figure 16. Flowchart for the automatic profiling method

Chapter 7 Experimental Results

In the basic experimental setup, all the experiments were run in a distributed manner, in the sense that, the subtasks of the periodic sensing task were run on different hosts. The hosts ran Redhat Linux 6.2 and all the machines were on an Ethernet network. The hosts were synchronized in time by NTP [Mills95]. Each plot corresponds to the data collected for around 300 experiments.

We now present the experimental results. The experiments were conducted for various track scenarios, where the increase in the track loads in various cases were 20 %, 30 %, 40 %, 55 %, 80 %, and 105%. Figure 17 presents the first parameter of interest namely the Missed Deadline Ratio (MDR).

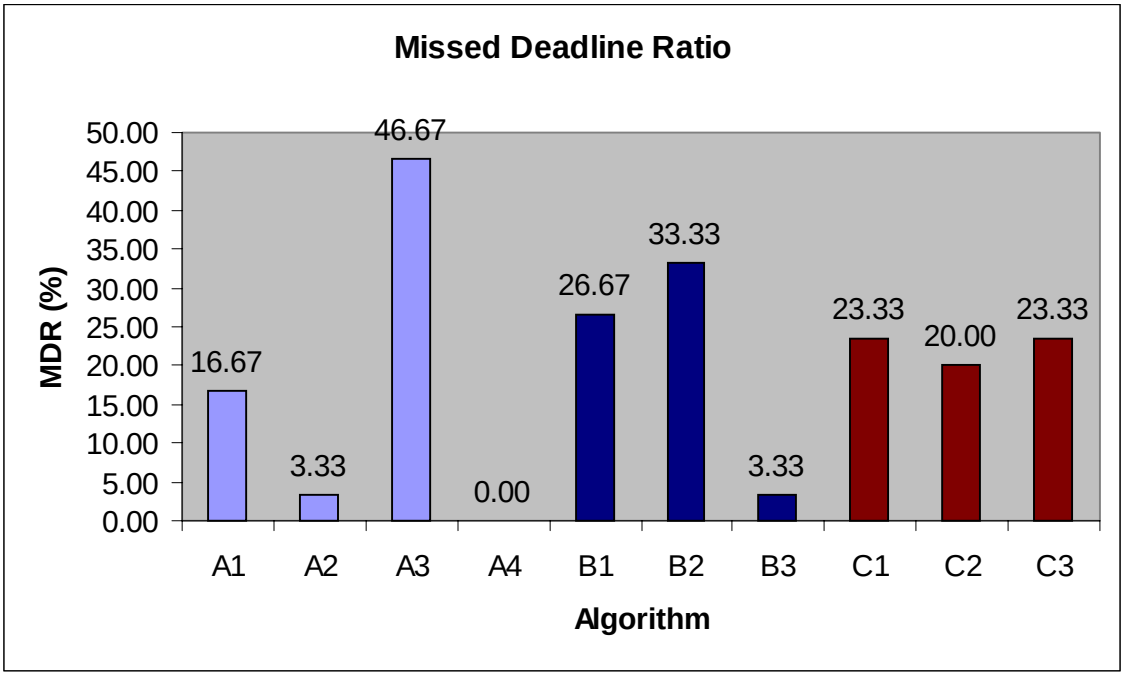

Figure 17. Missed Deadline Ratio for various algorithms

From Figure 17, it is clear that Algorithm A4 that replicates all the subtasks that miss their individual deadlines in the event of an end-to-end violation, and uses the regression equation whose coefficients have been calculated using the automatic profiling

method offers the lowest value of MDR. This means that the algorithm does not miss even one deadline after it has made the resource allocation decision. The result becomes more obvious when we see the average number of replicas predicted by the algorithm.

On the other hand, Algorithm A3 that replicates only the subtask with maximum execution latency missing its individual deadline on an end-to-end violation, and uses the same regression equation as used by Algorithm A4 (i.e., automatic profiler) offers the worst (highest) value of MDR. It misses almost half of its deadlines after it has made a resource allocation decision. We will give an explanation for this behavior once we examine the average number of replicas predicted by that algorithm.

In general, within Class A itself, algorithms that replicate all the subtasks missing their individual deadlines offer lower MDR than the algorithms which replicate only the subtask with maximum execution latency and which misses its individual deadline.

We also observe that Algorithm A2 that replicates all the subtasks missing their individual deadlines and that uses the regression equation whose coefficients have been calculated manually to predict the number of replicas offers the second best performance in terms of MDR along with Algorithm B3. An explanation for Algorithm A2 would be clear when we look at the average number of replicas predicted. For Algorithm B3 all can be said is that it picks the least loaded host at all times (even though the host already runs a copy of the subtask) and hence the performance. The reason for it being not performing on par with Algorithm A4 is clear from its prediction of required average number of subtask replicas.

Comparing across classes, we see that the Class C algorithms perform better (lower MDR) than their Class B counterparts except for Algorithm C3. The possible explanation is that Class C algorithms make use of the parallelism due to execution of the replicas of the subtasks on different machines. The exception for Algorithm C3 can be explained as follows. Since Algorithm C3 uses the worst fit mechanism, it looks for the least loaded host for running its replicas. But since it bypasses the host that already runs a

copy of that subtask, there is a possibility that this host that is bypassed is the least loaded host and the algorithm actually ends up choosing a host that is not actually the least loaded host.

Explanations would be clearer if we look at the average number of replicas predicted by each of the algorithms in Figure 18.

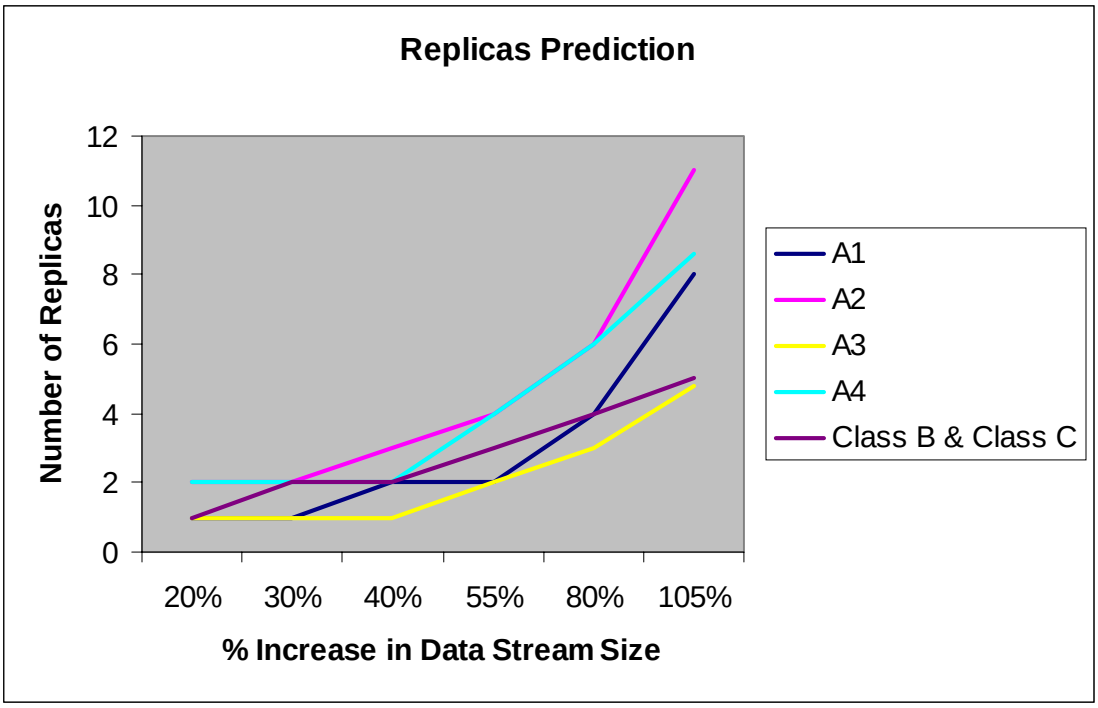

Figure 18. Average number of replicas predicted by the algorithms

We first explain the worst performance offered by Algorithm A3. From Figure 18 it is clear that Algorithm A3 predicts the lowest number of replicas of subtasks compared to the other algorithms as the data stream size increases. There is a very high possibility that this number of replicas is not sufficient for resource allocation compared to the increase in the data stream size.

Similarly, Algorithm A2 predicts the highest number of replicas and offers second best performance in terms of MDR, but later we would see that it offers the best performance in terms of steady state latency slack. Algorithm A4 follows A2 closely and

probably predicts the “optimum” number of replicas because it does not miss any of its deadlines and yet offers decent steady state latency slack (second best) as we will see later.

We also see that Algorithm B3 does not perform as well as A4 because its prediction of replicas needed is on the lower side when compared to A4 even though it picks the least loaded host at all times.

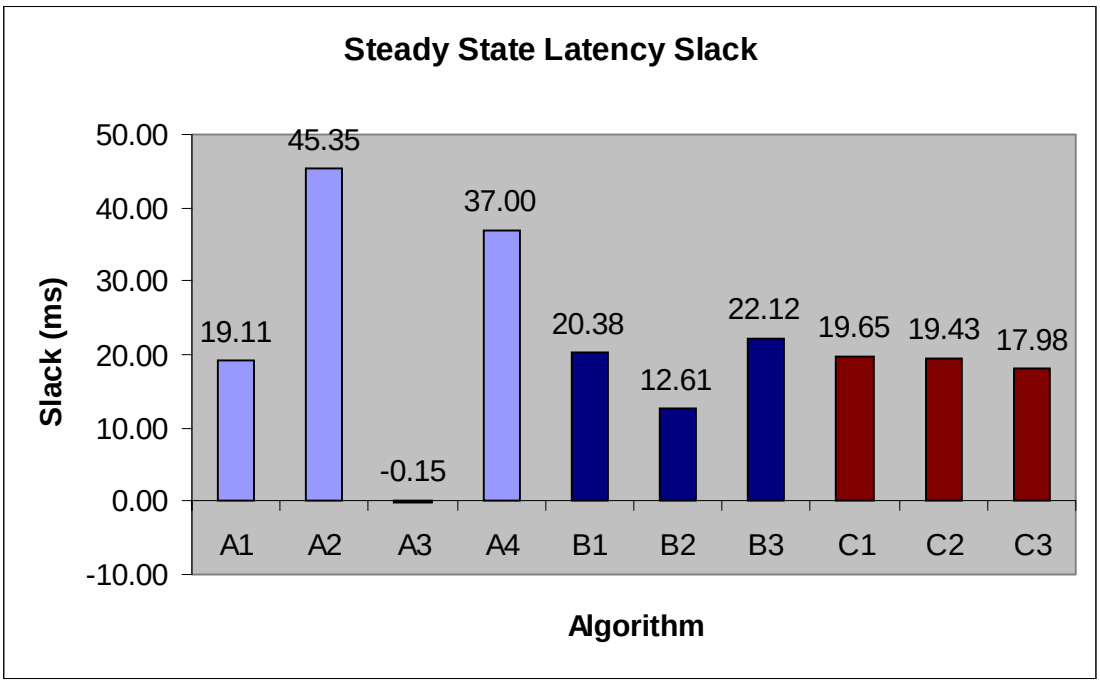

Figure 19. Average Steady State Latency Slack offered by the algorithms

Figure 19 presents the steady state latency slack offered by the algorithms. Algorithm A2 that predicts the highest number of replicas offers the highest steady state latency slack (margin between deadline and latency, higher, the better) followed by Algorithm A4 which offers the second best performance which comes next to A2 in prediction of replicas. Again we observe that Algorithm A3, which predicts the least number of replicas, suffers badly offering (average) negative slack. This means that the overall end-to-end latency in the cases of missed deadlines is so high that it shadows the overall end-to-end latency in the cases where it meets its deadlines.

Comparing across classes, we see that generally Class B algorithms perform better than Class C algorithms with the exception of Algorithm B2. The exception for B2 could be due to the fact that this algorithm picks the highest loaded host, and hence yields the worst performance. This result should also be applicable to Algorithm C2, but C2 performs better than C3. This exception can be explained as follows. When we give a probable explanation for worse performance of Algorithm C3 saying that it does not necessarily pick the least loaded host, similarly we can say that Algorithm C2, which also bypasses the host already running a copy of the subtask, not necessarily pick the heaviest loaded host and hence its performance is not necessarily worse. What might be happening is that Algorithm C3 does not pick the least loaded host, and Algorithm C2 does not pick the heaviest loaded host, and hence the performance in Class C is opposite to what is happening within Class B.

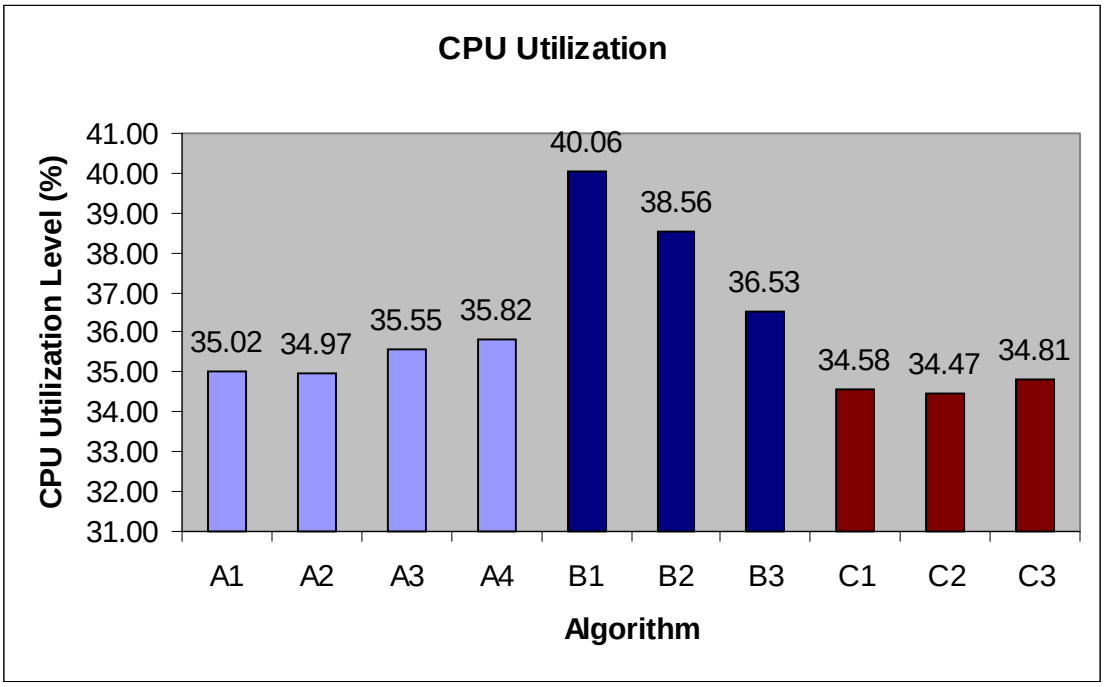

Figure 20. Average CPU Utilization Levels of the algorithms

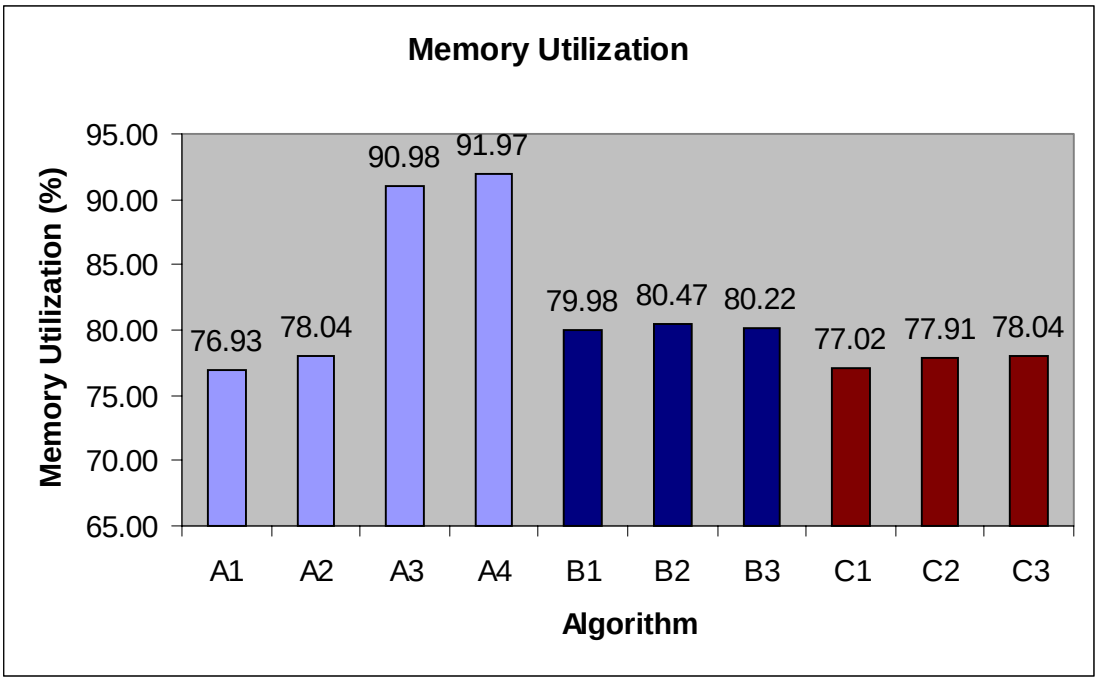

Figure 21. Average Memory Utilization Levels of the algorithms

From Figures 20 and 21, we observe that some of the results are counter-intuitive. Within Class A Algorithms, Algorithm A2 that predicts the most number of replicas surprisingly has the least average CPU Utilization, and also its memory utilization is not the highest amongst the Class A Algorithms.

Comparing across classes, we see that in general Class C algorithms consume lesser resources than their Class B counterparts. Within Class B itself, we see that Algorithm B1 does not exhibit any fixed pattern such as least or highest resources all the time because, the first fit mechanism randomly selects the first host which can satisfy the requirements, and hence it is unpredictable in its resource requirements. However, it can be seen that the Algorithm B3 based on worst fit (selecting the least loaded host all the time) consumes lesser resources all the time. This is not true within Class C where the worst fit Algorithm C3 performs worse than the best-fit Algorithm C2, which has already been explained due to the probable reason that Algorithm C3 may not actually pick the least loaded host all the time and Algorithm C2 may not pick the highest loaded host all the time.

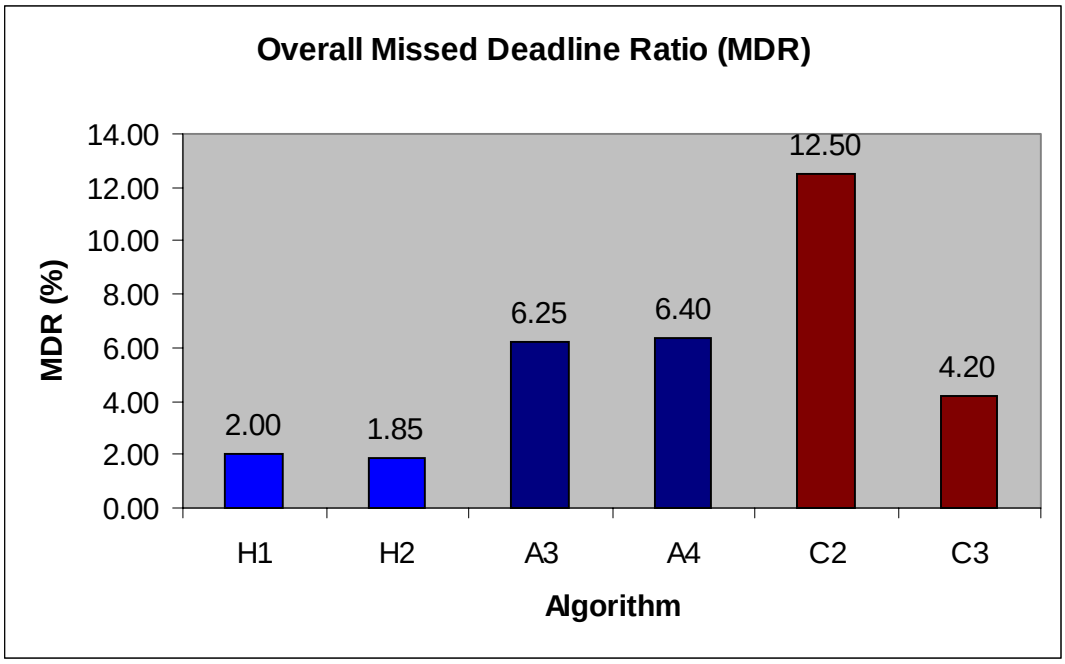

Figure 22. Overall Missed Deadline Ratio for multiple tasks

Figure 22 shows the overall missed deadline ratio for several (four) tasks running simultaneously in the system. Clearly, the algorithms based on the host load index, i.e. algorithms H1 and H2 yield the best performance. This is because they take into consideration the other tasks in the system, by including their slack information into the equation. However, Algorithm H2 that assigns a higher weight to the other subtasks of the task under consideration than only the subtasks running on that particular host performs slightly better than Algorithm H1, which assigns a higher weight to the subtask of a task on a host under consideration. This may be true because, if the subtask of the task on a host under consideration performs better does not necessarily mean that the rest of the subtasks of that task on other hosts are in good shape. It is good to concentrate on the rest of the subtasks rather than a single subtask of the task on a host under consideration.

It has been observed that the Class A algorithms employing the regression equation computed using the automatic profiling method for prediction of number of replicas of subtasks generally performed better than those employing equations derived analytically by hand. Also, Class C algorithms performed better than Class B algorithms.

Hence the experiments were conducted only for algorithms A3, A4, C2 and C3. Algorithm C1 employing the first fit technique also showed no particular trend, so it was not considered.

Within the Class A algorithms, it can be seen that the Algorithm A3 which replicates a single subtask using the regression equation (computed using automatic profiling) performs slightly better than Algorithm A4 which replicates all the subtasks. This is because, too many replicas may be useful to the task under consideration but they also use up the resources that might adversely affect other tasks. This result becomes more significant as the number of tasks in the system increase.

Within the Class C Algorithms, where all the algorithms use the same heuristic to determine the number of replicas, Algorithm C2 which uses the best fit technique (i.e. the heaviest loaded host) performs worse than Algorithm C3 which uses the worst fit technique (least loaded host). The result is obvious.

Before we make any conclusions, we also make an important observation. Usually in a complex and huge system, there might be several subtasks that could be replicated. An algorithm that decides to replicate one or few, but not all the subtasks would take less decision time than an algorithm that decides to replicate all the subtasks that miss their deadlines. The former type of algorithms would result in lesser tardy times, but also might result in a lower value of the steady state latency slack. The latter type of algorithms would spend more time on deciding the replicas for all the subtasks, hence resulting in a higher tardy time and at the same time a higher value of the steady state latency slack. In this case, the maximum number of replicable subtasks were two and the decision time taken by both the types of algorithms does not differ much and the order of the decision time was in milliseconds, where as the cycles of the benchmark were spaced about a second or thousand milliseconds. Hence, there was not much significant tardy time – steady state latency slack tradeoff visible.

Chapter 8 Conclusions

The following conclusions can be drawn from the experimental results presented in the previous chapter.

1. Predictive greedy algorithms generally produce lower missed deadline ratios than their non-predictive counterparts. (The only exception to this is the performance of the algorithm A3). Profile computation and regression equations are thus generally effective.

2. The algorithms that replicate all the subtasks offer better slack and lower percentage tardy time values, and lower missed deadlines ratio than the algorithms which replicate one or few subtasks but not all of them. As the number of subtasks increases, the former type of algorithms may offer higher values of percentage tardy time than the latter type of algorithms.

3. Algorithms which take advantage of the parallelism of execution by not replicating the subtasks on hosts that already run a copy of that subtask generally perform better than those algorithms which do not consider this issue of parallelism.

4. Among the various resource fitting mechanisms, the algorithms based on the worst fit, or which select the least loaded host perform better than those based on the best fit mechanism which end up picking the heaviest loaded host.

5. Algorithms with the worst fit technique and which also take advantage of parallelism of execution may not always pick the least loaded host and hence, may not necessarily perform better than the algorithms with the best fit technique with parallelism which, might actually end up not picking a heaviest loaded host.

6. Scalable (non-greedy) algorithms perform better than the rest of the greedy algorithms when there are several tasks running in the system simultaneously.

7. Observed MDR is in the range of 0 % to 48 % across 300 experiments for greedy algorithms and observed SSLC is in the range of –0.15 ms through 45.35 ms for the same.

8. Observed Overall MDR is in the range of ~2% for the non-greedy algorithms.

Contributions

The major contribution of this thesis is predictive, statistical regression theory-based resource allocation algorithms that are presented in Chapter 5. These algorithms are generally shown to produce lower missed deadline ratios than the empirically designed strategies for periodic tasks under the experimental conditions that were studied. Other (minor) contributions include:

(1) scalable resource management algorithms that minimize aggregate missed deadline ratios when there are multiple application tasks, and

(2) resource management middleware software that is capable of automatically profile the application tasks to determine regression equations that forecast subtask timeliness.

Limitations

The following are the major limitations of this work:

1. The application model used here is assumed to be “adaptable” by replicating some of the bottleneck subtasks. In general, this may not be true with all dynamic real-time applications.

2. The application subtasks must be programmed so that it interfaces with the resource management middleware and adheres to its communication semantics.

3. The design of the application subtasks should be such that its behavior must be amenable to regression analysis. This will enable the design of regression theory-based predictive resource allocation algorithms.

References

[AB98] A. Atlas and A. Bestavros, "Statistical Rate Monotonic Scheduling," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 123-132, December 1998.

[Bak91] T. P. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, Vol. 3, No. 1, pp. 67-99, March 1991.

[BN+98] S. Brandt, G. Nutt, et al., "A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 307-317, December 1998.

[CJR92] R.K. Clark, E.D. Jensen, and F.D. Reynolds, "An Architectural Overview of the Alpha Real-Time Distributed Kernel," *Proceedings of the USENIX Workshop on Microkernel and Other Kernel Architectures*, Seattle, April 1992.

[Cla90] R. K. Clark, "Scheduling Dependent Real-Time Activities," CMU-CS-90-155 (*Ph.D. Thesis*), Department of Computer Science, Carnegie Mellon University, 1990.

[CSR86] S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166-174, 1986.

[HS92] C-H. Hou and K. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 146-155, December 1992.

[HSNL97] D. Hull, A. Shankar, K. Nahrstedt and J. W. S. Liu, "An End-to-End QoS Model and Management Architecture," *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pp. 82-89, 1997.

[Jen92] E. D. Jensen, "Asynchronous Decentralized Real-Time Computer Systems," *Real-Time Computing*, W. A. Halang and A. D. Stoyenko (Editors), Proceedings of the NATO Advanced Study Institute, St. Martin, Springer-Verlag, October 1992.

[Kao95] B. C. Kao, "Scheduling in Distributed Soft Real-Time Systems With Autonomous Components," *PhD Thesis*, Princeton University, November 1995.

[Kao97] Ben Kao and Hector Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System", *IEEE Transactions on Parallel and Distributed Systems*, Vol.8, No. 12, pp. 1268-1274, December, 1997.

[KM97] T.-W. Kuo and A. K. Mok, "Incremental Reconfiguration and Load Adjustment in Adaptive Real-Time Systems," *IEEE Transactions on Computers*, Vol. 46, No. 12, pp. 1313-1324, December 1997.

[Koob96] G. Koob, "Quorum," *Proceedings of the Darpa ITO General PI Meeting*, pp. A-59-A-87, October 1996.

[Kop97] H. Kopetz, *Real-Time Systems: Design Principles for Distributed, Embedded Applications*, Kluwer Academic Publishers, 1997.

[Leh96] J. P. Lehoczky, "Real-time queueing theory," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 186-195, December 1996.

[LL73] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, 1973.

[Loc86] C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," CMU-CS-86-134 (*Ph.D. Thesis*), Department of Computer Science, Carnegie Mellon University, 1986.

[LRT92] J. P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 110--123, 1992.

[LSS87] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard-real-time environments," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261-270, 1987.

[LW82] J. Y. T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, Vol. 2, No. 2, 1982.

[Mill95] Mills, D.L, "Improved Algorithms for Synchronizing Computer Network Clocks," *IEEE/ACM Transactions on Networks*, pp. 245 – 254, June 1995.

[Mok83] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," *PhD Thesis*, M.I.T, Cambridge, Massachusetts, 1983.

[RCF97] I. Ripoll, A. Crespo, and A. G. Fornes, "An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic Priority Preemptive Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 6, pp. 388-400, June 1997.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek, "A Resource Allocation Model for QoS Management," *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 298-307, 1997.

[RSYJ97] D. Rosu, K. Schwan, S. Yalamanchili and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 320-329, December 1997.

[RSZ89] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1110-1123, August 1989.

[RTL93] S. Ramos-Thuel and J. P. Lehoczky, "On-line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 160-171, 1993.

[RWS99] B. Ravindran, L. R. Welch, and B. Shirazi, "Resource Management Middleware for Dynamic, Dependable Real-Time Systems," *Journal of Real-Time Systems*, Accepted for publication, to appear.

[SB96] M.Spuri and G.Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Journal of Real-Time Systems*, pp. 179-210, March 1996.

[Shi91] K.G. Shin, "Harts: A Distributed Real-Time Architecture," *IEEE Computer*, 24(5), pp. 25-35, May 1991.

[SK97] D.B. Stewart and P.K. Khosla, "Mechanisms for Detecting and Handling Timing Errors," *Communications of the ACM*, Vol. 40, No. 1, pp. 87-93, January 1997.

[SKG91] L. Sha, M. H. Klein, and J. B. Goodenough, "Rate Monotonic Analysis for Real-Time Systems," In A. M. van Tilborg and G. M. Koob (Editors), *Scheduling and Resource Management*, pp. 129-156. 1991.

[SL96] J. Sun and J.W.S. Liu, "Bounding Completion Times of Jobs With Arbitrary Release Times And Variable Execution Times," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 2-12, 1996.

[SLS88] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 251-258, 1988.

[SR91] J.A. Stankovic and K.Ramamritham, "The Spring Kernel: A New Paradigm for Real-time Systems," *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[SRC85] J.A. Stankovic, K.Ramamritham, and S.Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, C-34(12):1130--1141, December 1985.

[SSL89] B.Sprunt, L.Sha, and J.Lehoczky, "Aperiodic Task Scheduling in Hard Real-time Systems," *Journal of Real-Time Systems*, Vol. 1, No. 1, pp. 27-60, 1989.

[Sta96] J.A. Stankovic et al, "Strategic Directions in Real-time and Embedded Systems," *ACM Computing Surveys*, Vol. 28, No. 4, pp. 751-763, December 1996.

[SWR99] B. Shirazi, L. R. Welch, B. Ravindran, "DynBench: A Benchmark Suite for Dynamic Real-Time Systems," *Journal of Parallel and Distributed Computing Practices*, 1999, Accepted for publication, To appear.

[TD+95] T. S. Tia, Z. Deng, et al., "Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 164-173, 1995.

[TLS96] T-S. Tia, J.W.-S. Liu, and M.Shankar, "Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-Priority Preemptive Systems," *Journal of Real-Time Systems*, 10:23--43, January 1996.

[Ver95] J. P. C. Verhoosel, "Pre-Run-Time Scheduling of Distributed Real-Time Systems: Models and Algorithms," *PhD Thesis*, Eindhoven University of Technology, The Netherlands, January 1995.

[VWHL96] J. Verhoosel, L. R. Welch, D. Hammer, and E. J. Luit, "Incorporating temporal considerations during assignment and pre-run-time scheduling of objects and processes," *Journal of Parallel and Distributed Computing*, Vol. 36, No. 1, pp. 13-31, July 1996.

[WRSB98] L. R. Welch, B. Ravindran, B. A. Shirazi, and C. Bruggeman, "Specification and Modeling of Dynamic, Distributed Real-Time

Systems," *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 72 - 81, December 1998.

[WSM95] L. R. Welch, A. D. Stoyenko, and T. J. Marlowe, "Response Time Prediction for Distributed Periodic Processes Specified in CaRT-Spec," *Control Engineering Practice*, Vol. 3, No. 5, pp. 651-664, May 1995.

[XP90] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, pp. 360-369, March 1990.

Ravi Kiran Devarasetty

Ravi Devarasetty received his Bachelor of Engineering in Electronics and Communication Engineering in 1999 from Vasavi College of Engineering, Osmania University, India. After receiving his Bachelor degree, he went to Virginia Tech to pursue a Master of Science in Computer Engineering with a concentration in Computer networks. During his graduate work, Ravi worked under Dr. Binoy Ravindran in the field of real-time systems. Ravi's primary interest is in the area of computer network protocols.