Using Artificial Neural Networks for Admission Control in Firm Real-Time Systems

(HS-IDA-EA-00-107)

Magnus Thor Helgason (a97maghe@student.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

A final year project for SYP.

Supervisors: Dr. Tom Ziemke and Dr. Jörgen Hansson

Using Artificial Neural Networks for Admission Control in Firm Real-Time Systems

Submitted by Magnus Thor Helgason to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

9th of June, 2000

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: ____

Using Artificial Neural Networks for Admission Control in Firm Real-Time Systems

Magnus Thor Helgason (a97maghe@student.his.se)

Abstract

Admission controllers in dynamic real-time systems perform traditional schedulability tests in order to determine whether incoming tasks will meet their deadlines. These tests are computationally expensive and typically run in *n* * log *n* time where *n* is the number of tasks in the system. An incoming task might therefore miss its deadline while the schedulability test is being performed, when there is a heavy load on the system. In our work we evaluate a new approach for admission control in firm real-time systems. Our work shows that ANNs can be used to perform a schedulability test in order to work as an admission controller in firm real-time systems. By integrating the ANN admission controller to a real-time simulator we show that our approach provides feasible performance compared to a traditional approach. The ANNs are able to make up to 86% correct admission decisions in our simulations and the computational cost of our ANN schedulability test has a constant value independent of the load of the system. Our results also show that the computational cost of a traditional approach increases as a function of *n* log *n* where *n* is the number of tasks in the system.

Keywords: Firm Real-Time Systems, Transient Overloads, Artificial Neural Networks, Admission Controller, Overload Management.

Acknowledgements

I would like to thank my supervisors Dr. Tom Ziemke and Dr. Jörgen Hansson for giving me great support and useful comments during this project. Without them and their expertise this report would definitely not exist. I would also like to thank Jörgen for the idea of this project. Great thanks to all my friends for being supportive during my studies. I would also like to thank Ragnar Már Steinsen for being very helpful in the initial stages of this project.

Finally, I would like to thank my family in Iceland for being a great support and especially my mother and father and my two brothers for being supportive and understanding during my studies. I could not have done this with out their love and support.

Table of Contents

List of Figures

1 Introduction

Real-time systems are computer systems that have to respond to external stimuli. These systems should have two major properties, feasibility and timeliness. By feasibility we mean that there should be some guarantee that results are provided and be timeliness we mean that the results should be provided within some defined period of time. In some real-time systems the tasks in the system arrive dynamically, which means that there is no knowledge about the task characteristics before the tasks actually arrive. In such *dynamic real-time systems*, typically an admission controller is used to determine whether tasks will meet their deadlines by performing a *schedulability test* considering the current tasks in the system and the newly arrived task. If an incoming task fails the schedulability test it is rejected and we say that an *overload* has occurred in the system. On the other hand, if the task passes the schedulability test the task is accepted and no overload is said to occur.

This schedulability test takes a simple decision (accept or reject) but has a high complexity when there is a large number of tasks in the system because all current tasks have to be considered. A traditional schedulability test has a complexity of n * log n where n is the number of tasks in the system. That means, as the number of tasks in the system increases the computational cost increases as well. This might lead to arriving tasks missing their deadlines because the schedulability test takes too long time when there is a heavy load on the system.

An *Artificial Neural Network* (ANN) is a system that can be trained to approximate arbitrary functions based on labeled input-output examples. These systems differ from traditional programming approaches where the programming is done by hand. Steinsen (1999) showed that ANNs could be trained to predict up to 85% of transient overloads in dynamic real-time systems.

In this project we focus on replacing the admission controller with an ANN that performs the schedulability test. The ANN will be trained to make admission decisions using examples that are generated by a real-time simulator. The reason why ANNs are interesting in firm real-time systems is that it always takes equally much time to map inputs to outputs, i.e. in this case to get an admission decision. This could be a great advantage because a sufficiently accurate ANN could provide a schedulability test that would always have the same computational cost independent of the load of the system.

When the ANN has been trained the admission controller will be integrated into a real-time simulator and simulations will be run in order to evaluate the performance of the ANN admission controller, considering firm real-time systems. The result will be compared to a traditional approach. The timing constraints will also be considered and the time it takes to execute the ANN will be compared to a traditional schedulability test considering different number of tasks currently in the system.

1.1 Report Structure

In Chapter 2 we give some background to the problem addressed in this project and this includes a brief description of some relevant real-time and ANN concepts needed for this project. Chapter 3 follows with a detailed description of the problem and in Chapter 4 we introduce the design of our approach and which method will be used to train the ANN. Chapter 5 contains detailed descriptions of some metrics used in our experiments as well as detailed descriptions of the experiments in this project. Our results are presented in Chapter 6 and finally, Chapter 7 follows with our conclusions and discussion about our results. Some future work is also suggested in Chapter 7.

2 Background

This part of the report contains a background description. In Section 2.1 we define the most relevant real-time concepts used in this report. Section 2.2 describes different approaches for overload management in real-time systems. Finally, in Section 2.3 we give an introduction to Artificial Neural Networks (ANNs).

2.1 Real-Time Concepts

According to Burns and Wellings (1997), a *real-time system* is a computer system that has to respond to externally generated stimuli within finite and specified time limits. The correctness in real-time systems does therefore not only depend on what the results are, but also on when the results are being provided.

A good example is a nuclear powerplant. Many components of nuclear power plants are controlled by real-time systems. In case of a catastrophic accident, e.g., if one component becomes overheated, appropriate actions have to be performed within certain time limits in order to avoid any catastrophic consequences (e.g. an explosion). Otherwise, this accident might cause loss of human lives.

Tasks in real-time systems can be divided into three categories. Tasks that require regular activation are said to be *periodic* whereas tasks that have irregular arrival times are called *aperiodic.* The latter are typically used to serve random processing requirements (e.g., display activities). Finally, *sporadic* tasks are aperiodic with minimum interval times (Spuri and Buttazzo, 1996). Figure 1 illustrates the different types of task arrival.

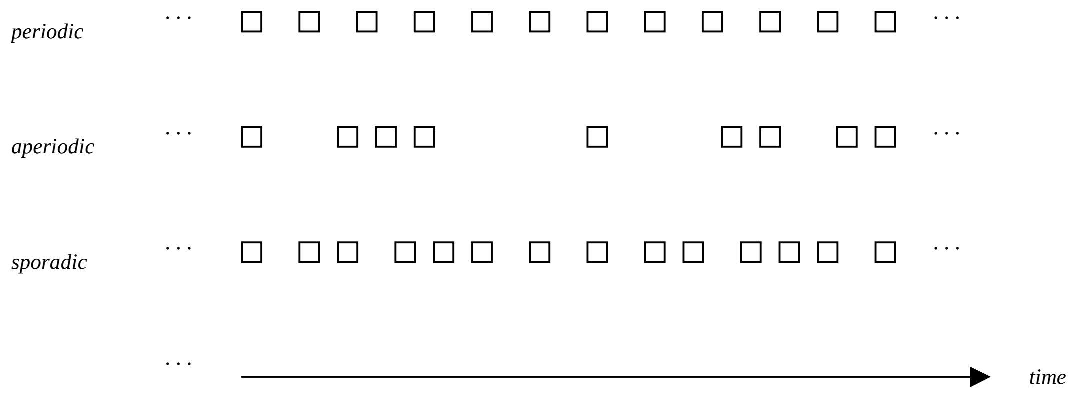

Figure 1: Tasks arrival in real-time systems.

A *deadline* is when a task has to complete its work within strict time constraints (Spuri and Buttazzo, 1996). Burns and Wellings (1997) classify deadlines into three categories:

- *Hard deadlines* can cause catastrophical consequences (e.g., nuclear powerplant meltdown) if they are missed.
- *Firm deadlines* are deadlines that are acceptable to miss occasionally, but the tasks with missed deadlines are of no value for the system, i.e. they are aborted after the deadline.

- *Soft deadlines* are deadlines that are acceptable to miss occasionally, but the tasks with missed deadlines can still be of value for the system (e.g., display activities where delays in refreshing the screen can be tolerated).

A *scheduler* is the component in real-time systems that is responsible for controlling the use of CPU and other resources (Burns and Wellings, 1997). The scheduler determines the execution order of tasks according to some scheduling algorithm that prioritizes the tasks according to some policy.

Scheduling can be either *static* or *dynamic* (Stankovic et al., 1998). In statically scheduled systems we have complete knowledge about the task set and its constraints (deadlines, execution times etc.) whereas in dynamically scheduled systems we have complete knowledge only of the currently active task set, but future new arrivals of tasks may occur without the scheduling algorithm knowing anything about it at the time when it is scheduling current tasks.

2.2 Overload Management

In this project we focus on dynamic scheduling in real-time systems where scheduling decisions are made during run-time. According to Stankovic et al. (1998), a *transient overload* is a limited period of time where tasks, arriving in the system, are not able to meet their deadlines. Transient overloads often occur in *planning-based* systems where tasks are either rejected or accepted, based on the timing constraints.

Stankovic et al. (1998) introduce three schemes that apply to overloads in the planning-based scheduling approach. These schemes are illustrated in figure 2.

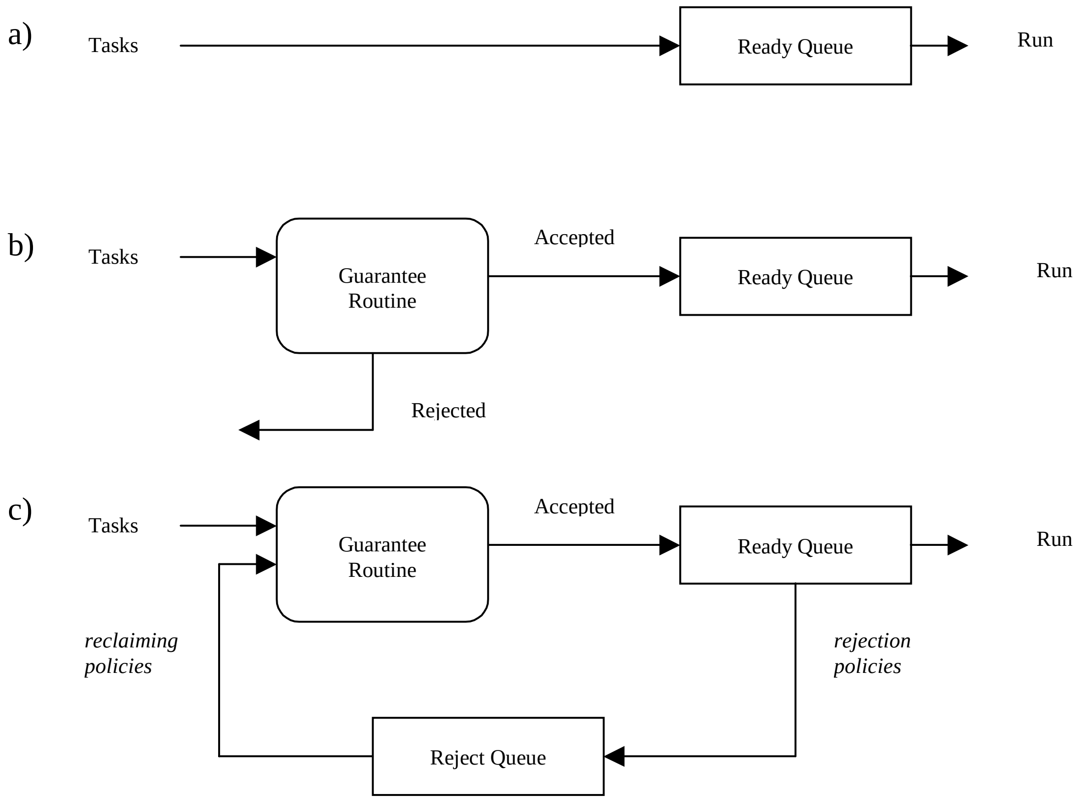

Figure 2: Scheduling schemes for overload situations.

Figure 2a illustrates the *best effort* scheme. All tasks that arrive to the system are accepted and put in the ready queue.

Figure 2b illustrates the *admission control* scheme. All tasks that arrive in the system are run through a schedulability test where the current load on the system is considered. If a task passes the schedulability test it is said to be *accepted* and is inserted into the ready queue. On the other hand, if a task fails the schedulability test, it is *rejected*.

The final approach, illustrated in figure 2c, is called the *robust* scheme. In this case two policies are being considered, a schedulability test and task rejection. When a task arrives in the system, it is accepted if it passes the schedulability test and is placed in the ready queue. If a task fails the schedulability test, one or more tasks currently in the system are being cancelled based on different policies.

Stankovic et al. (1998) define the RED (Robust Earliest Deadline) Algorithm for handling overload situations. This algorithm is discussed in detail in Section 2.2.1.

2.2.1 The RED (Robust Earliest Deadline) Algorithm

The RED algorithm is designed to handle firm aperiodic tasks in environments where overloads can occur (see figure 2c). This algorithm combines various features like graceful degradation in overload situations, deadline tolerance and resource reclaiming. Using this approach makes it possible to predict tasks missing their deadlines, and makes it even possible to estimate the size of the overloads and the impact on the system.

C_i = worst execution time

D_i = relative deadline

M_i = deadline tolerance

V_i = importance value

L_i = residual laxity

E_i = exceeding time

t_i= estimated finishing time

T = set of tasks

T_i = task i in the set of tasks T

Figure 3: RED definitions

In figure 3 some definitions are listed. In RED each task instance $T_i(C_i,D_i,M_i,V_i)$ is characterized by four values. C_i is the *worst execution time*, i.e. the maximum time it takes to execute the task. D_i is the *actual deadline* (primary deadline) and M_i is the *deadline tolerance* (secondary deadline) which is the amount of time that a task is allowed to be late. V_i represents the importance of task T_i.

The schedulability test is formulated in RED using the *residual laxity* L_i. This value is the interval between the estimated finishing time (t_i) and the primary deadline (D_i) of the task T_i. L_i is calculated as follows:

$$L_i = L_{i-1} + (D_i - D_{i-1}) - C_i(t)$$

The proof of this lemma can be found in Stankovic et al. (1998, p. 104).

To describe the schedulability test, the exceeding time E_i is defined as the time that task T_i executes after its secondary deadline. E_i is defined as follows:

$$E_i = \max(0, -(L_i + M_i))$$

Considering the whole task set T, E_{max} is the maximum exceeding time among all E_i in T, i.e. $E_{max} = \max_i(E_i)$. A schedule is said to be *strictly feasible* if and only if $L_i \geq 0$ for all tasks in the set T. It is said to be *tolerant* if and only if there exists some $L_i < 0$, but $E_{max} = 0$. If the schedule is strictly feasible then all tasks complete their execution before the primary deadline (D_i). On the other hand, if the schedule is tolerant, then tasks execute after the primary deadline and before the secondary deadline. The RED schedulability test algorithm is listed in figure 4.

```
Algorithm RED_schedulability_test($T$,$T_{new}$)
{
    $E = 0$;
    $L_0 = 0$;
    $D_0$ = current_time();

    $T' = T \cup \{T_{new}\}$;
    $k$ = < position of $T_{new}$ in the task set $T'$>;
    for each task $T_i'$ such that $i \geq k$ do {
        /*compute the maximum exceeding time */
        $L_i = L_{i-1} + (D_i - D_{i-1}) - c_i$;
        if ($L_i + M_i < -E$) then
            $E = -(L_i + M_i)$;
    }
    if ($E > 0$) then {
        <select a set $T^*$ of least value tasks to be rejected>;
        <reject all tasks in $T^*$>;
    }
}
```

Figure 4: The RED schedulability test (Stankovic et al. 1998, p. 105).

The aim of the schedulability test is to determine if a task T_{new} is likely to miss its deadline. If that is the case, the amount E_{max} of additional processing time is computed. Otherwise the task is accepted and is executed according to the EDF scheduling policy.

Using the E_{max} value, from a global point of view, makes it possible to plan an action in order to recover from overload conditions that would occur if the task was accepted. One approach is to reject the task that has the least importance value V_i and which can remove the overload situation. Rejected tasks are not removed from the system, unless a task has a negative laxity value. Otherwise the tasks are inserted in a reject queue ordered by decreasing values. Whenever a running task happens to complete its execution before the worst case finishing time the RED algorithm tries to reaccept the highest value tasks in the reject queue.

2.3 ANN Concepts

An Artificial Neural Network (ANN) is a mathematical model of the operation of the brain where mathematical computing elements correspond to neurons which are cells building a network where each cell takes part in performing information processing in the brain (Russell and Norvig, 1995).

ANNs have been used to solve different kind of problems, e.g., learning pronunciation, character recognition, and learning how to steer a car (Russell and Norvig, 1995). When e.g. learning how to drive the ANN is trained to make decisions about whether to turn left or right and also how much the steering wheel should be turned.

2.3.1 The Artificial Neuron

Figure 5 illustrates two types of ANNs, a feed forward network and a recurrent network. Each network consists of layers of nodes. The nodes in the output and hidden layers are artificial neurons that perform mathematical operations in order to provide output values (see figure 6) and the input nodes in the ANNs are nodes that do not perform any mathematical operations, the output values provided by these nodes are fixed.

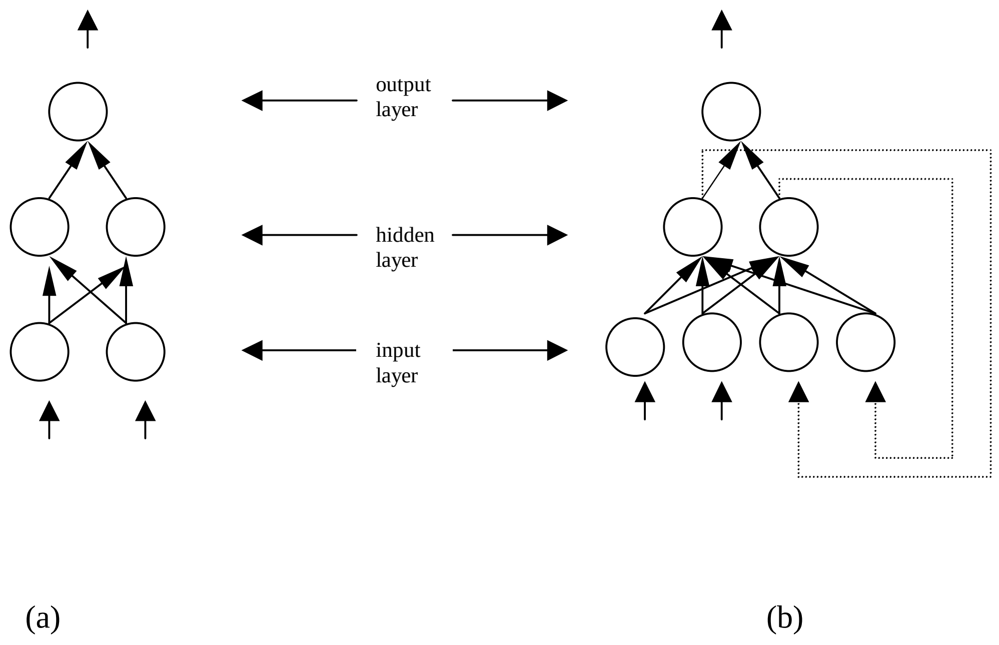

Figure 5: (a) Standard feed-forward network. (b) Recurrent network

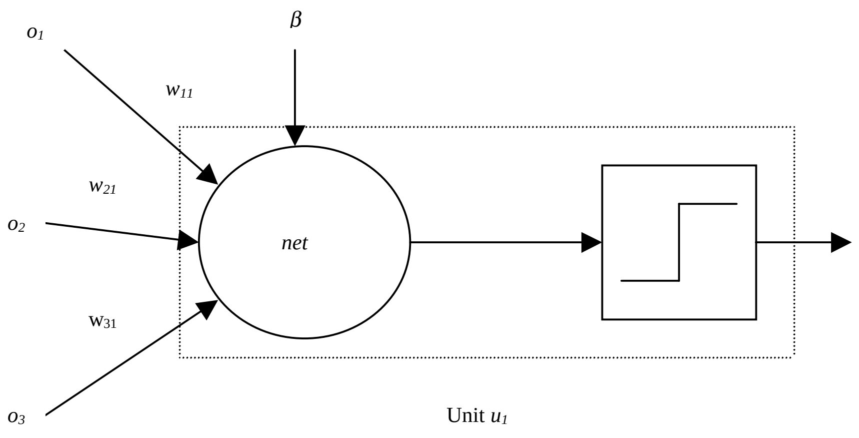

Figure 6: Artificial Neuron.

A node that performs mathematical operations uses outputs from a previous layer as well as the weights that connect all the nodes in the previous layer to the current node, in order to calculate the output value. Figure 6 illustrates the structure of an artificial neuron in an ANN.

In figure 6 w_{ju} is the numeric value of the weight between units *j* and *u*, e.g. w_{21} represents the weight between node j_2 in the previous layer and unit u_1. o_j represents the output from unit *j* in the previous layer.

To calculate the value of *net* the following equation is used where β is the *bias*, a special value used to gain correctness that can be viewed as a weight of a connection from a node that always has full activation and can be seen as setting the threshold for whether the activation will be high or low.

$$net_u = \beta + \Sigma o_j w_{ju}$$

The output of unit *u* is calculated by running the value of *net* through an *activation function* in order to get a value between zero and one. The S-shaped sigmoid function is commonly used as an activation function (see figure 7).

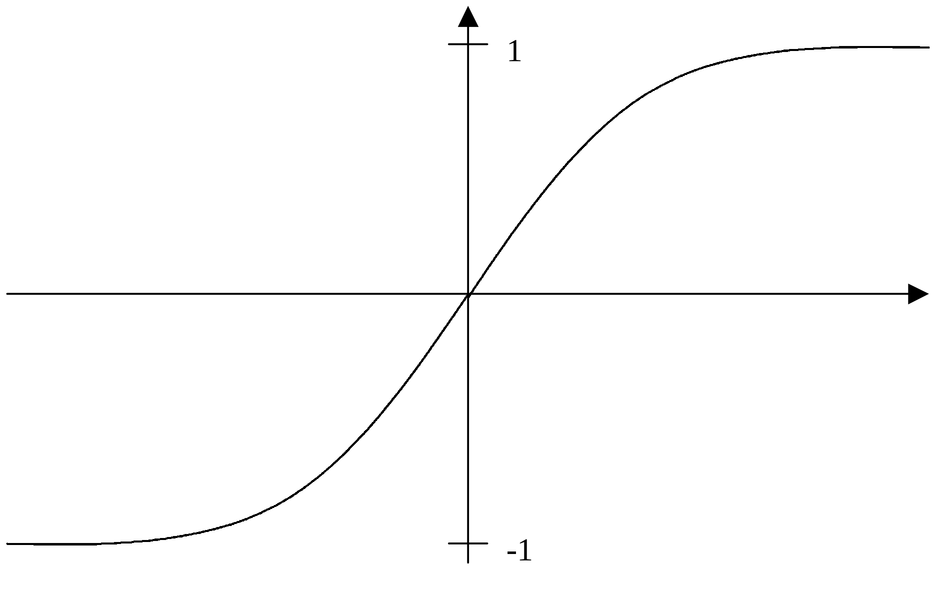

Figure 7: The S-shaped sigmoid function.

The number of nodes and layers in ANNs can vary depending in the problem to be solved. There are no methods for how to construct the optimal network (Russell and Norvig, 1995).

Figure 5 illustrates two types of ANNs, a standard feed forward network and a recurrent network. The main difference between those networks is that recurrent networks use hidden node activation from the previous timestep as additional inputs in current timestep. By using recurrent networks we are therefore able to have a memory function in the network, i.e. to use results from a previous timestep.

2.3.2 Supervised Learning in ANNs

In feed-forward networks a supervised learning method called back-propagation is the most commonly used (Russell and Norvig, 1995). A set of training data, i.e. a set of input vectors and corresponding correct output vectors, is used to train the network. One *epoch* means that all examples in the training set are run through the network and the weights are adjusted after each example has been fed forward. To evaluate the performance of the network, some test data is also used. By checking the error in the learning set after feeding each example in the training set forward and comparing it to the error in the test set we are able to monitor the learning process. This error is called the Mean Squared Error (MSE). We say that a network *overfits* the training data when the MSE of the test set begins to increase after a number of epochs. When the MSE in the test set is similar to the MSE in the training set (and both values are low) we say that the network *generalizes*. Figure 8 illustrates how the error typically changes during the learning process.

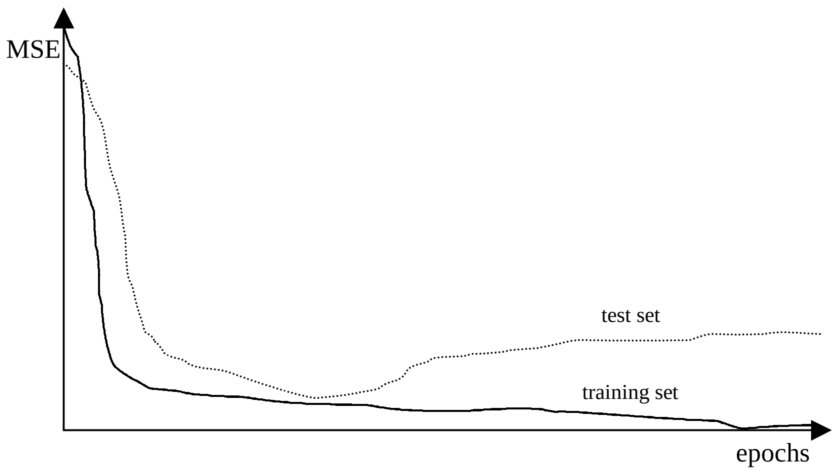

Figure 8: Typical learning process in ANNs.

The back-propagation algorithm basically works as follows. After each feed-forward action the error (e_u) is computed for each computational node. This error value is based on the target in the training example that was used to feed forward the input data. When the error has been computed the weights are updated using the following learning rule.

$$\text{update value for } w_{ju} = e_u * o_j * \eta$$

The learning rate (η) controls the speed of the weight correcting process. The higher the rate the more the weights are updated in each step. Still, using too low learning rate can result in never getting the optimal weight value due to the risk of getting stuck in a local optima (Russell and Norvig, 1995).

2.4 Related Work

In this section we describe Steinsen's (1999) approach to prediction of transient overloads in real-time systems using ANNs. Using this method it was shown that up to 85% of transient overloads could be predicted.

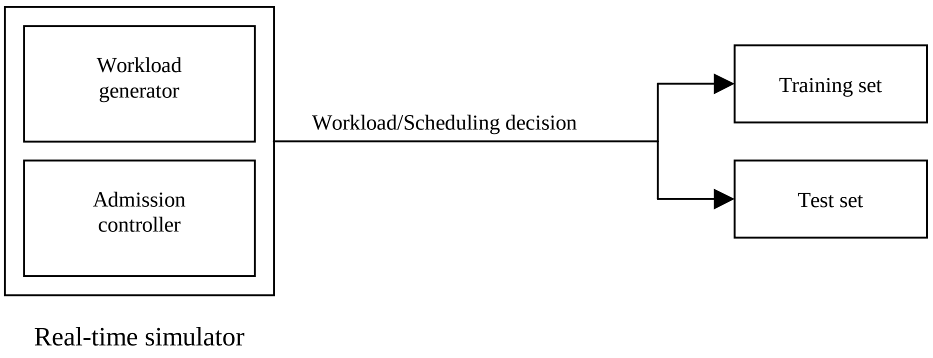

Figure 9: How the learning set and training set is generated.

Figure 9 shows how the training and test sets is generated. A part of a real-time simulator (RADEx++) is used to generate workload files that consist of a scenario of tasks with corresponding scheduling results for each task. RADEx++ is a real-time database simulator developed at the University of Massachusetts and extended by Hansson (1998). For more information about RADEx++ the reader is referred to Sivasankaran et al. (1995).

The training approach consists of using as input to the ANN a sliding window of information about a constant number of recently arrived tasks (see figure 10). When new task arrives it is added to the window and the oldest tasks in the window is removed and the information is propagated through the network which should predict if any of the tasks in the prediction window will cause overload. Each task is represented by a temporal scope that contains the task information (task characteristics, i.e. arrival time, deadline and worst-case execution time). The sliding window also contains some global variables that refer to the current tasks that are sliding in the window, such as the total deadline time minus arrival or total worst case execution time.

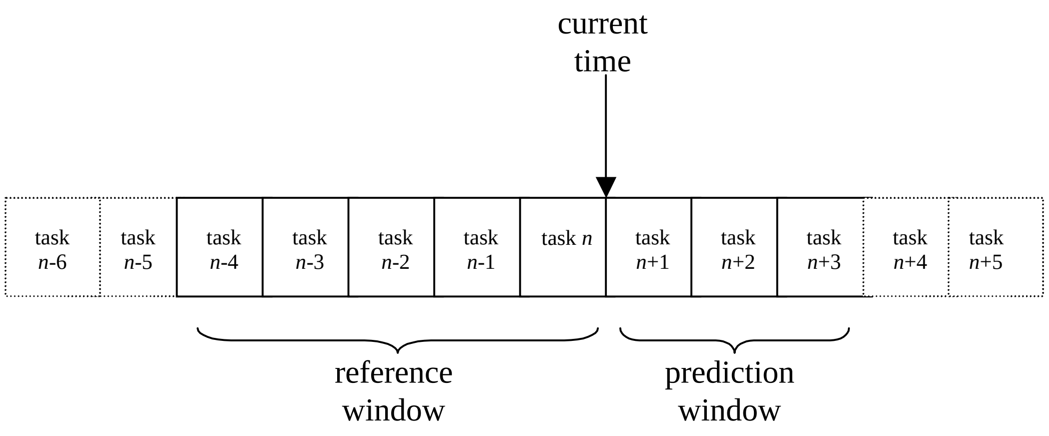

Figure 10: Reference and prediction window.

Hence, when training the ANN in order to predict transient overloads, Steinsen (1999) used two sliding windows, a *reference window* and a *prediction window*. The prediction window contains tasks that occur in the future and are to be considered when predicting transient overloads. Figure 10 shows a sliding window with a reference window of size 5 and prediction window of size 3.

Figure 10 illustrates the sliding window when task *n* arrives. tasks *n*-1, *n*-2, *n*-3 and n-4 are newly arrived tasks. Tasks *n*+1, *n*+2 and *n*+3 are tasks in the prediction window that will arrive in the future. If e.g. task *n*+3 causes an overload then this should be signaled when task n arrives. On the other hand, if task *n*-1 is arriving then task *n*+3 is not part of the prediction window and no overload should be signaled. When predicting overloads the prediction window is used when training the ANN in order to classify which workload scenarios could cause overloads in the nearest future.

One of the attributes in the temporal scope is the number of empty time-steps. Figure 11 illustrates an example of how preceding empty time-steps are counted. The number of empty time steps for task *n*-2 is 1 and number of empty time-steps for task *n*-1 is zero. On the other hand, the number of empty time-steps for task *n* is 2. The size of each time-step controls how many clock tics each step consists of. Figure 11 illustrates time steps of size 10. The algorithm for counting empty time steps can be found in Appendix A.

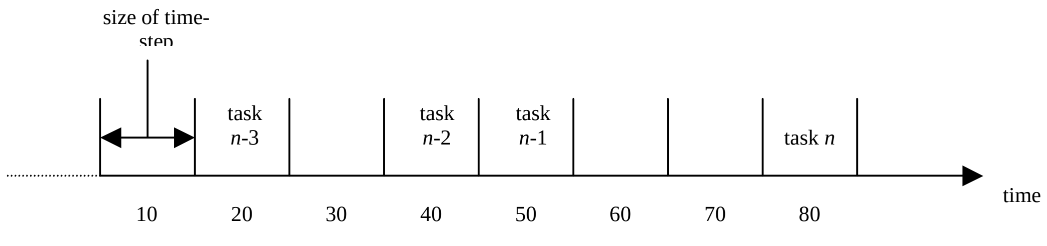

Figure 11: Time-steps.

Figure 12 shows in detail how the sliding window works and how the values are presented to the ANN. In this case we have a temporal scope that consists of the following attributes.

- deadline time minus arrival time (d-a)
- worst-case execution time (w)
- empty time-steps

id	Task *n*			d-a	w	empty time-steps	Task *n*-2			Task *n*-3			Σ(d-a)	Σw	target output
35	1265,9	150	15	1268,3	150	14	1256,4	120	3	1361	150	1	5151,6	570	0
36	1147,1	130	10	1265,9	150	15	1268,3	150	14	1256,4	120	3	4937,7	550	0
37	1177,5	120	9	1147,1	130	10	1265,9	150	15	1268,3	150	14	4858,8	550	0
38	1348,4	150	3	1177,5	120	9	1147,1	130	10	1265,9	150	15	4938,9	550	0
39	981,8	120	14	1348,4	150	3	1177,5	120	9	1147,1	130	10	4654,8	520	1

Local task attributes

Global attributes

Figure 12: The training set.

In this case we have a reference window of size 4 and two global attributes, Σ(d-a) (total deadline time minus arrival time) and Σw (total worst-case execution time). The number of input attributes are therefore (3*4) + 2 = 14. One line in figure 12 represents one example in the training set. When task 35 arrives it is put in the first position in the input vector and when task 39 arrives, task 35 is removed from the window. The gray cells show how a task moves through the sliding window as it arrives in the system. When task 39 arrives we have a target value of 1. This means that task 39 should be rejected. When the target value is 0 then task 39 is accepted.

3 Problem Description

This chapter contains a detailed description of the problem we will be dealing with in this project. In Section 3.1 we motivate the problem, Section 3.2 describes our objectives and, finally, Section 3.3 presents assumptions adopted in this project.

3.1 Motivation

Figure 13 illustrates a planning-based architecture for real-time systems (Stankovic et al., 1998). Tasks arrive in the admission controller which runs a schedulability test to determine whether tasks will meet their deadlines or not. A task is rejected and put into the reject queue if it fails the schedulability test. If it passes the schedulability test, it is sent to the scheduler for execution. When a task is put in the reject queue, an overload is said to occur in the system.

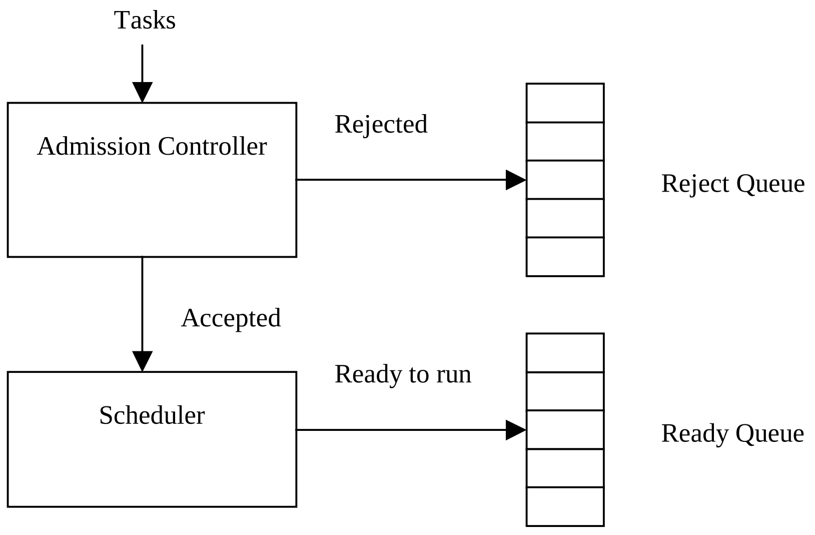

Figure 13: The planning-based architecture.

Stankovic et al. (1998) give several viewpoints on this architecture. Firstly, since real-time systems must provide graceful degradation in case of failures, transient overloads must be detected as soon as possible to prevent the consequences. However, in the scheme illustrated in figure 13 an overload cannot be detected until it occurs in the system. By then it can be too late to prevent the consequences. Secondly, the schedulability test is computationally intensive. The earlier an arriving task is run through the schedulability test, the sooner it is known whether it will meet its deadline. According to Steinsen (1999) a traditional schedulability test in a planning-based architecture runs in O(*n* log *n*) time where *n* is the number of tasks in the system. For large values of *n*, which are scenarios that present overloads, the computational cost of the schedulability test is very high. In this project we focus on overcoming the second problem, i.e. the computationally expensive schedulability test.

3.2 Objectives

Our work extends Steinsen's (1999) work. In contrast, in this project we aim for using ANNs for admission controller. He showed that, for certain workloads, the ANNs were able to predict up to 85% of transient overloads in dynamic real-time systems. Given that the ANN provides feasible results, the traditional schedulability test that is performed by the admission controller (see figure 13) can be replaced with an ANN that performs the test.

The reason why ANNs are interesting is that running an example through an ANN, i.e. feeding data forward from the input nodes to the output nodes in order to get results, always takes the same amount of time to execute. By using an ANN to perform the schedulability test, the computational cost of performing such a test would always be the same, or $O(c)$ where c is a constant, regardless of the load of the system.

n	$O(n \log n)$
1	0
5	3.49
10	10.00
15	17.64
20	26.02
25	34.95
.	.
.	.
.	.
100	200

a) Traditional approach.

n	$O(k)$
1	c
5	c
10	c
15	c
20	c
25	c
.	.
.	.
.	.
100	c

b) ANN approach.

Figure 14: Complexity of the schedulability test.

Figure 14 shows the complexity of a traditional approach and an approach where an ANN would be used. The number of tasks in the system is n at the time when the schedulability test is executed. The computational cost of the traditional approach increases as the number of current tasks in the system increases whereas the cost of using an ANN for the same purpose would always be the same. Given that the computational cost c is significantly smaller than the cost of using a traditional approach when there is a heavy load on the system the, i.e., for large values of n, the consequences of an overload will be reduced due to the reduced computational cost of the schedulability test when a task arrives.

In order to determine this, we will design a new admission controller that uses an ANN for performing the schedulability test and compare the time it takes to perform a schedulability test using the ANN approach to the time it takes to use the traditional approach that runs in $n \log n$ time, considering tasks with firm deadlines since they are simply cancelled if they do not meet their deadlines.

3.3 Assumptions

In this section we define the assumptions adopted in this project.

- Dynamic scheduling will be used since transient overloads only occur in dynamic systems (see Section 2.1).
- Earliest Deadline First (EDF) policy will be used to schedule tasks. This is a well-known and widely used policy (Stankovic et al., 1998).
- Tasks are sporadic because in a periodic workload there are no overloads and in an aperiodic workload no guarantees of the task information can be made (see Section 2.1).
- Tasks are mutually independent and have no precedence constraints.
- Tasks have firm deadlines because ANNs are not always (but can be) 100% correct and in firm real-time systems it is acceptable if deadlines are occasionally missed.
- The system has one CPU which is the only type of resource that is considered.
- The training method presented by Steinsen (1999) will be used, and the following task characteristics will be considered:
    - arrival time, i.e. when tasks arrive to the system
    - deadline, i.e. the time when tasks has a deadline
    - worst-case execution time.
- Recurrent ANNs will be used. Steinsen (1999) showed that recurrent ANNs and feed forward ANNs give almost equally good generalization and when using a small reference window (see Section 2.4) the recurrent network gave slightly better generalization.

4 Using ANNs for Admission Control

In this chapter we will introduce a new admission controller using ANNs. Section 4.1 presents the overview of our system, Section 4.2 gives an overview of our implementation and, finally, in Section 4.3 we discuss how the ANN is trained.

4.1 Design Overview

Figure 15 illustrates the planning-based architecture discussed in Section 2.2 where a traditional admission controller has been replaced with one that uses an ANN to make its decisions.

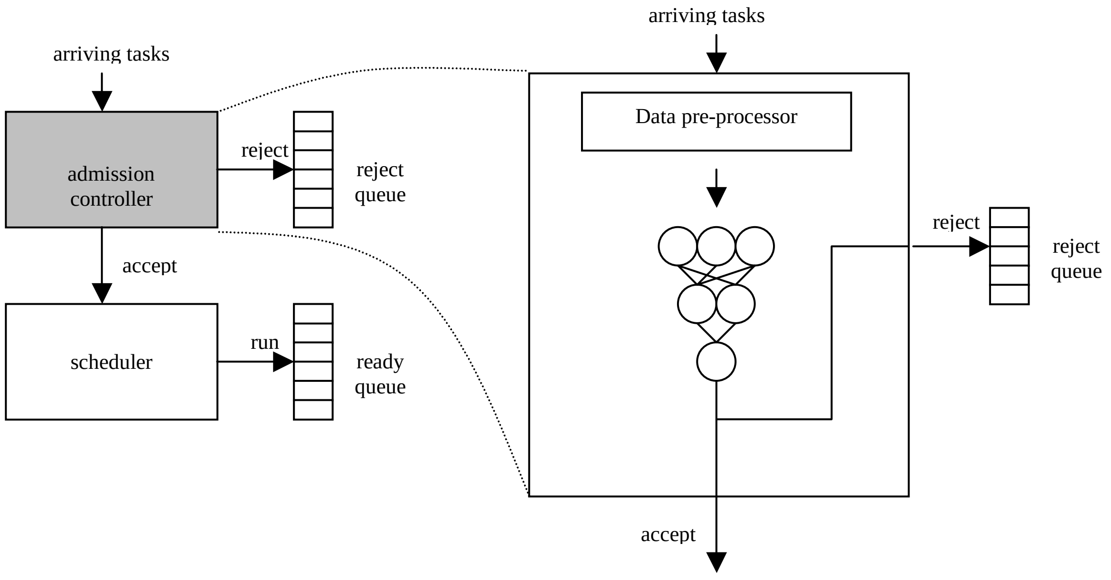

Figure 15: A schematic illustration of the ANN admission controller.

When a task arrives at the admission controller the task information is being pre-processed. This involves putting the task information about recent tasks into a workload array that contains values that are used as inputs for the ANN. The aim with the data pre-processor is not only to maintain information about recent tasks in the system, but also to normalize the values to values between zero and one.

When the task information has been pre-processed the values of the workload array are fed through a trained ANN (the training is discussed in Section 4.3). The admission decision is based on a rounded value of the ANN output. When using ANNs the threshold (*R*) is usually 0.5. This means that if the output value is e.g. 0.3 then we round the value down to 0. On the other hand, if the output value is 0.6 we round the value up to 1. The threshold *R* is defined to be in the following interval:

$$]\,0,1\,[$$

The function $reject_i$ is defined as:

$$reject_i = \begin{cases} 1 \text{ if } (outputANN_i \geq R) \\ 0 \; otherwise \end{cases}$$

and $outputANN_i$ is the output from the ANN when making the admission decision for task k. If the value of $reject_i$ is one then the task is rejected, otherwise it is accepted.

To give a formal description of the ANN schedulability test the algorithm is listed in figure 16.

```
Algorithm ANN_schedulability_test(workloadArray,ANNInputArray,$T_{new}$)
{
    rejectTask = false;
    outputANN = 0.0;
    threshold = 0.5;
    $s_w$ = <size of workloadArray>;
    $s_a$ = <size of ANNInputArray>;
    <remove oldest task in workloadArray>;
    for each position p in workloadArray such that $p \leq s_w$ do {
        <put task in position p in workloadArray into position p+1>;
    }
    <put task $T_{new}$ in first position in workloadArray>;
    for each position p in workloadArray such that $p \leq s_w$ do {
        <compute global variables>; /* see Section 2.4 for details */
    }
    for each position p in ANNInputArray such that $p \leq s_a$ do {
        ANNInputArray[p] = workloadArray[p];
    }
    <feed forward ANN>;
    if (outputANN ≥ threshold) then {
        rejectTask = true;
    }
    return rejectTask;
}
```

Figure 16: The ANN schedulability test.

4.2 ANN Training Approach

In our project we modify Steinsen's (1999) approach described in Section 2.4. Since we will not be predicting overloads but performing schedulability tests the prediction window is not used. We only keep one window in order make admission decision.

Figure 17 illustrates the sliding window used in our approach. If task *n* causes an overload it is rejected.

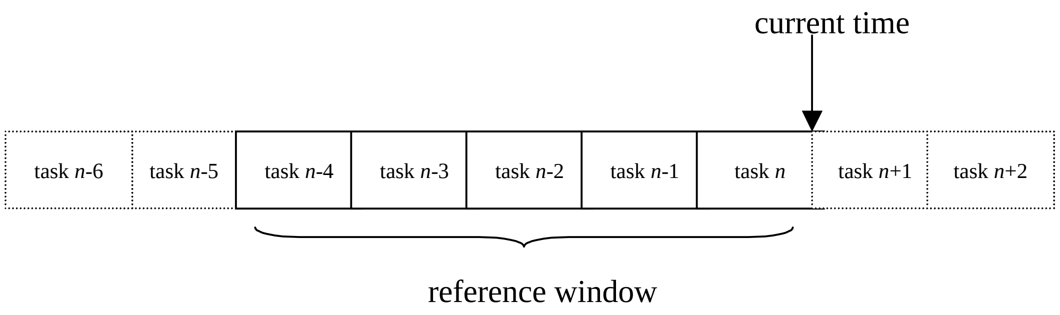

Figure 17: Reference window.

As mentioned earlier in Section 2.4, each task is presented using a temporal scope which contains some attributes that represent the task information. We define the following attributes.

- *d* – deadline
- *a* – arrival time
- *w* – worst-case execution time
- *t* – empty time-steps

d is the time where a task has a deadline, *a* is the arrival time of a task, *w* is the worst-case execution time for a task and finally, *t* is the number of empty time-steps before tasks enters the system at time *a*. The following attributes present the temporal scope in our project:

- *d-a*
- *w*
- *t*

As mentioned in Section 2.4 each training example contains some global attributes as well. In our approach we will use the following attributes.

- Σ (*d-a*)
- Σ *w*

where Σ (*d-a*) contains the total (*d-a*) in reference window and Σ *w* contains the total worst-case execution time.

5 Performance Evaluation

Section 5.1 contains a detailed description some metrics for our experiments and Section 5.2 includes a thorough description of our experiments.

5.1 Metrics

In order to evaluate the performance we will focus on the following three aspects:

- ANN generalization
- Real-time simulation performance using an ANN admission controller
- Time constraints

When evaluating the generalization of ANNs the MSE (defined in Section 2.3.2) is usually considered. But since we are working with an admission controller in real-time systems we define some other attributes in order to evaluate the performance. The task of the admission controller is to decide whether to reject a task or not. Since we are using an ANN for admission controller we cannot guarantee that all decisions will be correct. The admission controller might accept a task that otherwise would be rejected by a traditional admission controller. This could mean that this particular task would not finish its execution. On the other hand, the ANN admission controller might reject a task that otherwise would be accepted by a traditional admission controller. This would affect the utilization in the system because tasks that are able to finish their execution are not being executed which provides idle processor time.

In order to evaluate the performance of the ANN we will define some additional attributes. In Section 4.2 we defined the constant *R* as the threshold value. The function *correct*$_i$ is defined as:

$$correct_i = \begin{cases} 1 \text{ if } ((outputANN_i \geq R \wedge targetANN_i = 1) \vee \\ \quad (outputANN_i < R \wedge targetANN_i = 0)) \\ \\ 0 \; otherwise \end{cases}$$

where *outputANN*$_i$ is the output from the ANN for task *i* and *targetANN*$_i$ is the target value for task *i*. When the ANN accepts tasks *i* which should be rejected we define the function *falseAccepted*$_i$ as follows:

$$falseAccepted_i = \begin{cases} 1 \text{ if } (outputANN_i < R \wedge targetANN_i = 1) \\ \\ 0 \; otherwise \end{cases}$$

The final attribute is used when the ANN rejects task *i* that should be accepted. The function *falseRejected*$_i$ is defined as follows:

$$falseRejected_i = \begin{cases} 1 \text{ if } (outputANN_i \geq R \wedge targetANN_i = 0) \\ 0 \text{ } otherwise \end{cases}$$

All the attributes defined above relate to experiments where test sets are used to evaluate the classification and training of the ANN. To evaluate the simulation, i.e. after the ANN admission controller has been integrated into to RADEx++, we use values that are provided by RADEx++. We are interested in knowing how many of the tasks that are accepted actually finish their execution in time. To evaluate this we use the percentage value *c*, which is defined as follows.

$$c = (\text{number of completed tasks /number of accepted tasks}) * 100$$

The aim is to keep the value of *c* as high as possible. If the value of *c* is low, then the admission controller often accepts tasks that cannot finish their execution in time.

To measure the time it takes to perform the schedulability test we use the function $decisionTime_i$. This is defined as follows:

$$decisionTime_i = (endTime_i - startTime_i) / clocksPerSecond$$

where $startTime_i$ is the time when performing a schedulability test for task *i* begins and $endTime_i$ is the time when a schedulability test for task *i* has been performed. *clocksPerSecond* is the number of clock tics per second.

5.2 Experiments

When training and testing the ANN we use different workload scenarios. The following attributes are used to identify both the workload training scenarios and workload testing scenarios.

$trWo_j$ where $j \in \mathbf{N}$ and

$teWo_j$ where $j \in \mathbf{N}$

where $trWo_j$ is a workload scenario with the identity number *j* used to train an ANN and $teWo_j$ is a workload scenario with the identity number *j* used to test an ANN.

In our experiments we use six types of training scenarios and six test scenarios respectively. The difference between the scenarios lies in the arrival rate (transaction/second). Table 1 lists the training scenarios and table 2 lists the test scenarios, generated by RADEx++ (see section 2.4 for details about generation).

	Accepted tasks	Rejected tasks	Arrival rate (trans/sec)
$trWo_1$	100 %	0 %	5
$trWo_2$	80,5 %	19,5 %	8
$trWo_3$	70,8 %	29,1 %	9
$trWo_4$	63,7 %	36,3 %	10
$trWo_5$	30,9 %	69,1 %	20
$trWo_6$	64,8 %	35,2 %	varies

Table 1: Learning workload scenarios.

	Accepted tasks	Reject tasks	Arrival rate (trans/sec)
$teWo_1$	100 %	0 %	4
$teWo_2$	80,4 %	19,6 %	8
$teWo_3$	70,9 %	29,1 %	9
$teWo_4$	63,6 %	36,4 %	10
$teWo_5$	30,7 %	69,3 %	20
$teWo_6$	64,7 %	35,3 %	varies

Table 2: Testing workload scenarios.

As tables 1 and 2 illustrate, $trWo_6$ and $teWo_6$ do not have fixed arrival rates. In these scenarios the arrival rate ranges from 8 to 12.

Our experiments can be categorized into the following categories:

- ANN generalization experiments
- Real-time simulation experiments using an ANN admission controller
- Timing constraint experiments

These experiments are described in detail in Sections 5.2.1 – 5.2.4.

5.2.1 ANN Generalization I

The ANN classification experiments consist of training and testing the ANN using different workload scenarios and using different types of reference windows, i.e. windows with different sizes. The aim with our first experiment is to see whether there is any optimal size for the reference window, and perhaps some upper and/or lower bounds can be found. The classification results are used to evaluate these matters. When doing these kind of experiments we use training and testing workloads 1-5. These scenarios differ a lot and are therefore suitable for these experiments. We also take a look at the MSE in each experiment. In all cases we use scenarios of the same type for both training and testing, e.g. if an ANN is trained with $trWo_3$ it will be tested with $teWo_3$ etc.

5.2.2 ANN Generalization II

Experiments are done where an ANN will be trained using one type of workload scenario and then tested with another type of workload scenario, e.g. training with a workload with a constant arrival rate and then tested with a workload with an arrival rate that varies. Table 3 outlines this experiment.

training arrival rate	constant	constant	varies	varies
testing arrival rate	constant	varies	constant	varies

Table 3: Outline of an ANN experiment.

When the arrival rate is constant we will use $trWo_2$ and $teWo_2$ respectively but if the arrival rate varies we will be using $trWo_6$ and $teWo_6$ respectively. The aim with this experiment is to see if a trained ANN is able to handle situations where e.g. the arrival rate might vary or be constant over a workload.

5.2.3 Real-time Simulation Experiments using an ANN Admission Controller

Experiments are done where a traditional admission controller in a real-time simulator (RADEx++) is actually replaced with our approach that uses ANN for making decisions. In these experiments a trained ANN with the training workload that gave the lowest MSE in previous experiments is used (see Section 5.2.1). The size of the reference window will depend on the results from these other experiments as well and 3 of the best windows will be used for these experiments, in order to get some comparison. The aim with these experiments is to see how well the ANN admission controller really works and this is evaluated using the attributes defined in Section 5.1.

5.2.4 Time Constraint Experiments

In order to see whether $decisionTime_i$ (see definition in Section 5.1) for the ANN admission controller has a constant value and that $decisionTime_i$ for a traditional approach has a maximal value according to the function $n * \log n$ as the number of current tasks (n) increase in the system, experiments are done where $decisionTime_i$ is measured for both approaches.

Since the RED schedulability test (described in Section 2.2.1) is an approach that is commonly used, we implement a simulator that runs a schedulability test when tasks arrive in the system. The reason why RADEx++ is not be used for these experiments is that RADEx++ contains structures that might cause less accuracy in measuring the time. We therefore implement the RED schedulability test outside RADEx++ using workload scenarios that previously have been generated by RADEx++ and only perform the actual schedulability test. Figure 18 illustrates the overview of this implementation.

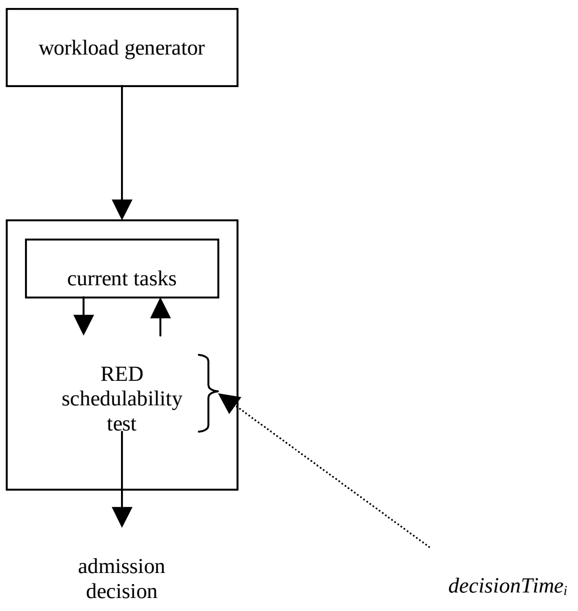

Figure 18: Outline of the implementation of the RED schedulability test outside of RADEx++.

The workload generator uses files that contain workload scenarios generated by RADEx++ and the simulator maintains the tasks in the system using a simulated clock. The clock is simply an increasing counter used to simulate the current time in the system. The tasks are then added or removed from the system according to the task information provided by the workload generator and the current time. By doing this we only have tasks in the system that are being executed. When measuring the time it takes to perform the RED schedulability test we use the metrics defined in Section 5.1. When measuring the time it takes to perform the schedulability test using an ANN we simply measure the time it takes to feed forward data using the same metrics.

6 Results

In this chapter we will present our results of this project. Section 6.1 and 6.2 contain the ANN generalization results, Section 6.3 contains simulation results using the ANN as an admission controller and finally, Section 6.4 contains the timing results.

In all ANN generalization experiments the ANN contained two hidden nodes. The number of input nodes depended on the number of tasks in reference window (see Section 2.4 for details about reference window). Each experiment was done three times with different number of epochs (from 1,000 epochs up to 20,000 in some cases) in order to get the best training in each case. In all experiments the learning rate (η) was set to 0.02. The training set contained 5,000 examples in all cases and the test set contained 2,000 examples.

6.1 ANN Generalization Results I

Figures 19-23 present the results from training an ANN with one type of workload scenario and testing it with a scenario of the same type. Details for the workload scenarios can be found in table 1 and table 2. Figure 19 illustrates the results from training an ANN with *$trWo_1$* and testing it with *$teWo_1$*. These scenarios do not contain any overloads and it is easy for the ANN to classify the tasks in the scenario. Consequently, all tasks are correctly classified. All tasks are accepted and no overloads occur, hence the size of reference window is not critical in this case.

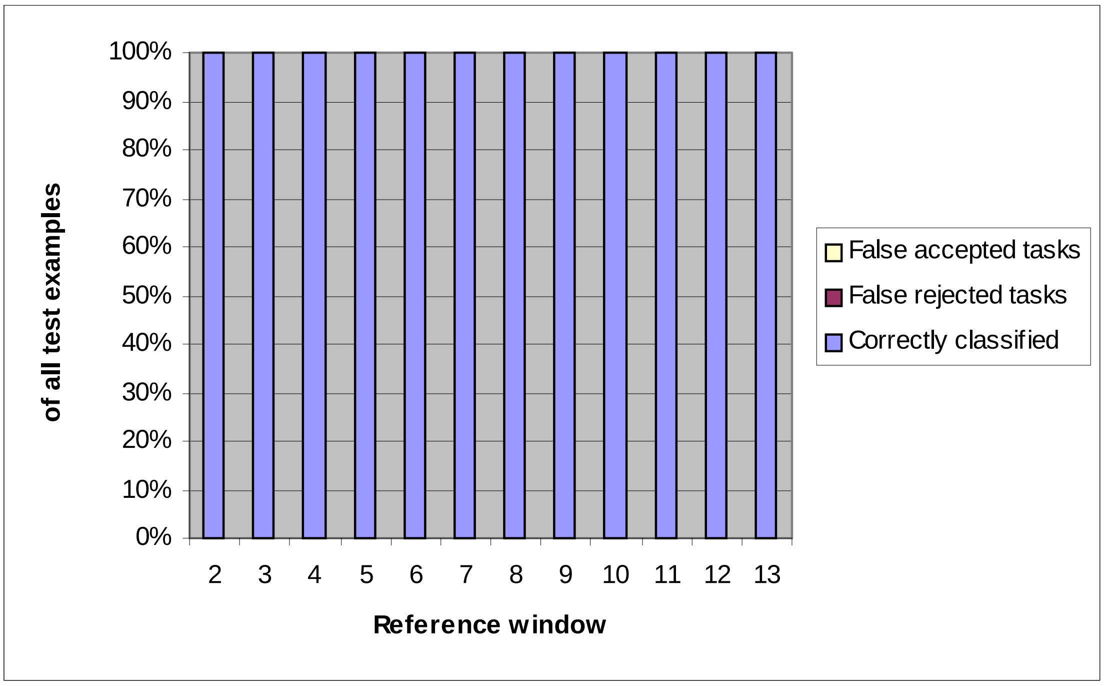

Figure 19: Classification results from training an ANN with *$trWo_1$* and testing with *$teWo_1$*.

Figure 20 illustrates the classification results from training an ANN with *$trWo_2$* and testing it with *$teWo_2$* respectively. In this case the upper bound for the size of reference window is 13 in which case the ANN does not detect any overloads at all. Other reference windows gave 81%-86% correct results where using reference window of size 4 gave the largest number of correctly classified tasks. False accepted tasks vary from 10% up to 19% and false rejections vary from 3% up to 6%. The number of false accepted tasks is therefore considerably more than number of false rejections.

When training the ANN using *trWo*$_3$ and testing it with *teWo*$_3$ we get the same upper bound for the size of reference window as when using *trWo*$_2$, i.e. 13. Figure 21 illustrates these classification results. When increasing the size of reference window we get higher number of correctly classified tasks and the number of false accepted tasks decrease at the same time. In this case the correct results vary from 71% up to 79%, and false accepted tasks vary from 10% up to 19%. False rejected tasks increase as the size of the reference window increases and vary from 4% up to 10%.

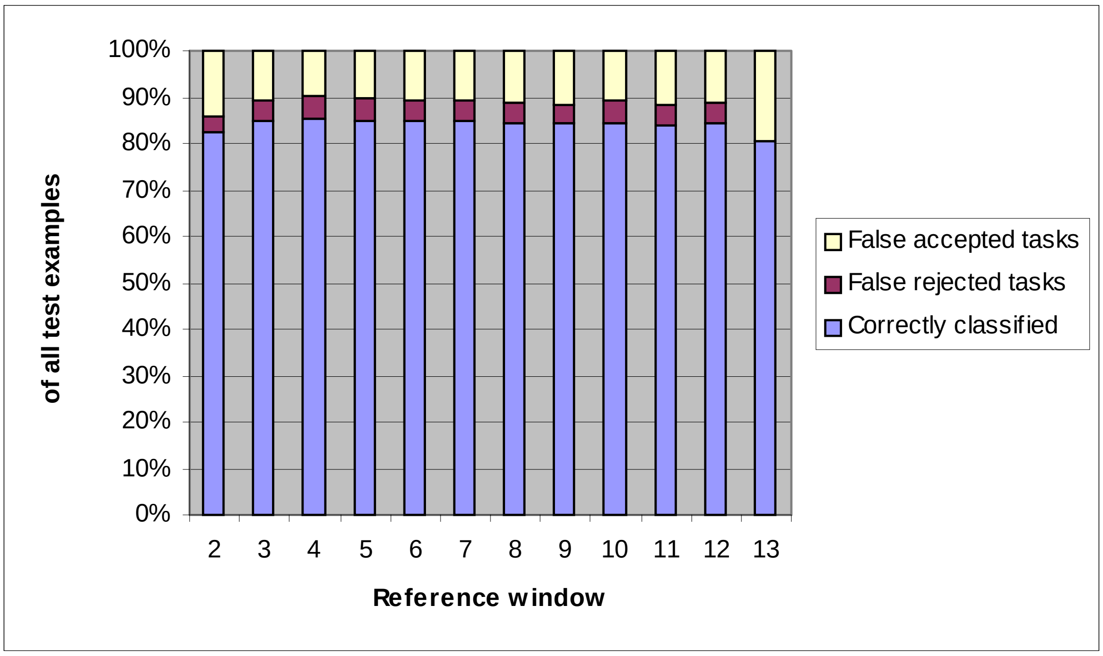

Figure 20: Classification results from training an ANN with *trWo*$_2$ and testing with *teWo*$_2$.

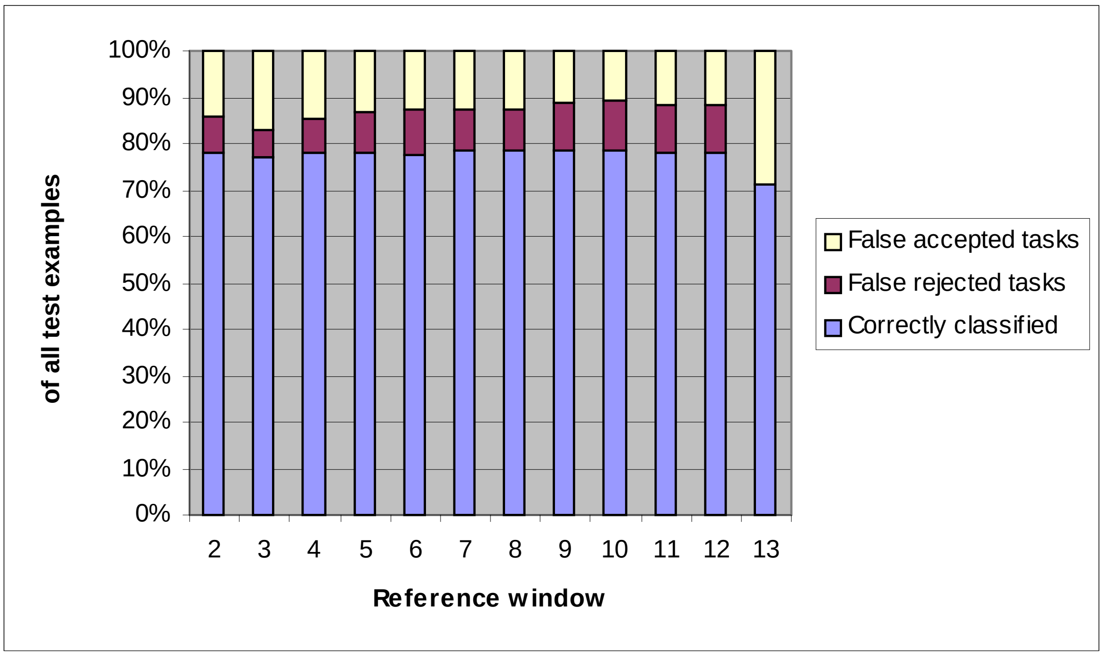

Figure 21: Classification results from training an ANN with *trWo*$_3$ and testing with *teWo*$_3$

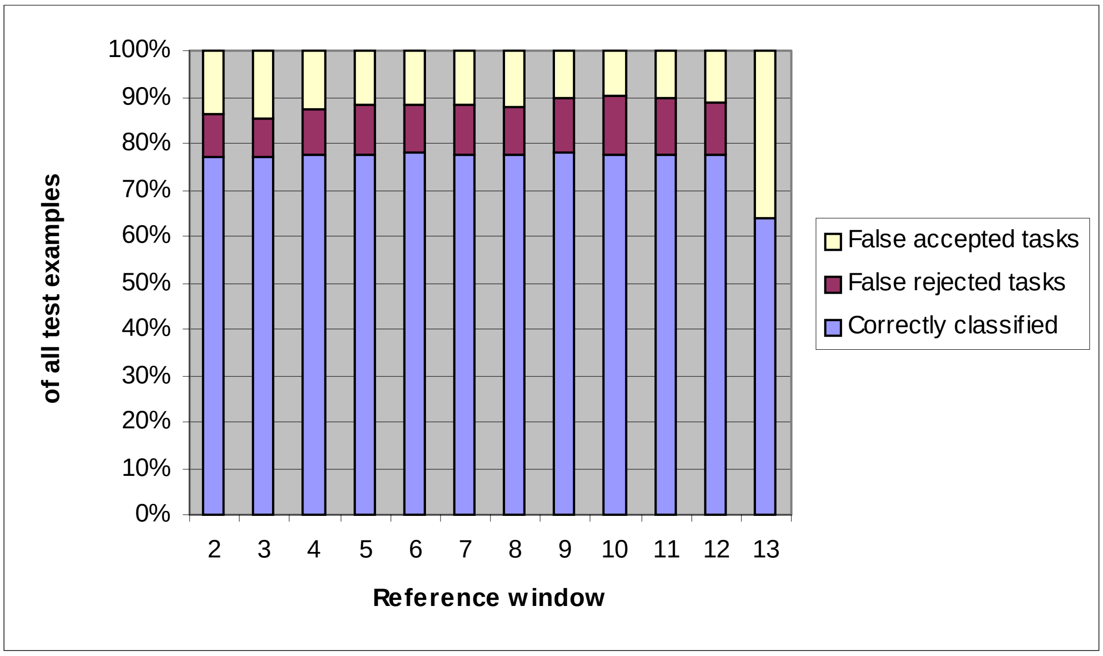

Figure 22: Classification results from training an ANN with *trWo*$_4$ and testing with *teWo*$_4$.

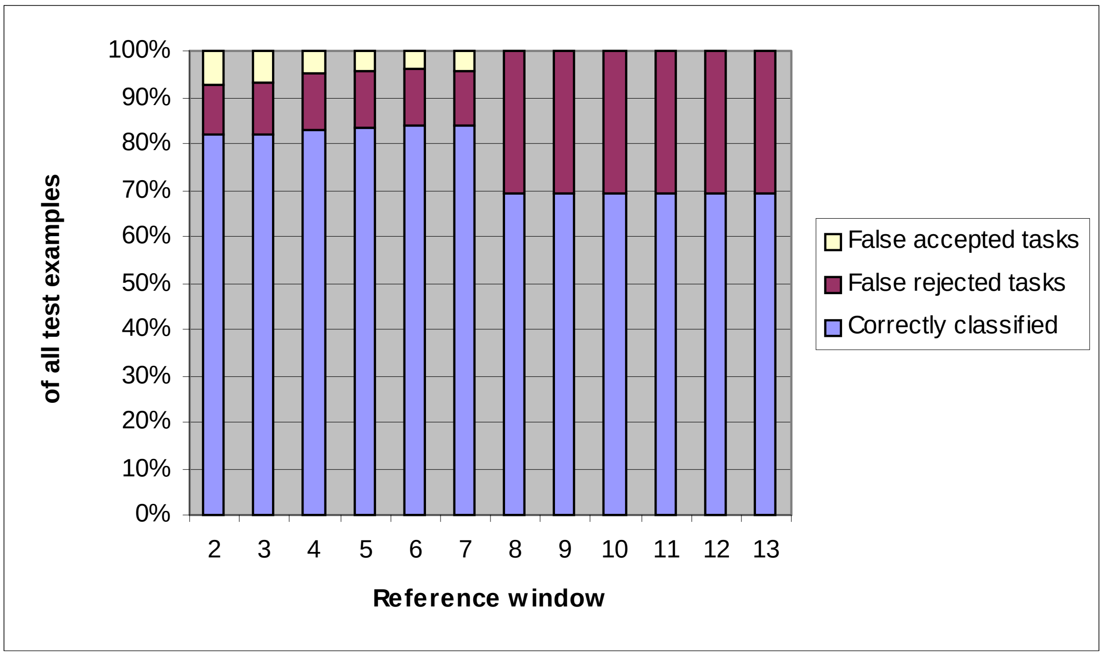

Figure 23: Classification results from training an ANN with *trWo*$_5$ and testing with *teWo*$_5$.

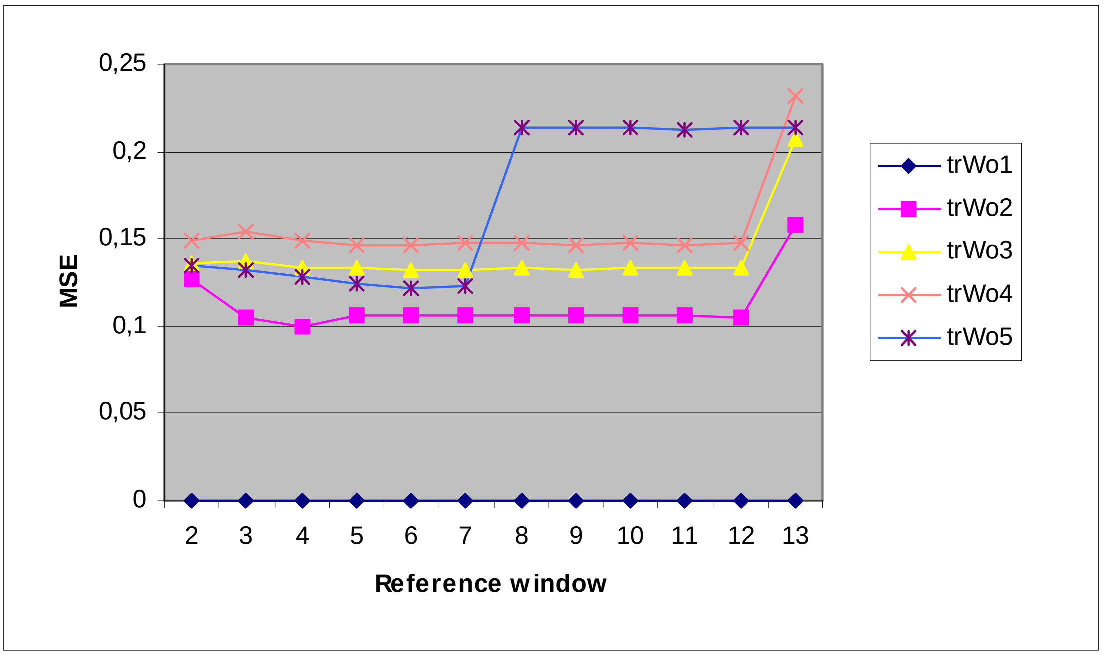

Figure 24: MSE for different training workload scenarios.

Figure 22 shows the classification results from training the ANN with *trWo*$_4$ and testing it with *teWo*$_4$. The upper bound of size of reference window is 13 where no overloads are detected. Correct results vary from 63% up to 78%. Number of false accepted tasks decrease, as the reference window becomes larger and number of false rejected tasks increases at the same time. This shows that there is a trade-off between false rejections and false accepted tasks and this was also the case where *trWo*$_3$ and *teWo*$_3$ were used (see figure 21).

The classification results from training the ANN with *trWo*$_5$ and testing with *teWo*$_5$ are illustrated in figure 23. Correct results vary from 70% up to 83%. In this case the upper bound for size of reference window is 7, where using reference window of sizes 8 to 13 gave only 70% correct results and the other 30% were false rejected tasks.

Figure 24 illustrates the MSEs in the ANNs for different types of training workload scenarios. Training the ANN with *trWo*$_1$ gave the lowest MSE. These results are not surprising since this scenario contains no overloads and makes it very simple for the ANN to learn to classify the tasks in the workload. The MSE in the ANN when using *trWo*$_2$ decreases when the size of reference window reaches 4 and then increases again. Training the ANN with *trWo*$_4$ gave the highest MSE of all the training scenarios.

6.1.1 Comparison

Using *trWo*$_2$ and *trWo*$_5$ gave considerably better results than using *trWo*$_3$ and *trWo*$_4$. The results from *trWo*$_1$ are really not particularly interesting because an admission controller is really not needed in these situations (since no overloads occur). The aim of testing *trWo*$_1$ was to show that an ANN could be trained to work as an admission controller where there is a light load on the system, where an admission controller is not really necessary. Correct results from using *trWo*$_2$ and *trWo*$_5$ go up to 86% for *trWo*$_2$ and up to 83% for *trWo*$_5$ where correct results for *trWo*$_3$ and *trWo*$_4$ go up to 79% for *trWo*$_3$ and up to 78% for *trWo*$_4$. Considering the characteristics of all the training workloads we can draw the conclusion that the number of rejected tasks in the training set has a critical affect. When the number of rejected tasks increases the number of correctly classified tasks decreases and at the upper bound for the reference window size no overloads are detected. Still, when rejected tasks are the larger part of the workload (like in *trWo*$_5$) the number of correctly classified tasks increases and at the upper bound of reference window the tasks that are incorrectly classified are all tasks that should be accepted instead of being rejected (false rejected tasks). To support this we could look at the MSE for each training workload where the error increases as the number of rejected tasks in the training workload increases. This applies to training workloads 1-4. On the other hand the MSE decreases when *trWo*$_5$ is used.

6.2 ANN Generalization Results II

In many real-time systems the arrival rate can vary at run-time. Figure 25 shows the results from training an ANN with either a constant arrival rate or an arrival rate that varies. Table 3 illustrates the outline of this experiment.

Training the ANN with a constant arrival rate and testing it with a constant arrival rate provides as high as 86% correct results (see Section 6.1). When testing this same ANN with a workload scenario where the arrival rate varies gives up to 78% correct results. This means that this particular training set is considerably more suitable for situations where the arrival rate is constant but gives feasible results where the arrival rate may vary.

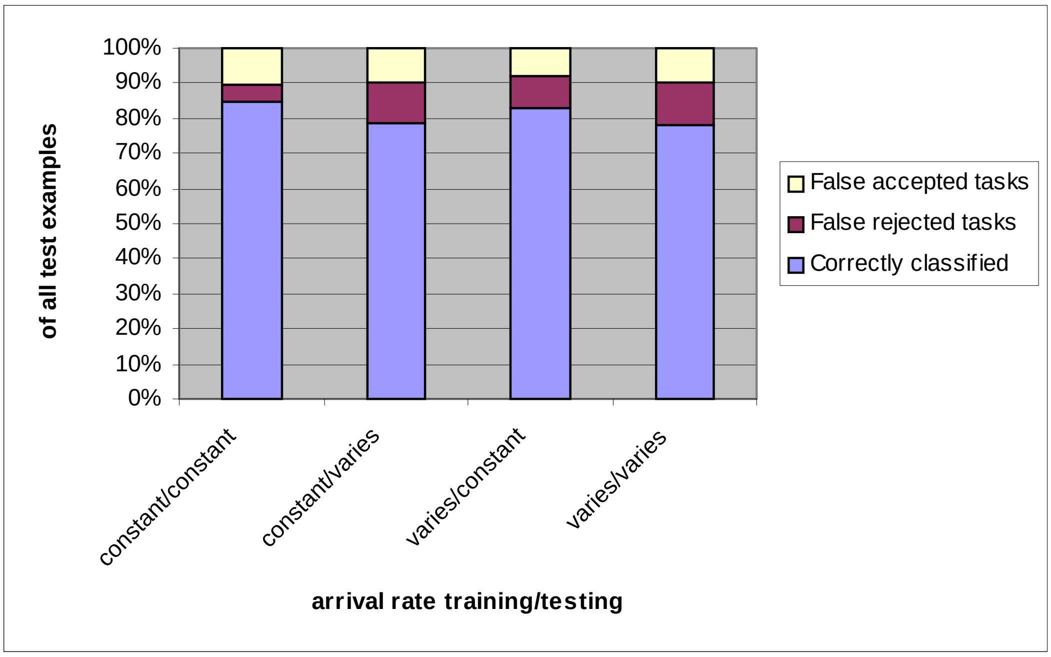

Figure 25: Classification results.

When the ANN is trained with a workload that has a varying arrival rate it gives up to 82% correct results when tested on workload scenario with a constant arrival rate. When using testing workload with a varying arrival rate gives up to 77% correct results.

6.3 Real-time Simulation Results

In these experiments we trained the ANN with *trWo*$_2$ using reference window of sizes 3, 4 and 5. The reason is that these approaches gave the best classification results and in the simulations the three ANNs are tested for different arrival rates. Table 4 shows the ANN classification results from training the ANN with *trWo*$_2$ and testing it with *teWo*$_2$.

	Correctly classified	False rejected	False accepted
Reference window of size 3	85%	4%	11%
Reference window of size 4	86%	5%	9%
Reference window of size 5	85%	5%	10%

Table 4: Classification results from using *trWo*$_2$ with reference windows of sizes 3, 4 and 5.

Figure 26 shows the number of tasks that the ANN admission controller accepts using different ANNs compared to a traditional approach. In all cases the number of accepted tasks decreases as the arrival rate increases. This is normal because as the arrival rate increases in real-time systems more overloads occur and more tasks need therefore to be rejected. When the arrival rate is 5 all three approaches accept up to 100% of all tasks in the system. When increasing the arrival rate up to 8 size of reference window seems to be critical. Having 4 tasks in reference window lets the admission controller accept considerably fewer tasks than using reference window of size 3 and 5 and the number of tasks that are accepted using 4 tasks in the reference window is closest to the number of tasks that a traditional approach accepts. This is also the case when the arrival rate is 9 or 10. When the arrival rate is 20, ANN with reference window of size 4 accepts much fewer tasks than a traditional approach.

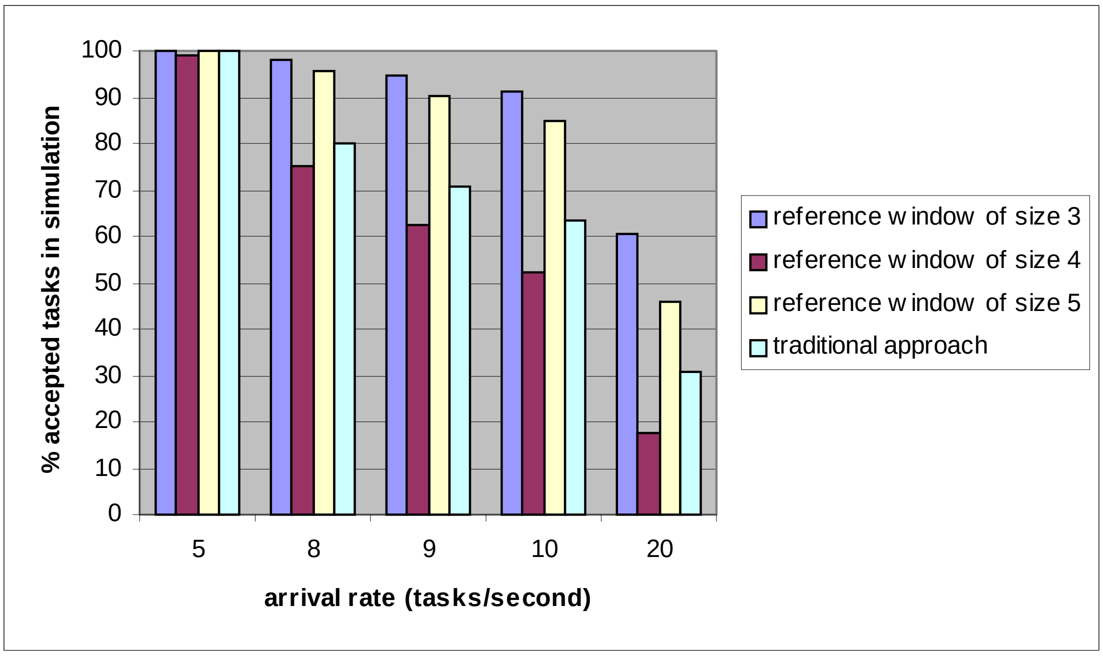

Figure 26: Accepted tasks in simulations.

What is interesting is that even though the admission controller using a reference window of size 4 accepts considerably less tasks than when sizes 3 and 5 are used, it does not show as obvious difference when considering number of completed tasks in the system (see figure 27). When the arrival rate is 5 the size of reference window does not seem to be so critical. As the arrival rate increases the number of completed tasks decreases and this is normal because the number of overloads in the system increases as the arrival rate increases. Using reference window of size 4 has the highest number of completed tasks of all the windows when the arrival rate is 7, 8 and 9. When the arrival rate is 5, keeping 3 tasks in reference window provides the highest number of completed tasks. Using reference window of size 5 provides the highest number of completed tasks when the arrival rate is 20. The number of completed tasks is closest to a traditional approach when using reference window of size 4 considering all arrival rate except for 20 where reference window of size 5 get closest to the traditional approach.

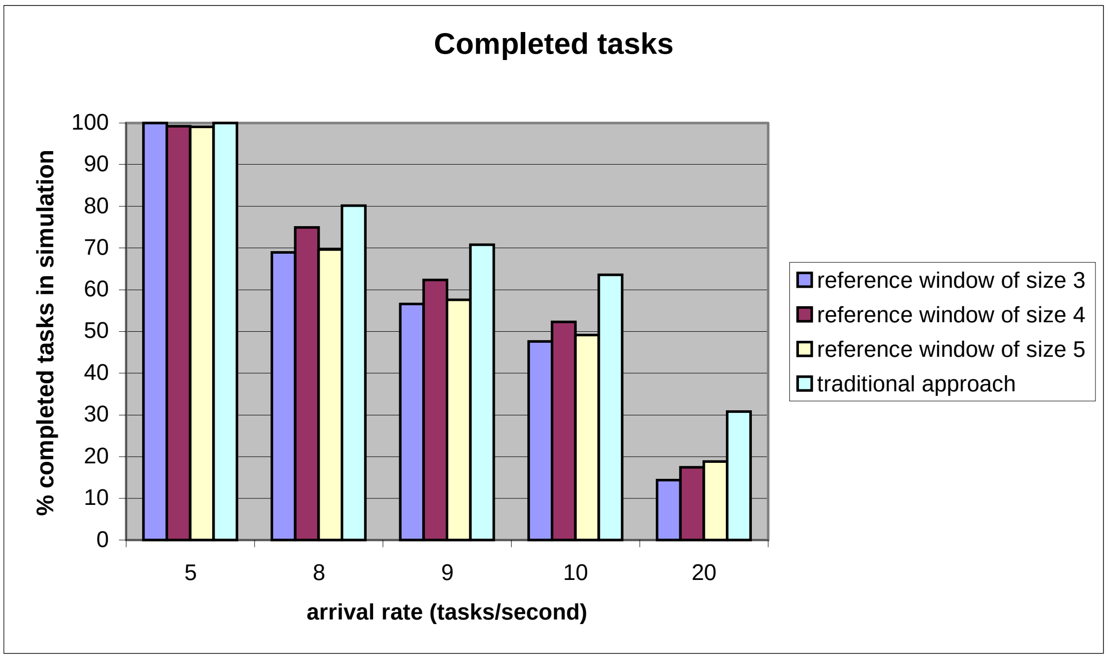

Figure 27: Completed tasks in simulations.

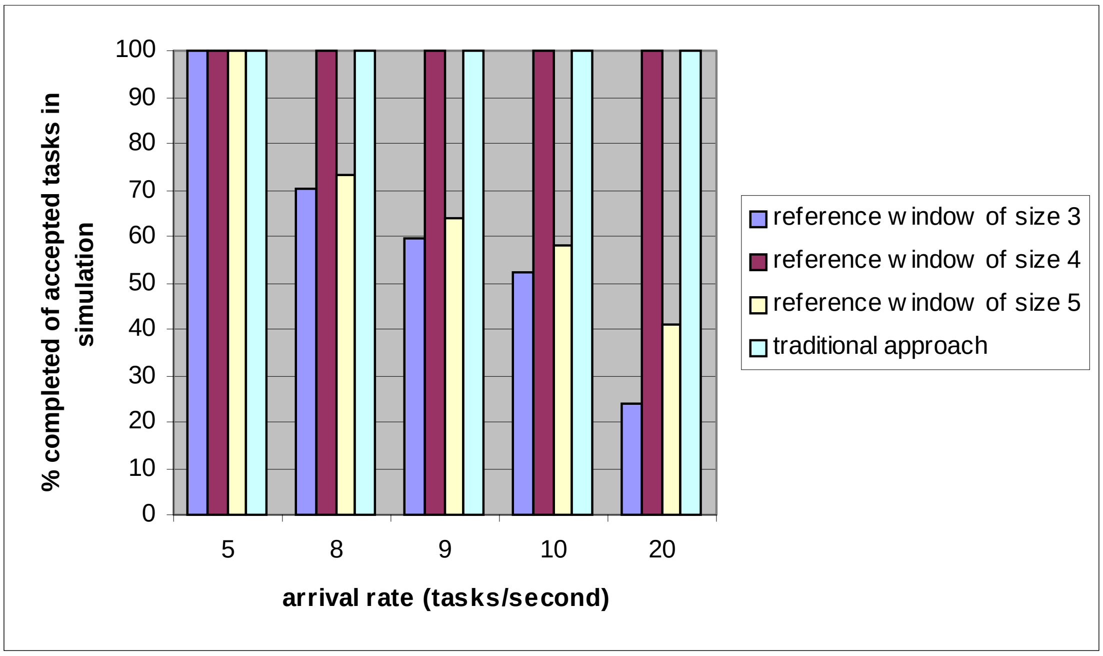

Figure 28: Completed tasks that are accepted in simulations.

Figure 28 shows the value of *c* that was defined in Section 5.1. When using reference window of size 4, all tasks that are accepted finish their execution just like when using a traditional approach. This applies for all arrival rates. The other reference windows (with 3 and 5 task in window) give considerable different results but when arrival rate is 8, 9, 10 or 20 then it is better to keep 5 tasks in reference window than 3. The reader is referred to table 5.1 and 5.2 in order to get an idea how the arrival rate affects the number of accepted tasks in the system.

6.4 Timing Results

Figure 29 shows our results from measuring the time it takes to perform a RED schedulability test and ANN schedulability test. The ANN approach takes a bit longer time to execute than RED when there are 1-5 tasks in the system but as the number of tasks increases the time it takes to perform the RED schedulability test increases considerably. On the other hand, the time it takes to perform the test using an ANN has a constant value independent of the load of the system because we always consider the same number of tasks.

There are 70 tasks on average in the system when an overload is detected. When the number of tasks is 70 the time it takes to perform the RED schedulability test is 37 times more than the time it takes to perform the test using an ANN. This demonstrates that the computational cost of using ANNs for performing schedulability tests is much lower than using a traditional approach when there is a high load on the system.

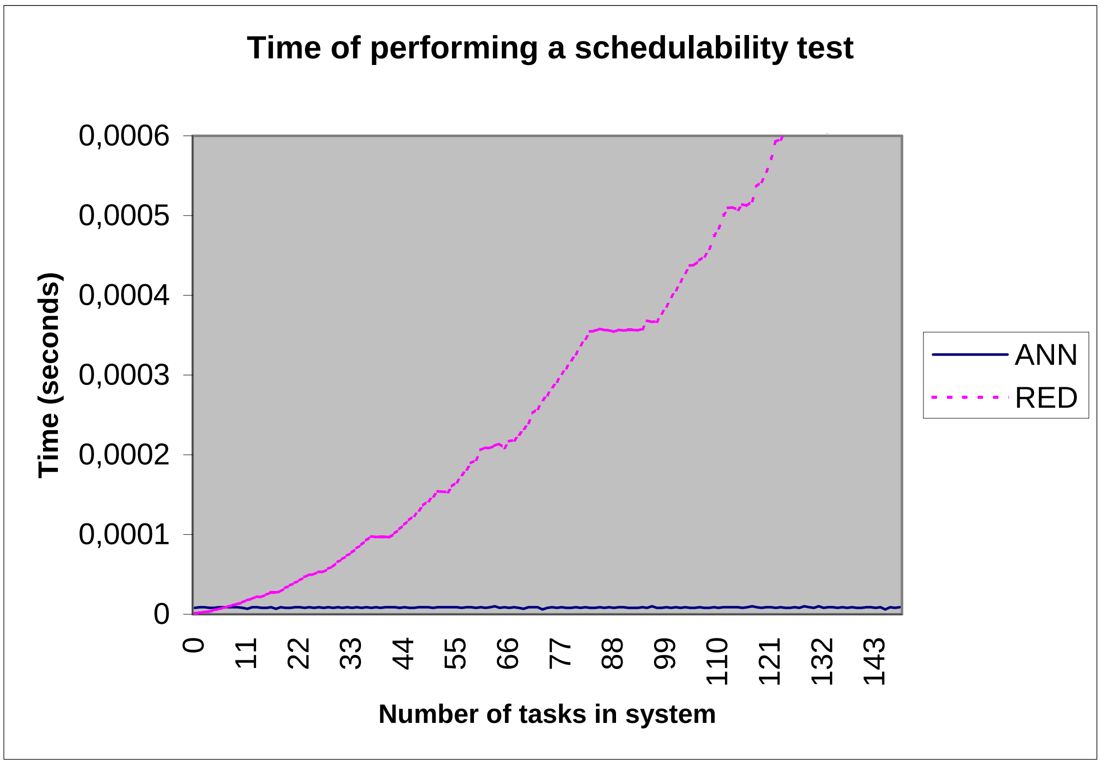

Figure 29: The time it takes to perform a schedulability test using a traditional approach compared to using an ANN.

6.5 Summary of Results

6.5.1 ANN Generalization

In this section we will sum up the results from training the ANN. This involved training the ANN using one type of workload scenario and then testing it with the same type of scenario. This also involved using training and testing scenarios with varying arrival rate (see Section 5.2.2).

In order to sum up our ANN generalization results, we get the following:

- Training and testing an ANN with a workload with the arrival rate 8 gave the highest number of correctly classified tasks (up to 86%).
- When using training and testing workloads of a high arrival rate (20 transactions/second) the number of false accepted tasks is much lower than false rejected tasks. On the other hand when the arrival rate is lower the number of false accepted tasks has tends to be higher than number of false rejected tasks.
- There is an upper bound for the size of reference window. When the arrival rate is 8-10 the upper bound is 13 and when the arrival rate is 20 the upper bound is lower, or 8. On the other hand when situations where no overloads occur (low arrival rate) there is no upper bound for the size of reference window.
- Using a reference window of size 4 and training and testing the ANN with the arrival rate 8 gave the least MSE.

6.5.2 Real-time Simulations

In this section we will sum up our results from running real-time simulations using an ANN admission controller. In these experiments we used reference windows of size 3, 4 and 5. The ANN was trained using *trWo2* (see Section 5.2).

The following sums up our results:

- The ANN admission controller decreased the number of tasks that were accepted as the arrival rate increased. Using reference window of size 4 decreased the number of tasks that were accepted considerably more than using reference window of size 3 and 5. Using reference window of size 5 decreased the number tasks that were accepted more than using reference window of size 3.
- Using reference window of size 4 provided higher number of completed tasks than the other windows for all arrival rates that were tested and size 5 gave more number of completed tasks than size 3.
- All tasks that were accepted completed their execution only if a reference window of size 4 was used. Size 5 gave slightly better results than size 3. In cases where size is 3 or 5 the number of tasks that were accepted and completed their execution decreased as the arrival rate increased.

6.5.3 Time

In this section we will sum up the results from the time measurements. In our experiments we compared the time it takes to perform a schedulability test using a traditional approach to an approach where an ANN is used.

- When there are 1-5 tasks currently in the system the time it takes to perform the schedulability test using an ANN is a bit more than the time it takes to perform the test using RED.
- As the number of tasks currently in the system increases the time it takes to perform the schedulability test using RED increases like the function $n * \log n$ where n is the number of tasks in the system.
- The time it takes to perform a schedulability test using an ANN has a constant value.

- There are 70 tasks in the system on average when an overload is detected. When the number of current tasks in the system is 70 the RED schedulability test takes 37 times more time to perform than then ANN schedulability test.

7 Conclusions

In this chapter the conclusions of our work is given. There are many factors that have to be considered when looking at the results in our project and this section covers a discussion about our results.

7.1 Summary

In dynamic real-time systems typically an admission controller is used to handle transient overloads. Task arrives to the system and the admission controller runs a schedulability test in order to see whether the task will meet its deadline if it is accepted. If a task passes the schedulability test it is accepted, otherwise rejected and we say that an overload has occurred. There are two major disadvantages that refer to these systems. Firstly, in order to provide graceful degradation the overload must be detected as soon as possible in order to prevent the consequences. Secondly, the traditional schedulability test is computationally expensive and normally runs in $n * \log n$ time where n is the number of tasks in the system.

In our project we have focused on overcoming the second problem, and therefore reduce the consequences of a transient overload. We have introduced a new admission controller that uses ANNs to perform the schedulability test. ANNs have the advantage of being computationally cheap in use. What we have done in this project is that we have (i) trained an ANN in order to perform a schedulability test. Then we (ii) evaluated the generalization of the ANN and (iii) evaluated the performance of the new admission controller using a real-time simulator. Finally, we (iv) compared the time it takes to perform a schedulability test using an ANN to a traditional schedulability test.

7.2 ANN Admission Controller

Our new admission controller uses an ANN to perform a schedulability test. When task arrives to the system the task information is pre-processed and put in an array containing information about recent tasks in the system. This array is used as an input to the ANN which provides admission decision after feed forwarding the array through the ANN. The ANN is trained off-line using data generated by a real-time simulator.

7.3 Evaluation of using an ANN for Admission Control

When evaluating the training of the ANN using test examples we were able to get up to 86% correctly classified tasks with 4% false rejected tasks and 10% false accepted tasks. We have also showed that ANNs can be trained to provide up to 78% correctly classified tasks where the arrival rate varies at run-time with 13% false rejected tasks and 9% false accepted tasks. These results show that ANNs can be trained to work as an admission controller in dynamic real-time systems where firm deadlines are considered. If a task with a firm deadline misses its deadline, it is simply cancelled.

When evaluating the performance of using an ANN admission controller in real-time simulations we showed that our admission controller is able to provide results that do not differ that much from a traditional approach. Our approach accepts slightly fewer tasks and slightly fewer tasks finish their execution than when a traditional approach is used. Still, all tasks that our ANN admission controller accepts finish their execution, which is also the case when a traditional admission controller is used.

We have shown that the time it takes to perform the ANN schedulability test is much less than using a traditional approach when there is a heavy load on the system. There are 70 tasks in average in the system when an overload is detected. The time it takes to perform the test takes 37 times more time using a traditional approach than using an ANN. The computational cost of a traditional approach proved to be $n * \log n$ where the computational cost of using the ANN proved to be a constant c.

7.4 Discussion

In this section we discuss the results and draw some conclusions regarding the results. We will begin with discussing the training of the ANNs, then move on to the simulation results using an ANN admission controller and finally we will discuss the time measurements made in this project.

7.4.1 Training the ANN

When training the ANN the arrival rate seems to be crucial for the training because a higher arrival rate provides higher number of rejected tasks in the training set. What is interesting is that even though rejected tasks are increased in the set the number of false rejected tasks increases considerably more than number of false rejected tasks. One would think that this should be vice verse. This indicates that there are a certain number of tasks that present overloads which are difficult for the ANN to detect. We were not able to identify these tasks in details but in many cases the false output value is very close to the threshold. E.g. in many cases if the target value was 1 the ANN provided a value just below the threshold. This also applies to when the target was 0 and the ANN provided values just above the threshold. Changing the threshold value did not increase the number of correctly classified tasks. Instead, a trade-off occurred between false rejected tasks and false accepted tasks which is not surprising. We can therefore draw the conclusion that by changing the threshold value it is possible to control whether to keep few false rejected tasks or false accepted tasks. In some firm real-time system it could e.g. be desired to keep the number of false rejected tasks as low as possible in order to keep as much utilization in the system as possible.

When the majority of tasks in the training set are rejected the ANN provides a bit different results than discussed earlier. The number of false rejected tasks is considerably higher than the number of false accepted tasks. This could be because more effort is put on teaching the ANN how to classify overloads than non-overloads due to the higher number of examples in the training set that present overloads.

In our experiments where the ANN was trained and tested with workload scenarios that either had a constant arrival rate or an arrival rate that varies (see Section 6.2 for details) the results showed that the number of false accepted tasks did not change at all between experiments. On the other hand the number of false rejected tasks increased in some cases. This is probably due to what we discussed above, i.e. there seems to be a certain number of tasks that present overloads that the ANN simply is not able to detect. As mentioned before these tasks were not identified.

Training an ANN with a constant arrival rate or with an arrival rate that varies gives almost exactly the same results when tested on a scenario that contains a varying arrival rate. The same can be concluded when training the ANN with either a constant arrival rate or with a varying one. From this we draw the conclusion that ANN can be trained to provide feasible results in situations where the arrival rate might be varying at run-time.

In our project we have not considered different number of hidden nodes in the hidden layer of the ANN. The reason why not any time was spent on this is that according to Steinsen (1999), increasing the number of hidden nodes did only slow the learning process and did not provide any different results. For that reason, we did not investigate the number of hidden nodes in this project.

7.4.2 Real-time Simulations

When using ANNs for admission controller it is important that the ANN provides as good generalization as possible in order to accept/reject tasks in a correct manner. By comparing the results to a traditional one we get an idea of how well the ANN admission controller works. It would be optimal if the ANN admission controller would provide the exact same results as a traditional one.

Our approach accepts slightly fewer tasks than a traditional approach. This could be because the ANN does not provide 100% correct classification and some tasks are obviously rejected when they should be accepted. This could be a disadvantage when the aim is to keep as high utilization in the system as possible. If the ANN would accept more tasks than the traditional approach, then the ANN would obviously be accepting tasks that are not able to finish their execution. This is obviously not desirable because then we could just as well skip the admission controller since the real-time system has its resource limitations and would be overloaded all the time anyway.

All tasks that are accepted finish their execution using our approach which is also the case when using a traditional approach. Still, this criterion must not be taken to seriously because if e.g. 1 task of 1000 would be accepted then all accepted tasks would finish their execution in time. The other criteria like number of accepted tasks and number of completed tasks are important as well.

7.4.3 Time Constraints

In our project we have investigated the schedulability test carried out by the admission controller in dynamic real-time systems. The general aim, following Ramamritham et al. (1999), has been to reduce the time spent on scheduling and thus increase the time left to execute tasks. In order to understand the relative cost of performing a schedulability test using an ANN compared to traditional approach, we have implemented a traditional schedulability test (Robust Earliest Deadline) and then measured the time it takes to run the test. Both approaches have been tested using different simulated workloads with varying numbers of tasks in the system. Our results confirm that the time it takes to perform a schedulability test, using a traditional algorithmic approach, increases significantly as the number of tasks increases in the system. Normally the cost increases as function of $n * \log n$, where n is the number of current tasks in the system. Further we have shown that the time it takes an ANN to perform the schedulability test is (almost) constant, independent of the load of the system. Our measurements illustrate the *relative* differences between the two approaches and their respective complexity. The *absolute* differences measured here, however, are less significant. This is due to the fact that the SUN-platform, on which these measurements were taken, performs some additional operations which might affect the results because this platform is normally not used for real-time systems. Our results show that execution time is gained by using an ANN admission controller but at the same time we have to pay for this with a certain number of incorrect admission decisions in the system. However, further research will be required in order to better understand the tradeoff between the reduced run-time costs of an ANN admission controller and the possible consequences of incorrect admission decisions, using hardware that is normally used for real-time systems.

7.5 Future Work

It would be interesting to see which attributes are most critical for the ANN. Perhaps some of the attributes could be reduced and that would provide even lower computational cost. One way to do this is to use ANN inversion. Some previous work has been done on ANN inversion and the reader is referred to e.g. Jacobsson (1998).

In our project the ANN has been trained off-line, i.e., when the system is not operational. If the ANN would be trained on-line, i.e. when the system is operational, the ANN would learn how to classify new workloads as tasks arrive dynamically to the system. When there is a heavy load on the system, the information about the workload scenarios could be saved so when the load on the system decreases, time could be spent on training the ANN using the saved information.

We focused on training the ANN to make scheduling decisions. Steinsen (1999) introduced a method to predict future overloads. It could be interesting to combine these approaches into one. ANNs with two outputs could be used where e.g. 0 0 would mean accept task and no overload is about to occur, 0 1 would mean accept task and overload is about to occur, 1 0 would mean reject task and no overload is about to occur and finally, 1 1 would mean reject task and overload is about to occur.

In order to get an idea about the *relative* difference between using ANNs and traditional approaches for admission control some further investigations are necessary. The aim of these investigations would be to evaluate the tradeoff between gained execution time and incorrect admission decisions. In order to get an idea of the *relative* difference it would be optimal to do these experiments on platforms running real-time operating systems, e.g. OSE Delta.

References

Burns, A. and Wellings, A. 1997. *Real-Time Systems and Programming Languages,* second edition. Addison-Wesley. England.

Hansson, J. 1998. *RADEx++*. Technical report. The Department of Computer Science. University of Skövde. Sweden.

Hansson, J. 1999. *Value-Driven Multi-Class Overload Management in Real-Time Database Systems,* Phd thesis. Institute of Technology, Linköpings Universitet. Sweden.

Jacobsson H. 1998. *Inversion of an Artificial Neural Network Mapping by Evolutionary Algorithms with Sharing.* Final year dissertation. HIS-IDA-EA-98-113. University of Skövde. Sweden.

Ramamritham K., Atif Y., Hamidzadeh B. 1999. To Schedule or Execute: Decision support and Performance Implications. Real-Time Systems. Volume 16. Issue 2. pp. 281-313.

Russell, S. and Norwig, P. 1995. *Artificial Intelligence, a Modern Approach*. Prentice-Hall. USA.

Sivasankaran R. M., Purimetla B., Stankovic J. A., Ramamritham K. and Towsley D. 1995. *Design of RADEx – real-time active database experimental system. Technical report*. University of Massachusetts. Amherst.

Spuri, M. and Buttazzo, G. 1996. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*. Volume 10. Issue 2. pp. 179-210.

Stankovic J. A., Spuri M., Ramamritham K. and Buttazzo C. C. 1998. *Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms*. Kluwer. USA.

Steinsen R. M. 1999. *Predicting Transient Overloads in Real-Time Systems Using Artificial Neural Networks.* M.Sc. dissertation. HS-IDA-MSD-99-006. University of Skövde. Sweden.

Appendix A

Pseudo code for counting empty time-steps (Steinsen, 1999).

```
//clock 'tick'
t := 0;
//go through the whole workload file
while not workloadFile.eof do
begin
    task := readOneTask(workloadFile);
    //Check to see if the arrival time of the task is less then the time
    //granularity multiplied with number of 'tics'
    if ((t*TIMESTEP)<=task.arrivalt) then
    begin
        //Check if the arrival time is less than the next time step
        if ((t*TIMESTEP+TIMESTEP)>task.arrivalt) then
        begin
            //no empty steps – do nothing
        end
        else
        begin
            task.emptyStep++;
        end
        t := t+1;
    end
    else
    begin
        //If this happens there are two tasks in one time-step which is
        //not allowed – ERROR!!
    end
end
```