Algorithms for Managing QoS for Real-Time Data Services Using Imprecise Computation

Thesis Report LiTH-IDA-Ex-02/90

Mehdi Amirijoo
University of Linköping
meham@ida.liu.se

Abstract

Lately the demand for real-time data services has increased. Applications used in manufacturing, web-servers, e-commerce etc. are becoming increasingly sophisticated in their data needs. In these applications it is desirable to process user requests within their deadlines using fresh data. The real-time data services are usually provided by a real-time database. Here, the workload of the databases cannot be precisely predicted and, hence, the databases can become overloaded. As a result, many deadline misses and freshness violations may occur. To address this problem we propose a QoS-sensitive approach to guarantee a set of requirements on the behavior of real-time databases, even in the presence of unpredictable workloads. Our approach is based on imprecise computation, where it is possible to trade off resource needs for quality of requested service. Imprecise computation is applied on both data and transactions. A QoS specification is given in terms of data and transactions quality. We propose two dynamic balancing algorithms to balance the workload and the quality of the data and transactions, so that a given QoS specification can be satisfied. Further, we apply feedback control scheduling to provide robustness against unpredictable workload variations. We have carried out a set of experiments to evaluate the performance of our algorithms. In our simulation studies we have applied a wide range of workload and run-time estimates to model potential unpredictabilities and the performance analysis show that the proposed algorithms give a robust and controlled behavior of real-time databases, in terms of transaction and data quality, even during transient overloads and when we have inaccurate run-time estimates of the transactions.

Keywords: Quality of Service, Real-time Databases, Imprecise Computation, Feedback Control Scheduling

Acknowledgments

I would like to thank my supervisors Dr. Jörgen Hansson and Dr. Sang H. Son for their guidance and help throughout my work. I am indebted to Kyoung-Don Kang for providing and helping me with the simulator used to perform the experiments. Many thanks to the members of RTSLAB, as it has been a pleasure working with them.

I would like to dedicate my thesis to my family, especially my parents, for their support and help during my years at the University of Linköping. Special thanks to Claudia for her wonderful company, which has always been welcomed after a long day of study.

Mehdi Amirijoo
October 2002

Contents

1 Introduction **1**

1.1 Report Outline . . . 2

2 Background **3**

2.1 Real-time Databases . . . 3

2.1.1 Data in RTDBs . . . 4

2.1.2 Transactions in RTDBs . . . 5

2.1.3 Trading off Quality for Timeliness . . . 6

2.2 Feedback Control Scheduling . . . 7

2.2.1 Specification . . . 8

2.2.2 Modeling . . . 10

2.2.3 Controller Design . . . 10

2.2.4 Stability and Steady-State Error . . . 11

2.3 Imprecise Computing Techniques . . . 12

3 Problem Formulation **15**

3.1 Background . . . 15

3.2 Problem Analysis . . . 16

3.3 Objective . . . 17

4 Approach **19**

4.1 Data and Transaction Model . . . 19

4.1.1 Database Model . . . 19

4.1.2 Transaction Model . . . 19

4.1.3 Data Model and Data Management . . . 21

4.1.4 Admission Control, Scheduling and Concurrency Control 22

4.2 Performance Metrics and QoS specification . . . 23

4.3 Feedback Control Scheduling Architecture . . . 24

4.3.1 Streams and Sources . . . 25

4.3.2 Transaction Handler 26
4.3.3 Feedback Control 26
4.3.4 Admission Control 27
4.3.5 Precision Control 27
4.4 System Modeling 27
4.5 Feedback Controller Design and Tuning 30
4.5.1 Miss Percentage Controller 32
4.5.2 Utilization Controller 32
4.5.3 Derivation of Single Control Signal 33
4.5.4 Integrator Antiwindup 34
4.5.5 Controller Implementation 35
4.5.6 Controller Tuning and Performance 35
4.6 Algorithm Specification 38
4.6.1 FCS-IC-1 39
4.6.2 FCS-IC-2 41
4.7 QoD Management 41
4.7.1 Effects of Varying MDE 41
4.7.2 Modeling and Online Tuning 44
4.7.3 Derivation of f 46

5 Performance Evaluation 47
5.1 Experimental Goals 47
5.2 Baselines 48
5.2.1 Baseline-1 48
5.2.2 Baseline-2 48
5.3 Simulation Setup 49
5.3.1 Data and Update Transactions 49
5.3.2 User Transactions 49
5.3.3 QoS Specifications 50
5.4 Experiments and Results 51
5.4.1 Varying Load 51
5.4.2 Varying EstErr 54
5.4.3 Transient Performance 56
5.4.4 Varying QoS specification 58
5.4.5 Summary of Results and Discussions 62

6 Related Work 65
6.1 Feedback Control Scheduling 65
6.2 QoS Management in RTDBs 66
6.3 Imprecise Computation 67

7 Summary **69**

7.1 Conclusion 69

7.2 Future Work 70

7.2.1 System Modeling and Feedback Control Scheduling . . 70

7.2.2 Management of Derived Data 70

7.2.3 Service Differentiation 71

7.2.4 QoS Specification 72

7.2.5 Managing Data Impreciseness 72

Table of Abbreviations **79**

Table of Variables **81**

List of Figures

2.1 Feedback control scheduling architecture 8
2.2 Definition of settling time (T_s) and overshoot (M_p) 9

3.1 A set of arriving updates. Updates one and four are discarded since their value does not deviate much from the currently stored value. 16

4.1 Feedback control scheduling architecture 25
4.2 Model of the controlled system 28
4.3 Miss percentage / utilization controller architecture 31
4.4 System profiling and model tuning 36
4.5 Profiling gained utilization with respect to different MDE . . 45

5.1 Average performance for *Load* = 50, 75, 100, 150 and 200%, *EstErr* = 0, QoSSpec1 52
5.2 Average performance for *EstErr* = 0, 0.25, 0.50, 0.75 and 1.0, *Load* = 200%, QoSSpec1 55
5.3 Transient performance for FCS-IC-1. *EstErr* = 1.0, *Load* = 200%, QoSSpec1 57
5.4 Transient performance for FCS-IC-2. *EstErr* = 1.0, *Load* = 200%, QoSSpec1 59
5.5 Transient performance for FCS-IC-2. *EstErr* = 0.0, *Load* = 200%, QoSSpec1 60
5.6 Average performance for *Load* = 50, 75, 100, 150 and 200%, *EstErr* = 0, QoSSpec2 61

Chapter 1

Introduction

Lately the demand for real-time data services has increased. Applications used in manufacturing, web-servers, e-commerce etc. are becoming very sophisticated in their data needs. In these applications it is desirable to process user requests within their deadlines using fresh data. Since the external environment is constantly changing, it is imperative to maintain consistency between the environment and the database. In dynamic systems, such as web servers and sensor networks with non-uniform access patterns, the workload of the databases cannot be precisely predicted and, hence, the databases can become overloaded. As a result, many deadline misses and freshness violations may occur. To address this problem we propose a quality of service (QoS) sensitive approach, to guarantee a set of requirements on the behavior of the database, even in the presence of unpredictable workloads. Our scheme is important to applications where timely execution of transactions is emphasized, but where it is not possible to have exact analysis of the worst case execution times.

Our approach is based on imprecise computation [18], where it is possible to trade off resource needs for quality of requested service. Imprecise computation techniques have been successfully applied to applications where timeliness is important. In our work, imprecise computation is employed on both data and transactions. A QoS specification can be given in terms of data and transaction quality. A transaction is divided into one mandatory and one or several optional subtransactions. The mandatory subtransaction is necessary for an acceptable result and must be computed to completion before the transaction deadline. The optional subtransactions are executed when requested utilization is available. By executing more optional subtransactions, the overall quality of the result produced by the transaction is enhanced. During transient overload the miss percentage of optional subtransactions may increase, degrading the quality of transactions. Here, a higher transaction quality can be achieved by allocating more resources to transactions. This is achieved by lowering the workload of the data updates, resulting in a decrease in data quality. For managing the workload allocation between user and update transactions, we propose two dynamic balancing algorithms, FCS-IC-1 and FCS-IC-2, to balance the workload and hence the quality of the data and transactions. Main challenges include unpredictability of workload and effective workload balancing between user transaction and data quality.

We apply feedback control scheduling policy [20] to provide robustness against unpredictable workload variations. In a feedback control system, reference (target) performance can be achieved by dynamically adjusting the system behavior based on the difference of the current performance and the reference. Feedback control is very effective to support the specified performance when the dynamics of the controlled system includes uncertainties. By adapting the robustness of feedback control, we can provide the guaranteed real-time data services in terms of transactions and data quality.

The suggested algorithms, FCS-IC-1 and FCS-IC-2, are designed such that the behavior of a real-time database (RTDB) can be controlled, even in the presence of load variation and inaccurate run-time estimated. We have carried out a set of experiments to evaluate the performance of our algorithms. In our simulation studies we have applied a wide range of workload and run-time estimates to model potential unpredictabilities. FCS-IC-1 and FCS-IC-2 give robust and controlled behavior of transaction and data quality, even during transient overloads and when we have inaccurate run-time estimates of the transactions. This has been shown by comparing the performance against selected baseline algorithms.

1.1 Report Outline

The rest of this report is organized as follows. In chapter 2, preliminaries and terminology needed for the rest of this report is given. The background and analysis of the problem to be solved is discussed in chapter 3. The approach and suggested algorithms are defined in chapter 4. In chapter 5, the result of the simulation studies are presented. This report ends with a section on related work, given in chapter 6, and a summary in chapter 7 where conclusions and future work are discussed.

Chapter 2

Background

In this chapter we give the theory of the techniques used throughout this report. First, a short description of real-time databases is given, followed by a section on feedback control scheduling. Finally, an overview of the topic of imprecise computing is given.

2.1 Real-time Databases

RTDBs differ from traditional databases in several ways. Their differences and common properties are discussed below with regards to several aspects, such as data and transaction semantics and timing considerations. Also techniques for trading off quality for timeliness are discussed. First, a general comparison between traditional databases and real-time databases is given.

RTDBs facilitate the same feature as traditional databases, below referred to as simply databases. RTDBs must be able to handle mechanisms for describing data, maintaining the correctness and integrity of data, efficient access to data and the correct execution of queries and transactions despite concurrency and failures. Although the similarities, they differ in two major aspects; data and transactions semantics.

Considering data semantics, databases deal with persistent data, i.e. data that does not age. RTDBs on the other hand, for the most part, deal with *temporal* data, i.e. data that may become outdated. For example, consider the case of e-commerce applications. They span the spectrum from low-level data, such as stock prices to high-level aggregated data, e.g. recommended selling/buying point. It is well known that stock prices change constantly, leading to the requirement that the database keeps the information fresh. Hence, it is imperative that the state of the environment, as given by the

RTDB is consistent with the actual state of the environment. In order to maintain high freshness, timely monitoring of the environment as well as timely processing of the sensed information is necessary. The sensed data in turn is used to derive other data.

In addition to the time constraints of data, timing requirements also arise because of the need to process and make data available for requesting applications (e.g. a transaction should return an answer within a certain point in time). This leads to the notion of predictability. As predictability can be expressed in functional and temporal requirements, we only discuss the latter in the following sections. Below a more detailed description of the differences with regards to data and transactions semantics is given.

2.1.1 Data in RTDBs

The need to maintain consistency between the state of the environment as given by a RTDB and the actual environment leads to the notion of temporal consistency, which is defined as [23]:

- *Absolute consistency*, which refers to consistency between the environment and the state of the RTDB, and
- *Relative consistency*, which refer to consistency between a set of data objects that are used to derive a new data object.

Before giving further explanations of the above consider the following. A data object d_i is defined by the following: (CV_i, TS_i, AVI_i), where CV_i is the *current value* of d_i, TS_i the *timestamp* giving the time when the observation of CV_i was done, and AVI_i the *absolute validity interval* of d_i. AVI_i refers to the length of time followed by TS_i, during which d_i is considered to have absolute validity, i.e. d_i is considered fresh. Temporal data objects can be further classified into *base data* and *derived data*. Base data holds the view of the outside environment, while derived data is derived from possibly multiple base or derived data. A set of data objects used to derive a new data object form a *relative consistency set R*. Each such set R is associated with a *relative validity interval* R_{rvi}. Assume that $d_i \in R$. d_i has a correct state iff:

- d_i is logically consistent, i.e. satisfy database consistency constraints, and
- d_i is temporally consistent, i.e.
  - $CurrentTime \leq TS_i + AVI_i$

– $\forall d_j \in R, |TS_j - TS_i| \leq R_{rvi}$

Considering the second temporal consistency requirement, this means that for a derived data object, the data objects in the relative consistency set, $d_i \in R$, should not be observed far away from each other. This, of course, is to maintain a certain degree of freshness among members of R.

Further, temporal data can be updated periodically or aperiodically. A periodic update occurs at fixed intervals, while an aperiodic update is not predictable and occurs due to an event, such as when the data value has changed.

2.1.2 Transactions in RTDBs

RTDBs employ three kinds of transactions, also discussed in databases. These are:

- *Write-only transactions*, writing a sensed value (from the outside environment) to the database.
- *Update transactions*, computing derived data and storing it into the database.
- *Read-only transactions*, reading a value from the database.

As discussed above, RTDBs should process and make data available for user applications with regards to time constraints. Timing requirements are not emphasized in databases, as their main goal is to maximize the total throughput. As in real-time systems we associate a *deadline* with each transaction in a RTDB. The deadline is a point in time when the execution of a transaction must be finished, i.e. the time when the results of executing the query should become available to the user. Depending on the transaction semantics, different kinds of deadlines exist in RTDBs:

- *Hard* deadlines, where the deadlines cannot be missed or else the implication will be a catastrophe. We say the value imparted to the system is negative at a deadline miss.
- *Firm* deadlines, where the deadlines may be missed. We say the value imparted to the system is zero after the a deadline miss. If a transaction delivers its service too late, the system cannot draw values from the tardy service.

- *Soft* deadline, where the deadlines may be missed. We say that the value imparted to the system is gradually decreased to zero after a deadline miss. A transaction with a soft deadline may still deliver its service late, as the system will still draw some value from the tardy service.

Hence, in RTDBs processing of transactions must take their different characteristics into account. Since meeting deadlines is the goal, it is important to understand how transactions are scheduled and how their scheduling relates to time constraints. Scheduling techniques used in real-time systems can also be applied here, e.g. earliest deadline first (EDF) where the transactions are executed in the order determined by their absolute deadlines. Scheduling is not further discussed here and the reader is referred to literature, e.g. [26, 27, 7].

Another issue is *concurrency control*, i.e. how to satisfy consistency among the data accessed by transactions. Here the notion of isolation is emphasized, i.e. for each pair of transaction T_i and T_j, the result of executing T_j should become visible to T_i before T_i has started or after T_i has finished. In other words, data objects accessed by T_i must not be changed by T_j during the execution of T_i. Here it is important to *serialize* the schedules with regards to consistency and also time constraints in the case of RTDBs.

2.1.3 Trading off Quality for Timeliness

Since time constraints in RTDBs are important, it is acceptable in some cases to trade off quality for timeliness. In other words, we can tolerate that a transaction delivers a service of lower quality in the exchange for reduced timing faults. In [23], several strategies with regards to completeness, accuracy and consistency have been pointed out.

Considering completeness, a RTDB can reject a set of write-only or update transactions when the system is overloaded. The choice of rejection can be based on the criticalness of the data being stored in the database.

Considering accuracy, a transaction may not be able to access a data (e.g. due to freshness) or the data itself may not be available (e.g. in the case when another transaction is writing to the data). Instead other methods resorting to approximate query evaluation can be used [11, 31]. In [31], a query processor produces approximate answers if there is not enough time available. The accuracy of the answer increases monotonically as the computation time increases. A relational database system proposed in [11], can produce approximate answers to queries within certain deadlines. Approximate answers are provided by processing a segment of the database by sampling. Given a time quota, the sizes of the samples are computed and an estimation of the query is provided. The correctness of the estimation is improved as the number of samples increase.

Further, if a certain data object is not available or stale, various approximation methods can be used. Approximating a data object can be done by numerical analysis, e.g. extrapolation.

Turning to consistency, strict serializability can be relaxed, giving shorter response time. For example, a transaction can be executed in spite of concurrent updates to the data objects used by the transaction (which in the traditional sense violates consistency) [22]. Such relaxations allow more transactions to execute concurrently thereby improving performance.

2.2 Feedback Control Scheduling

We now give the framework of a typical *feedback control scheduling* (FCS) structure. Further, a general feedback control design methodology is discussed, followed by a description of each of the steps in the method.

A typical structure of a feedback control system is given in Figure 2.1. It consists of a *controlled system* and a *controller*. Input to the controller is the difference between the *performance reference* (y_r), below referred to as simply reference, and the *controlled variable* (y). The goal is to control the controlled system such that the controlled variables converge towards their corresponding references. The controller changes the value of the controlled variables by adjusting a set of *manipulated variables* (u), which are inputs to the controlled system. We refer to *closed loop* control if the controlled variables are used to control the controlled system. In contrast, the controlled variables are not used when *open loop* control is applied and, hence, the controlled system is controlled without the knowledge of the controlled variables. The system in Figure 2.1 applies closed loop control.

As depicted in Figure 2.1, the controlled system consists of an *actuator*, which adjusts the "configuration" of a *processing plant* according to the manipulated variables. The adjustment of the configuration can take form in different ways and is application dependent. In [20], the configuration of the processing plant is modified by changing the QoS-level for a set of tasks, where each QoS-level of a task is defined by a set of tuples, such as CPU utilization, deadline, period and execution time. By changing the QoS-level the CPU utilization can be adjusted. The configuration adjustment can also be enforced using admission control of arriving tasks. If rejected, the task

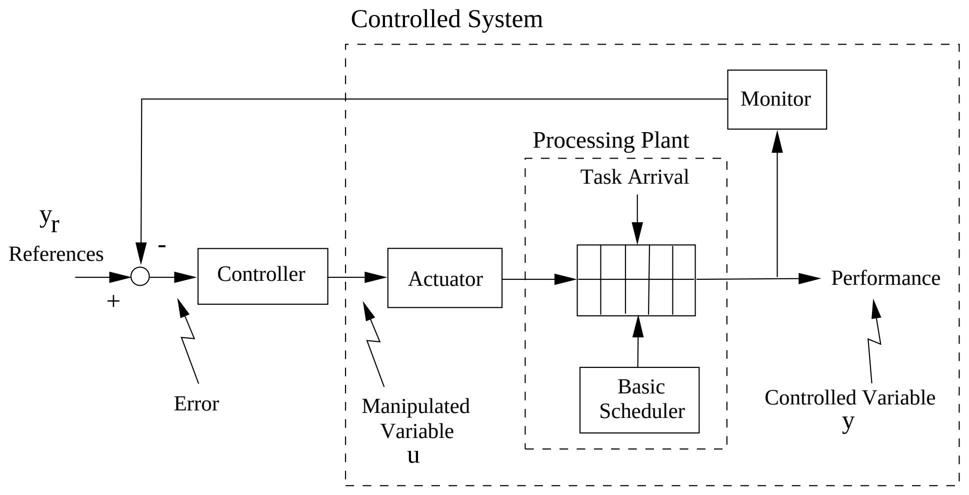

Figure 2.1: Feedback control scheduling architecture

is discarded otherwise it is admitted to the system and allowed to execute. Other ways include changing the frequency or voltage of the actual hardware that the processing plant resides on.

The *basic scheduler* schedules the tasks according to a policy (e.g. EDF). The performance of the processing plant, including the controlled variable, is monitored by the *monitor*.

A general feedback control design methodology is given below. Each of the items are discussed further in the following text.

1. The system designer specifies the desired behavior of the system with a *performance specification* based on steady-state and transient performance.

2. A *dynamic model* of the controlled system is derived for the purpose of performance control.

3. Based on the specification and the dynamic model derived in steps 1-2, a controller is designed.

2.2.1 Specification

A designer can specify the desired behavior of the controlled system with regards to various performance metrics (below referred to as metrics), such

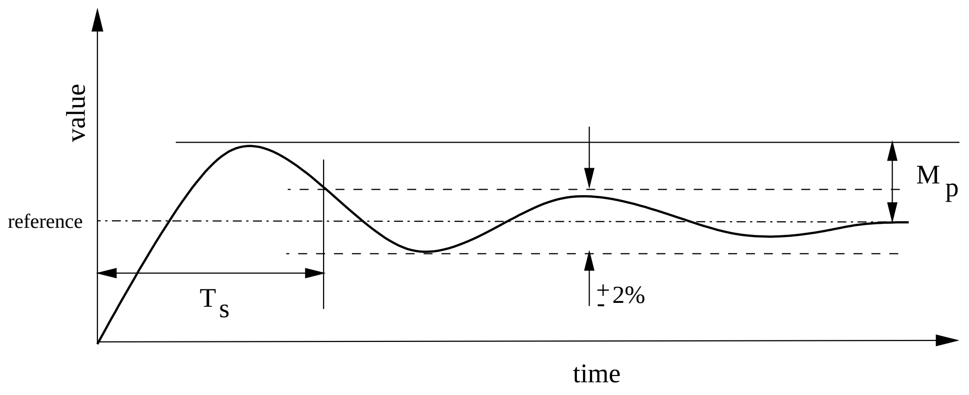

Figure 2.2: Definition of settling time (T_s) and overshoot (M_p)

as deadline miss percentage and utilization. The performance metrics have to be chosen such that the state of the system can be captured. Traditional ways of specifying the behavior such as average performance cannot capture the transient behavior of the system in response to changes to the environment. Here, by the use of *time-domain specifications* or *transient specifications* one can specify the responsiveness and efficiency of a real-time system adapting to changes in the run-time environment. Figure 2.2 illustrates some of the time-domain specifications used for control systems. The *settling time* (T_s) is the time it takes the system transients to decay to a certain level. The *overshoot* (M_p) is the maximum amount the system overshoots its final value divided by its final value (often given as percentage).

Further, a designer may consider the *steady-state error*, which is the difference between the values of a controlled variable in steady state and its reference. *Stability* refers to whether an input to the closed loop (i.e. a reference) causes bounded outputs (i.e. controlled variables). We define stability and present methods for testing stability and computing steady-state error in section 2.2.4.

Further issues such as *sensitivity* and *robustness* are also mentioned in control theory literature. In the case of sensitivity, it is of interest to investigate the impact of various disturbances (e.g. arrival and termination of tasks) on the feedback control loop. Because the designs of control systems are based on simplified models, it is also interesting to know how accurate the model has to be for the design to be successful. Here robustness of a controller to model errors is important. The issues of sensitivity and robustness are out of the scope of this report and the reader is referred to literature in control theory [29, 9].

2.2.2 Modeling

The system designer establishes a dynamic model of the real-time system for the purpose of performance control. A dynamic model describes the mathematical relationship between the control input and the controlled variables of a system with differential/difference equations or state space matrices. In modeling it is often convenient to split a system into interconnected subsystems. A model for each subsystem is then derived and by merging the models one can establish a model of the entire system.

Two different approaches can be used to establish the dynamic model of a system. With an analytical approach, a system designer describes the system directly with mathematical equations based on the knowledge of the system dynamics. Example of such modeling can be found in [25], where a liquid tank model is used. Each accepted task contributes to an amount of liquid in the tank, where as termination of a task result in a decrease in liquid. The level of the liquid corresponds to the CPU utilization.

When an analytical model is not available, a system identification approach can be used to estimate the system model based on profiling experiments [30]. In the latter case, a model can be generated by using experimental data from e.g. transient response and frequency response. Example on this work can be found in [21], where a statistical model is tuned by system profilation.

2.2.3 Controller Design

Given a performance specification and a model, one can design a controller based on existing mathematical techniques, such as root locus, frequency response and state space design. Using mathematical techniques enable us to derive analytic guarantees on the transient and steady-state behavior of the system. The choice of design methodology depends on the desired performance (from the specification) and the accuracy of the model.

It is known that *proportional integrator* (PI) controllers do not require an accurate model [28]. However, they also show relatively poor performance compared to other controllers designed by frequency response or state space. Below we give an overview of the concept of PI controllers, as they are used in this work.

PI controllers work on the following principle. If the controlled variable is smaller than the reference, increase the manipulated variable. If the controlled variable is greater than the reference, decrease the manipulated variable.[1]

We define the *error*, $e(t)$, as the difference between the reference and the controlled variable, i.e. $e(t) = y_r(t) - y(t)$. As described above the goal is to minimize $e^2(t)$ for all t. This is achieved by the PI controller using two different parts, the proportional and the integral controller. The proportional controller (P controller) is given by,

$$u(t) = K_P e(t)$$

where K_P is the *proportional gain*. We can view the P controller as an amplifier that adjusts the manipulated variable $u(t)$ up or down. A system with a P controller may have a steady-state offset in response to a constant reference and may not be entirely capable of rejecting a constant disturbance [28]. One way to improve the steady-state accuracy is to introduce integral control, hence forming PI control. A PI controller is given by,

$$u(t) = K_P(e(t) + K_I \int_{t_0}^{t} e(\xi)d\xi) \tag{2.1}$$

$$u(k) = K_P(e(k) + K_I \sum_{j=0}^{k} e(j)) \tag{2.2}$$

where t_0 denotes the point in time when control is started. The discrete version of equation (2.1) is given by equation (2.2). From (2.1), we can see that if the error persists during a period of time, the integral term increases in value and, consequently, the manipulated variable increases. An increase in manipulated variable pushes for changes in the controlled variable. Therefore, the steady-state error is in general zero when integral control is used. The same effect occurs for equation (2.2), as the integral is replaced by a summation.

The designer needs to tune K_P and K_I based on the performance specification and the model. There are various ways of tuning the parameters such as root locus and Ziegler-Nichols tuning method. These methods are not further discussed here as they require knowledge in transform and control theory. The reader is referred to literature in control theory [28].

2.2.4 Stability and Steady-State Error

In this work, we consider stability in terms of the Bounded-Input-Bounded-Output (BIBO) relation. A system is defined as BIBO stable if a bounded

[1]We assume that an increase in manipulated variable results in an increase in controlled variable and vice versa.

input gives a bounded output for every initial value of the input. When the word *stable* is used without further qualification in this text, BIBO stability is considered. Below a test for stability is given. The underlying theory of the methods discussed below is given in [29].

There are various methods for investigating stability. Given the z-transform of a controller, $C(z)$, and the model, $P(z)$, the closed loop is stable if the roots of the denominator (i.e. the poles) of the closed loop transfer function,

$$H(z) = \frac{C(z)P(z)}{1 + C(z)P(z)}$$

are within the unit circle. If this is ensured, then the closed loop system is stable.

Further, when analyzing control systems, it is important to calculate the steady-state error of the controlled variables, i.e., the difference between a controlled variable in steady-state and its reference. Assume a simple feedback system, as shown in Figure 2.1. If the system is stable, then the steady-state error of a controlled variable,

$$E_{ss} = y_r - \lim_{z \to 1}(z-1)\frac{y_r z}{z-1}\frac{1}{1 + C(z)P(z)}$$

is computed by taking the difference of the reference, y_r, and the steady-state value of the controlled variable, which is derived by applying the final value theorem to the closed loop transfer function [29]. Observe, in order to use the final value theorem, the stability of the system must be proved.

2.3 Imprecise Computing Techniques

In real-time systems, a deadline is associated with each task. In the case of hard real-time systems, the result of missing a deadline can be catastrophic, i.e. the value imparted to the system is negative. Various factors such as varying execution time make meeting all deadlines at all times difficult. Here, the *imprecise computation technique* can minimize this difficulty. It prevents timing faults and achieves graceful degradation by giving the user an approximate answer, in return for timeliness guarantees.

The main idea is to divide a task into one mandatory and one or several optional subtasks. The mandatory subtask has the highest priority and must be completed before its deadlines. If the system is not overloaded, the optional subtasks are executed as well, producing more accurate results. Existing imprecise computation techniques are classified into several categories [18]:

- *Monotone* computations, where the quality of a tasks intermediate results does not decrease as it executes longer. Practically, the results of a task are recorded at appropriate times during its execution. The task is then able to return an answer along with an error indication at any time during its execution. This method for returning imprecise results is called the *milestone approach*. The drawback of this method is the overhead caused by storing the intermediate results.

- Use of *sieve functions*, where computational steps can be skipped to reduce time. This requires however more complex scheduling algorithms, since the execution of a step must be planned in advance, such that the completion of the task before the deadline can be guaranteed. In radar applications, the step that computes noise estimation may be skipped. During transient overloads, an old estimation of the noise may be used.

- In applications where neither the milestone nor the sieve functions can be applied, one can use *multiple versions* of a task. Here each version produces results of varying degree of impreciseness in return for resource needs (such as execution time etc). In the case of a transient overload, the system can choose to execute an alternative version of a task. The drawback of this method is the overhead caused by storing multiple versions as well as the increased scheduling complexity due to the same reasons described for sieve functions.

Chapter 3

Problem Formulation

3.1 Background

It is known that RTDBs may become overloaded, since the load applied on the systems cannot be predicted and various run-time estimates are difficult to accurately estimate. Initial research has shown that by adapting feedback control scheduling [20] and dynamic update policy [14] (the QMF approach), user transactions and updates can be dynamically balanced to guarantee transaction miss percentage and freshness requirements at the same time. The deadline miss percentage of user transactions can be decreased by switching the update policy of one or more data objects from immediate to on-demand policy, giving a higher CPU power to user requests. In contrast, improved data freshness can be achieved by favoring updates. Intuitively, during transient overloads, more effort can be put on user transactions in order to keep the miss percentage low. Whereas when the load is small, more effort can be put on keeping the data fresh and consistent. The QMF approach has shown that the behavior of RTDBs can be controlled to a greater extent compared to a set of simple baselines.

Another possible approach that has been successfully applied in overload management is imprecise computation [18]. Here, we can trade off execution time for the quality of a certain task or in our case a transaction. By doing so we are able to prevent timing faults and we can to a greater extent control the behavior of RTDBs. The main strategy is as follows. During transient overloads, the quality of the transactions can be degraded, thus lowering the resource needs of the updates and the user transactions. Lowering the resource needs results in fewer timing faults, more transactions admitted and, consequently, more users requests serviced. Similarly, when the system

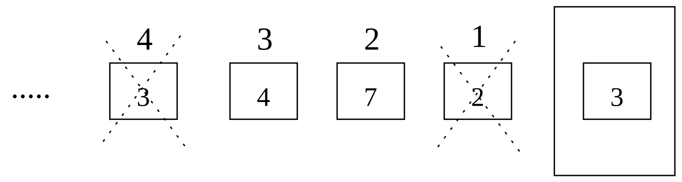

Figure 3.1: A set of arriving updates. Updates one and four are discarded since their value does not deviate much from the currently stored value.

is underutilized, the quality of the transactions and the service provided is increased. Associated with imprecise computing is a certain *error* that indicates the degree of service quality provided.

Integrating imprecise computation with feedback control scheduling should provide a robust way of managing overloads in RTDBs. Here, the database administrator can specify the lowest level of quality tolerated during overloads. By the use of feedback control scheduling, service quality can dynamically be adjusted, such that the desired quality of service can be maintained.

3.2 Problem Analysis

In a RTDB, the notion of imprecision may be applied at data object and/or transaction level. For a data object representing a real-world variable, we can allow a certain degree of deviation compared to the real-world value. Hence we relax the temporal consistency requirement. If such a deviation can be tolerated, arriving updates may be discarded and, hence, the saved CPU power can be allocated to other transactions. Figure 3.1 illustrates this concept. Here we allow the value of a certain data object to deviate from the real-world value by one unit at most. Initially, the value of the data object is three. Since the first update deviates from the initial value by one unit, this update is discarded. The next update cannot be discarded since its value deviates by more than one unit.

In general, we let the *data error*,

$$DE_i = f(CurrentValue_i, UpdateValue_j)$$

of a data object d_i be a function of the current value ($CurrentValue_i$) of d_i, and the update value ($UpdateValue_j$) of the latest arrived transaction (T_j) that updated or was going to update d_i. Note, T_j also denotes update transactions that were going to update d_i, but that were discarded. The data error gives an indication of how much the values of data objects stored in the RTDB deviate from the corresponding values in the external environment.

We introduce the notion of *maximum data error*,

$$MDE = g(PerformanceVariable_1, \ldots)$$

as a function of a set of performance variables giving the current state of the RTDB (e.g. load, deadline miss percentage). The variable MDE gives the maximum data error tolerated considering a set of performance variables. The idea is to vary MDE according to the system performance. If the RTDB is overloaded, we may increase the data error tolerance in exchange for increased saved resource due to discarding update transactions. The data error is adjusted by the following criteria. An update transaction (T_j) is discarded if the data error of the data object (d_i) that is to be updated by T_j is less or equal to MDE (i.e. $DE_i \leq MDE$). Otherwise, the update is committed (executed). In both cases the time stamp of d_i is updated, hence we say that the value of d_i has been revised. In the example above, we define DE_i as,

$$DE_i = |CurrentValue_i - UpdateValue_j|$$

and set MDE to one.

Introducing impreciseness in transactions gives us another dimension by which we can trade off execution time for quality. For transactions, there are a variety of imprecise computation models one can consider. These include milestone, use of sieve functions and multiple version (see section 2.3). The main idea is to logically divide a transaction into a mandatory and one or more optional parts. The mandatory subtransaction is necessary for an acceptable result and must be computed to completion before the transaction deadline. During overloads, the optional parts can be discarded and consequently decreasing the execution time and required resources for the transaction. Discarding one or more optional parts give rise to a certain error which we call the *transaction error*.

3.3 Objective

The objective of this thesis is to investigate how impreciseness in form of data and transaction can be applied and maintained according to a given QoS specification using feedback control scheduling. This includes:

1. Investigating how impreciseness can be applied at data object level.

2. Investigating how impreciseness can be applied to user transactions.

3. Defining a QoS specification in terms of data and user transaction impreciseness.

4. Developing algorithms for managing data and user transactions quality such that a given QoS specification can be satisfied.

5. Evaluating the performance of the proposed algorithms.

It should be noted that in this work, we do not consider derived data. Hence, our data model only addresses base data. The data model is discussed in more detail in section 4.1.3.

Chapter 4

Approach

The following chapter gives a description of our approach. First, the data and the transaction models are given, followed by the definition of a QoS specification. The rest of this chapter is devoted to the presentation of the proposed algorithms, FCS-IC-1 and FCS-IC-2.

4.1 Data and Transaction Model

4.1.1 Database Model

We consider a firm RTDB model, in which tardy transactions, i.e. transactions that have missed their deadlines, add no value to the system and therefore are aborted. We consider a main memory database model, where there is one CPU as the main processing element.

4.1.2 Transaction Model

Transactions are classified either as *update transactions* or *user transactions*:

- Update transactions arrive periodically and may only write to base data objects.

- User transactions arrive aperiodically and may read temporal and read-/write non-temporal data. The inter-arrival time of user transactions is exponentially distributed.

User and update transactions (T_i) are assumed to be composed of one *mandatory subtransaction* (M_i) and $\#O_i$ *optional subtransactions* ($O_{i,j}$, $0 \leq$

$j \leq \#O_i$). For the remainder of the paper, we let $t_i \in \{M_i, O_{i,1}, \ldots, O_{i,\#O_i}\}$ denote a subtransaction of T_i.

We use the milestone approach (see section 2.3) to transaction impreciseness. Here, we have divided transactions into subtransactions according to the milestones. A mandatory subtransaction is completed when it is completed in a traditional sense. The mandatory subtransaction is necessary for an acceptable result and must be computed to completion before the transaction deadline. The optional subtransactions depend on the mandatory subtransaction and may be processed if there is enough time or resources available. It is assumed that all subtransactions (t_i) of a transaction arrive at the same time as the transaction (T_i). The first optional subtransaction (i.e. $O_{i,1}$) becomes ready for execution when the mandatory subtransaction is completed. In general, an optional subtransaction, $O_{i,j}$, becomes ready for execution when $O_{i,j-1}$ (where $2 \leq j \leq \#O_i$) is completed. Hence, there is a precedence relation given by $M_i \prec O_{i,1} \prec O_{i,2} \prec \ldots \prec O_{i,\#O_i}$.

We set the deadline of all subtransactions (t_i) of a transaction to the deadline of the transaction (T_i). A subtransaction is terminated if it is completed or has missed its deadline. A transaction (T_i) is terminated when its last optional subtransaction (i.e. $O_{i,\#O_i}$) is completed or one of its subtransactions has missed its deadline. In the latter case, all subtransactions that are not completed are terminated as well.

Each transaction and its respective subtransactions are characterized by the following attributes.

The following attributes are common to all transactions and subtransactions:

- EET_i, the estimated (average) execution time of T_i. $EET_i[t_i]$, the estimated (average) execution time of t_i.
- AET_i, the average execution time of T_i. $AET_i[t_i]$, the average execution time of t_i.
- AT_i, the arrival time of T_i. We set $AT_i[t_i] = AT_i, \forall t_i$.

In addition to the common attributes, we apply the following attributes to update transactions:

- AV_i, the average update value of T_i.
- P_i, the period of T_i. We set $P_i[t_i] = P_i, \forall t_i$.
- D_i, the relative deadline of T_i. We set $D_i[t_i] = D_i = P_i, \forall t_i$.

- EU_i, the estimated utilization of T_i, $EU_i = EET_i/P_i$.
  $EU_i[t_i]$, the estimated utilization of t_i, $EU_i[t_i] = EET_i[t_i]/P_i$.
- AU_i, the average utilization of T_i, $AU_i = AET_i/P_i$.
  $AU_i[t_i]$, the average utilization of t_i, $AU_i[t_i] = AET_i[t_i]/P_i$.

In addition to the common attributes, we apply the following attributes to user transactions:

- EIT_i, the estimated inter-arrival time of T_i. We set $EIT_i[t_i] = EIT_i, \forall t_i$.
- AIT_i, the average inter-arrival time of T_i. We set $AIT_i[t_i] = AIT_i, \forall t_i$.
- D_i, the relative deadline of T_i. We set $D_i[t_i] = D_i = AIT_i, \forall t_i$.
- EU_i, the estimated utilization of T_i, $EU_i = EET_i/EIT_i$.
  $EU_i[t_i]$ the estimated utilization of t_i, $EU_i[t_i] = EET_i[t_i]/EIT_i$.
- AU_i, the average utilization of T_i, $AU_i = AET_i/AIT_i$.
  $AU_i[t_i]$, the average utilization of t_i, $AU_i[t_i] = AET_i[t_i]/AIT_i$.

For update transactions, we assume that there are no optional subtransactions (i.e. $\#O_i = 0$). Hence, each update transaction is composed of a single mandatory subtransaction. This assumption is based on the fact that updates do not use complex logical or numerical operations and, hence, have a lower execution time than user transactions.

In our user transaction model, we assume that the number of optional subtransactions is greater or equal to zero (i.e. $\#O_i \geq 0$). However, for imprecise transactions it is required, inherent by the assumptions, that transactions consist of one mandatory and at least one optional subtransaction. Our work primarily focuses on imprecise transactions.

It should be noted that a feature in our transaction model is that it models systems in unpredictable environments where the actual CPU utilization of transactions is time-varying and unknown to the scheduler. With such modeling it is possible to use feedback control loops to dynamically adapt the scheduling to load variations at run-time.

4.1.3 Data Model and Data Management

As described in chapter 2, data can be classified into two classes, temporal and non-temporal. For temporal data, we only consider base data, i.e. data that hold the view of real-world and updated by sensors. In our model, a data object d_i has the following attributes.

- CV_i, the current value of d_i.
- TS_i, the time stamp of d_i.
- AVI_i, the absolute validity interval of d_i.

A base data object d_i is considered temporally inconsistent or stale if the current time is later than the timestamp of d_i followed by the absolute validity interval (avi) of d_i, i.e.,

$$CurrentTime > TS_i + AVI_i.$$

Hence, the absolute validity interval is the length of the time a temporal data object remains fresh or temporally consistent. We set the absolute validity interval of d_i to twice the period of T_j, i.e. $AVI_i = 2 \times P_j$, where P_j denotes the period of the transaction updating d_i. This is done in order to guarantee that the absolute consistency of d_i is not violated [23].

We let *data error*,

$$DE_i = 100 \times \frac{|CV_i - V_j|}{|CV_i|}(\%) \tag{4.1}$$

where V_j is the value of the latest arrived transaction (T_j) that updated or was going to update d_i. For example, consider Figure 3.1. Here, in the time period between the arrival and completion (commitment) of the second update transaction, we have a data error of $100 \times \frac{|3-7|}{3}\% = 100 \times \frac{4}{3}\%$ (the current value is three since the first update transaction was discarded). Once the second update transaction commits, the data error becomes zero since the current value stored in the database is equal to the value of the last arrived update transactions, i.e. update two. The data error remains zero until update transaction number three arrives.

4.1.4 Admission Control, Scheduling and Concurrency Control

For user transactions, admission control is applied to reduce the chance of potential overload. A newly arrived user transaction can be admitted to the system if the required CPU utilization is currently available. A more detailed description of admission control is given in section 4.3.4.

We apply earliest deadline first (EDF) [17] to schedule user transactions. Update transactions and mandatory user subtransactions are considered important and, thus, scheduled before optional user subtransactions.

For concurrency control, we employ two phase locking with highest priority (2PL-HP) [2], in which a low priority transaction is aborted and restarted upon a conflict. 2PL-HP is selected since it is free of priority inversion.

4.2 Performance Metrics and QoS specification

In our approach, the database administrator (DBA) can explicitly specify the required database QoS, which defines the desired behavior of the database. An overview of performance specifications and metrics for real-time systems in general can be found in [19]. In this work we adapt steady-state and transient-state performance metrics. The metrics are as follows:

- *Deadline Miss Percentage of Mandatory User Subtransactions* (M^M). In a QoS specification the DBA can specify the deadline miss percentage of the mandatory subtransactions. M^M is defined as,

$$M^M = 100 \times \frac{\#DeadlineMiss^M}{\#Terminated^M} (\%) \tag{4.2}$$

  where $\#DeadlineMiss^M$ denotes the number of mandatory subtransactions that have missed their deadline and $\#Terminated^M$ is the number of terminated mandatory subtransactions. We exclusively consider user transactions admitted to the system.

- *Deadline Miss Percentage of Optional User Subtransactions* (M^O). M^O is the percentage of optional subtransactions that have missed their deadline. M^O is defined by,

$$M^O = 100 \times \frac{\#DeadlineMiss^O}{\#Terminated^O} (\%) \tag{4.3}$$

  where $\#DeadlineMiss^O$ denotes the number of optional subtransactions that have missed their deadline and $\#Terminated^O$ is the number of terminated optional subtransactions. We exclusively consider user transactions admitted to the system.

- *Maximum Data Error* (MDE). This metric gives the maximum data error tolerated for the data objects. Note that MDE is common to all data objects. An update transaction (T_j) is discarded if the data error of the data object (d_i) that is to be updated by T_j is less or equal to MDE (i.e. $DE_i \leq MDE$). Increasing MDE results in a higher error tolerance (i.e. we allow less precise data) and, hence, more updates are discarded.

- *Overshoot* (M_p) is the worst-case system performance in the transient system state. Overshoot is applied to M^O, M^M, and MDE.
- *Settling time* (T_s) is the time for the transient overshoot to decay and reach the steady state performance.
- *Utilization* (U). In a QoS specification the DBA can specify a lower bound for the utilization of the system.

We define *Quality of Data* (QoD) in terms of MDE. An upgrade in QoD refers to a decrease in MDE. In contrast a degrade in QoD refers to an increase in MDE.

We measure user transaction quality in terms of deadline miss percentage of optional subtransactions, i.e. M^O. This is feasible in the case when optional subtransactions contribute equally to the final result.[1]

The DBA can specify a set of target levels or references for the metrics listed above. A QoS specification can hold the following: $M_r^M = 1\%$ (reference M^M), $M_r^O = 10\%$ (reference M^O), $MDE_r = 2\%$ (reference MDE), $U \geq 80\%$, $T_s \leq 60s$ and $M_p \leq 30\%$. This gives the following transient performance specifications: $M^M \leq M_r^M \times M_p = 1.3\%$, $M^O \leq 13\%$ and $MDE \leq 2.6\%$. The QoS given above requires some explanation. We have specified that the average miss percentage for mandatory subtransactions to be 1%, maximum 1.3%. This does not mean that the system will keep the average miss percentage at 1% at all times, but that the system will try to keep the miss percentage at the specified reference during overloads. When the system is underutilized, one can expect to see small or zero miss percentage.

4.3 Feedback Control Scheduling Architecture

In this section we give an overview of the feedback control scheduling architecture used in this work. Further, we identify a set of control related variables, i.e. manipulated variables and controlled variables. The general outline of the feedback control scheduling architecture is given in Figure 4.1.

Admitted transactions (or subtransactions) are placed in the ready queue. The transaction handler manages the execution of the transactions. At each sampling instant, the controlled variables, miss percentages and utilization,

[1]This is a simplification, which we intend to address in future work by computing errors considering the amount of completed subtransactions having different values. One can then specify a QoS with the notion of transactions error.

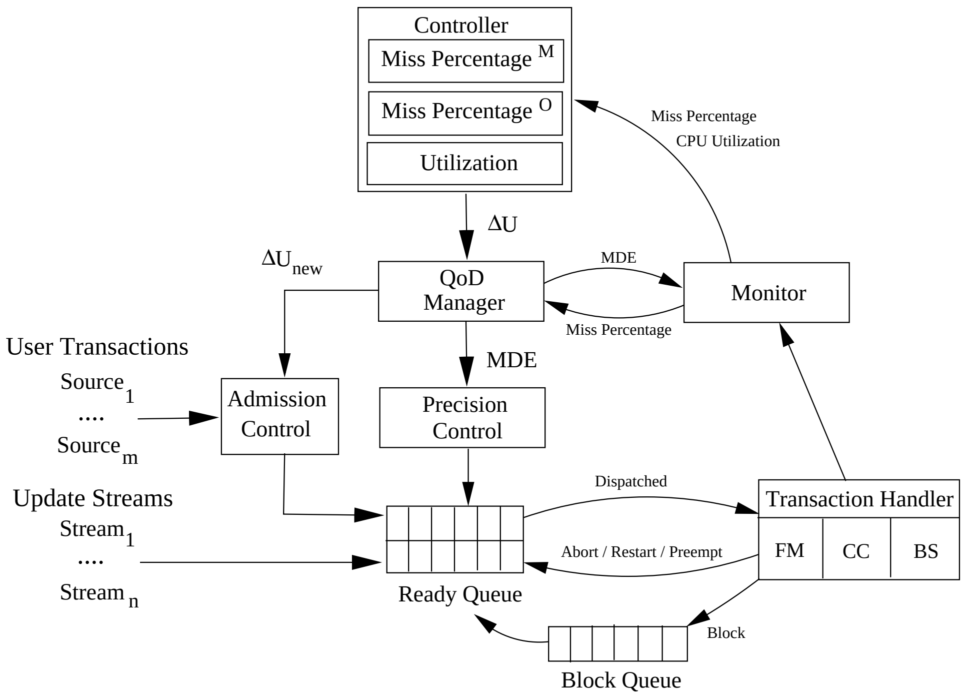

Figure 4.1: Feedback control scheduling architecture

are monitored and fed into the miss percentage and utilization controllers, which compare the performance references, M_r^M, M_r^O, and U_r, with the corresponding controlled variables to get the current errors and compute a change, denoted ΔU, to the total estimated requested utilization. We refer to ΔU as the manipulated variable. Based on ΔU, the QoD manager changes the total estimated requested utilization by adapting the QoD level (i.e. adjusting MDE). The precision control then schedules the update transactions based on MDE. The portion of ΔU not accommodated by the QoD manager, denoted ΔU_{new}, is returned to the admission controller, which enforces the remaining utilization adjustment.

4.3.1 Streams and Sources

The streams ($Stream_i$) generate update transactions, whereas the user transactions are generated and submitted by sources ($Source_i$).

4.3.2 Transaction Handler

The transaction handler provides a platform for managing transactions. It consists of a freshness manager (FM), a unit managing the concurrency control (CC) and a basic scheduler (BS).

Freshness Manager FM checks the freshness before accessing a data object, using the timestamp and the avi of the data. If the transaction (T_{user}) is accessing a stale data object (d_i) there are two options:

- *Wait for next update*, if the estimated execution time of T_{user} (EET_{user}) added to the next arrival time of the transaction updating d_i is earlier than the deadline of T_{user} (D_{user}).
- *Use stale data* if the above condition does not hold.

Concurrency Control We employ two-phase locking with highest priority (2PL-HP) [2, 12] for concurrency control, where a conflict is resolved by allowing the transaction with the highest priority (in our case the transaction with the earlier deadline) to lock the data object. 2PL-HP is chosen since it is free from priority inversion and has well known behavior.

Basic Scheduler The FCS architecture includes a basic scheduler that schedules admitted tasks according to a scheduling policy (e.g. Earliest Deadline First, Rate Monotonic etc). We have chosen earliest deadline first (EDF), due to its high performance [17, 27]. Since the properties of the scheduling policy can have significant impact on the design of the feedback loop [20], our design and model tuning are based on EDF. Conceptually, transactions are scheduled in a multi-level queue system. Update transactions and mandatory user subtransactions are placed in the highest priority queue, whereas optional user subtransactions are placed in a lower priority queue.

4.3.3 Feedback Control

As discussed above, controllers are used to compute a change to the total estimated requested utilization, such that a given QoS specification can be satisfied. A detailed discussion regarding the design and tuning of the controllers is given in section 4.5.

4.3.4 Admission Control

The admission controller (AC) controls the flow of transactions into the database. When a new transaction is submitted to the database, AC decides whether it can be admitted to the system. Given the total estimated requested utilization (U_{EstReq}), the AC admits a transaction (T_i) if the estimated utilization of admitted subtransactions and EET_i is less or equal to U_{EstReq}.

In the current setting, all optional subtransactions are taken into account when admitting a transaction. A negotiation process may be added, so that a transaction is admitted with fewer optional subtransactions. This gives us another way of regulating M^O.

4.3.5 Precision Control

The precision controller discards an update transaction writing to a data object, having an error less than the maximum data error allowed, i.e.,

$$DE_i \leq MDE$$

where DE_i is the current data error of the data object d_i that is to be updated by T_j. If the data error of d_i is greater than MDE, then the update transaction is executed. In both cases the time stamp of d_i is updated.

4.4 System Modeling

In system modeling, we identify the main components of the controlled system, the data values that flow in and out of them and the mathematical relationships between these values. This section describes our approach to system modeling. The particular form of the models we construct (linear models) enables us to use a set of analytical methods that are available in control theory. For analysis purposes, we apply the principles of z-transform theory.

There has been several initial work on modeling real-time systems using linear models [20, 21]. In [20], an analytical approach to system modeling is used whereas in [21] a statistical (ARMA) model is fit to historical measurements of the controlled system. In our work, we adapt the model used in [20], since it has been well-studied and shown to provide good results. The model is shortly discussed below. The reader is referred to [20] for a more detailed description.

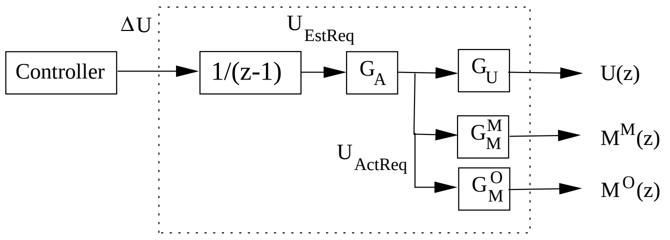

Figure 4.2: Model of the controlled system

The controlled system includes AC, QoD manager, BS, and monitor. The input to the controlled system is the change in the total estimated utilization (ΔU). The output of the system is the controlled variables (i.e. M^M, M^O, and U).

As mentioned earlier, it is hard to model a real-time system due to its underlying complexity, resulting in a non-linear and time varying system. However, since PI controllers are known for their robustness, it is often adequate to model such a system with a linear model for the purpose of control design methodology. The block diagram of the system is depicted in Figure 4.2.

We use the notation, where $X(k)$ and $X(z)$ denote the time domain and the z-domain respectively of the variable X. The goal is to derive a *transfer function* describing the relation between the manipulated variable, i.e. $\Delta U(z)$, and the controlled variables, i.e. $M^M(z)$, $M^O(z)$, and $U(z)$.

Starting from the control input, the estimated requested utilization is the integration of the control input ΔU. Formally, the estimated requested utilization in period $k+1$,

$$U_{EstReq}(k+1) = U_{EstReq}(k) + \Delta U(k) \tag{4.4}$$

$$\frac{U_{EstReq}(z)}{\Delta U(z)} = \frac{1}{z-1} \tag{4.5}$$

is the summation of the estimated requested utilization ($U_{EstReq}(k)$) and estimated utilization adjustment $\Delta U(k)$ in period k. Note, the estimated utilization adjustment, $\Delta U(k)$ refers to the desired change in utilization during period $k+1$. The z-transform of equation (4.4) is given in equation (4.5). Further, the ***actual requested utilization*** ($U_{ActReq}(k)$) may differ from $U_{EstReq}(k)$. This is due to incomplete knowledge about the controlled system, e.g. unknown execution times of the transactions and data conflicts.

Therefore, $U_{ActReq}(k)$ is computed according to the following,

$$U_{ActReq}(k) = G_A \times U_{EstReq}(k) \tag{4.6}$$

where G_A, the *utilization ratio* represents the extent of worst case workload variation in terms of actual total required utilization.

The next step becomes to model the utilization and the miss percentages based on the actual requested utilization. The relationship between U_{ActReq} and U is non-linear due to saturation. This occurs when $U_{ActReq} > 100\%$ (CPU overloaded), resulting in no change of U even if ΔU is varied. This can formalized by the following:

$$U(k) = \begin{cases} U_{ActReq}(k), & U_{ActReq}(k) \leq 100\% \\ 100\%, & U_{ActReq}(k) > 100\% \end{cases} \tag{4.7}$$

The case is somewhat different with the miss percentages $M^M(k)$ and $M^O(k)$. The *schedulable threshold* for mandatory subtransactions, denoted $U^M_{th}(k)$, and optional subtransactions, denoted $U^O_{th}(k)$, is defined as the utilization threshold, in the k^{th} sampling period, for which no deadline miss can be observed for the respective type of subtransaction. When the miss percentages are saturated (i.e. inside the saturation zones given by $U_{ActReq}(k) \leq U^M_{th}(k)$ and $U_{ActReq}(k) \leq U^O_{th}(k)$) no deadline misses are observed, since adjustments of ΔU and consequently $U_{ActReq}(k)$ will not affect the miss percentages (until the utilization becomes greater than the threshold and miss percentages start increasing). However, when outside the saturation zones (i.e. $U_{ActReq}(k) > U^M_{th}(k)$ or $U_{ActReq}(k) > U^O_{th}(k)$), the miss percentage for either subtransaction is increased nonlinearly. Note, since mandatory subtransactions have higher priority than optional subtransactions, the schedulable threshold for mandatory subtransactions is greater than the threshold for optional subtransactions (i.e. $U^M_{th}(k) > U^O_{th}(k)$). One way of linearizing the relationship between $U_{ActReq}(k)$, $M^M(k)$ and $M^O(k)$ is to take the derivative at the vicinity of $U^M_{th}(k)$ and $U^O_{th}(k)$. Hence, we let,

$$G^M_M = \frac{dM^M(k)}{dU_{ActReq}(k)} \qquad (U_{ActReq}(k) = U^M_{th}(k)) \tag{4.8}$$

$$G^O_M = \frac{dM^O(k)}{dU_{ActReq}(k)} \qquad (U_{ActReq}(k) = U^O_{th}(k)) \tag{4.9}$$

where G^M_M and G^O_M denote the *miss percentage factors*. At the vicinity of the schedulable utilization thresholds, we have:

$$M^M(k) = M^M(k-1) + G^M_M(U_{ActReq}(k) - U_{ActReq}(k-1)) \tag{4.10}$$

$$M^O(k) = M^O(k-1) + G^O_M(U_{ActReq}(k) - U_{ActReq}(k-1)) \tag{4.11}$$

From equations (4.4)-(4.11), we can derive a transfer function for each of the controlled variables when it is outside its saturation zone:

Utilization Controller. Under the condition that $U_{ActReq} \leq 100\%$, there exists a transfer function,

$$P_U = \frac{G_A}{z-1}$$

from the control input ΔU to CPU utilization U.

Mandatory Subtransaction Miss Percentage Controller. Under the condition that $U_{ActReq} > U_{th}^{M}$, there exists a transfer function,

$$P_M^M = \frac{G_A G_M^M}{z-1}$$

from the control input ΔU to miss percentage of mandatory subtransactions $M^M(z)$.

Optional Subtransaction Miss Percentage Controller. Under the condition that $U_{ActReq} > U_{th}^{O}$, there exists a transfer function,

$$P_M^O = \frac{G_A G_M^O}{z-1}$$

from the control input ΔU to miss percentage of optional subtransactions $M^O(z)$.

Since the same control design and tuning method is applied to both mandatory and optional subtransaction miss percentage controllers, we use the same symbol $P_M(z)$ to denote $P_M^M(z)$ and $P_M^O(z)$ for the remainder of this report.

4.5 Feedback Controller Design and Tuning

In this section, we discuss different kinds of controllers and how to apply control design methods and analysis to the controllers. Based on the analytical system model derived in section 4.4, we tune a set of model parameters and then use root locus to tune the controllers. Mathematical analysis is applied to evaluate the performance of the tuned controllers.

From the discussions in section 2.2, we learned that feedback control scheduling is able to deal with dynamic systems that are resource insufficient

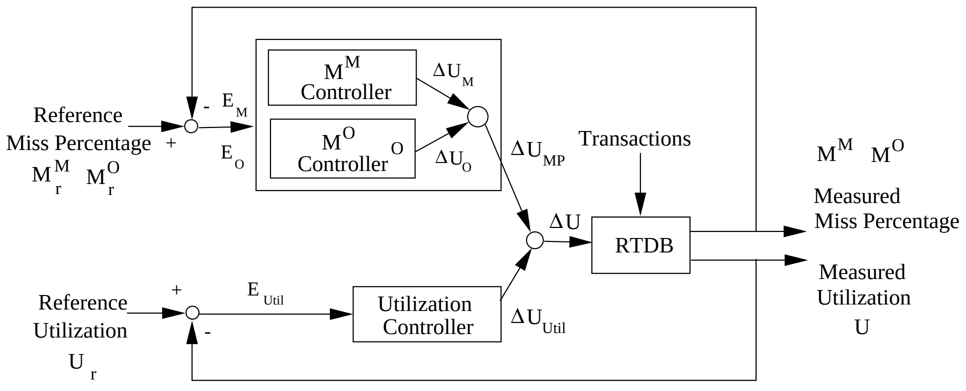

Figure 4.3: Miss percentage / utilization controller architecture

and operating in environments where the load applied on the system cannot be predicted. We employ PI regulators, since compared with other regulators they do not require a precise model of the controlled system.

There have been several FCS policies proposed, such as FC-U, FC-M and FC-UM [20]. FC-U uses a utilization control loop to control the utilization according to a reference, U_r. FC-U can guarantee that the system has zero or low miss percentage in steady state if its reference U_r is less than the schedulable utilization threshold of the system. Note, this requires that the schedulable utilization threshold is known in advance.

FC-M on the other hand uses a miss percentage control loop to directly control the system miss percentage. Compared with FC-U, the advantage of FC-M is that is does not depend on any knowledge of utilization bounds.

Finally, FC-UM integrates the miss percentage and the utilization controllers. This has the advantage that the DBA can simply set the utilization reference to an expected value (based on profiling etc.) that causes zero or low deadline miss percentage. Similarly, the DBA can set the miss percentage references, M_r^M and M_r^O, according to the application requirements.

We use FC-M and an extended version of FC-UM in our algorithms for controlling the quality of data and user transactions (given in section 4.6). An overview of the controllers and how they are integrated is given in Figure 4.3. Depending on the FCS policy being used, different controllers are employed. In FC-M only the miss percentage controllers are used, whereas in FC-UM both miss percentage and utilization controllers are used. Here we assume that our controllers are of single-input-single-output type. By using separate controllers we can achieve modularization and also provide flexibility against varying workload mixtures among mandatory and optional subtransactions. The former statement results in a design methodology where FCS policies can more easily be designed by simply joining controllers that are needed.

Using separate controllers raises the question of integration of the signals from each controller. In the case where miss percentage controllers are used, we first need to integrate the signal from the mandatory and the optional subtransaction miss percentage controllers. If using FC-M the signal from the miss percentage controllers are returned to the QoD manager. However, when a combination of miss percentage and utilization controllers (e.g. FC-UM) is used, an integrated signal from both the miss percentage and the utilization controller is computed and returned (discussed in section 4.5.3).

4.5.1 Miss Percentage Controller

Miss percentage controllers are employed for mandatory and optional subtransactions, respectively, to guarantee their miss percentages. As specified by the QoS specification, there is a target miss percentage associated with mandatory and optional subtransactions (M_r^M and M_r^O respectively). We employ separate controllers for mandatory and optional subtransaction miss percentages to compute control signals, ΔU_M and ΔU_O, based on the current performance error, E_M and E_O, which are the differences between the references and the measured performance (i.e. $E_M = M_r^M - M^M$ and $E_O = M_r^O - M^O$). If the miss percentage of mandatory subtransactions overshoots its reference, then ΔU_M becomes negative as a request to lower the estimated requested utilization. Similarly, if the miss percentage of optional subtransactions overshoots its reference, then ΔU_O becomes negative as a request to lower the estimated requested utilization.

4.5.2 Utilization Controller

When using FC-UM, an utilization controller is used as well. This is done in order to prevent a potential underutilization when all the miss percentage requirements are satisfied due to underutilization. It should be noted that underutilization is avoided in FC-M by either setting $M_r^M \neq 0$ or $M_r^O \neq 0$ [20]. At each sampling period, the utilization controller computes the utilization control signal ΔU_{Util} based on the utilization error, E_{Util}, which is the difference between the reference utilization (U_r) and the current utilization (U).

It is known that accurate execution time estimates may not be available in RTDBs. This is due to data conflicts that result in aborts or restarts of transactions, and also varying data needs of transactions. Consequently, utilization cannot be accurately predicted and, hence, the miss percentages may overshoot. In order to suppress potential overshoots, we have extended FC-UM such that the reference utilization, U_r, is constantly updated on-line.[2] The utilization reference is dynamically updated according to a linear increase/exponential decrease scheme. Initially, U_r is set to 80%. As long as the utilization controller has the control (i.e. the miss percentages are below the references), the utilization reference is increased by a certain step. As soon as one of the miss percentage controllers take over (i.e. one of the miss percentages is above its reference), U_r is reduced exponentially, i.e., we have

$$U_r(k+1) = \frac{U_r(k) + 80}{2}(\%)$$

where $U_r(k+1)$ is the new utilization reference. This is to prevent a potential overshoot due to a too optimistic utilization reference (for which a miss percentage overshoot can be observed). Note, this approach is self adapting and does not require any knowledge about the underlying run-time estimates.

4.5.3 Derivation of Single Control Signal

Derivation of ΔU_{MP} is done as follows.

- If both miss percentage control signals are negative (i.e. $\Delta U_M < 0 \wedge \Delta U_O < 0$, we set,
$$\Delta U_{MP} = \Delta U_M + \Delta U_O$$
to the sum of both control signals. This is done since both miss percentages are above their references and both signals must be considered to compensate for miss percentage overshoots.

- If the above does not hold, we set,
$$\Delta U_{MP} = \min(\Delta U_M, \Delta U_O)$$
to the minimum of the control signals. If one of the control signals is negative (due to an overshoot), we return the negative one to reduce the miss percentage of the corresponding subtransaction's type. If both are positive, the *min* operator provides a smooth transition from one system state to another one (transition between low and high miss percentages among mandatory and optional subtransactions) [20].

[2] In the original FC-UM the reference utilization is not modified on-line.

In the case of FC-M, we set ΔU to ΔU_{MP}. However, when FC-UM is used, we derive,

$$\Delta U = \min(\Delta U_{MP}, \Delta U_{Util})$$

by taking the minimum of ΔU_{MP} and ΔU_{Util}. This is done for the similar reasons as mentioned above.

4.5.4 Integrator Antiwindup

For real-time systems, it may happen that a control variable reaches its miss percentage saturation point (e.g. when U reaches 100%). When this happens the feedback control loop is broken and the system runs as an open loop because the controlled variable will remain unchanged independently of the controller output. This may have an undesired effect, as the error will continue to be integrated, growing to a large value. We say that the integrator "winds up". It is then required that the error has the opposite sign for a long time before a controlled variable reaches its respective reference. The same effect happens when a controlled variable is far away from its reference. For example, consider the case when M^O reaches M_r^O. If M^M is less than M_r^M then E_M is greater than zero and this results in the miss percentage controller (of mandatory subtransactions) accumulating the error. If at a later stage, M^M overshoots its reference, then it may take some time for the miss percentage controller to respond to the high miss percentage. For this reason we use an *integrator antiwindup* mechanism [28, 29, 9].

We implement the integrator antiwindup by using a *conditional integration* [28] technique, where the integration is switched off when a controlled variable is far from its reference. Integration is, thus, used only when a certain condition is fulfilled. We employ the following scheme for conditional integration:

- If ΔU_{Util} is less than ΔU_{MP}, i.e. the utilization controller takes over, all integrators (see equation 4.12) in miss percentage controllers are turned off.

- If ΔU_{Util} is greater than ΔU_{MP}, i.e. one of the miss percentage controllers take over, the integrator of the utilization controller is turned off. Here there are two further options.

  - If ΔU_M is less than ΔU_O, turn off the integrator of the optional subtransaction miss percentage controller, since the mandatory subtransaction miss percentage controller takes over.

  - If ΔU_M is greater than ΔU_O, turn off integrator of the mandatory subtransaction miss percentage controller, since the optional subtransaction miss percentage controller takes over.

Note that in the case of FC-M, the conditional integration scheme is only employed on miss percentage controllers.

4.5.5 Controller Implementation

In this section we present how the controllers are implemented. Since the same control function (with different parameters) is used for all controllers, we use the same symbol $E(k)$ to denote $E_M(k)$, $E_O(k)$, and E_{Util} in the remainder of this section.

A digital version of the PI controller is given below. Equations (4.12) and (4.13) are equivalent, but equation (4.13) is more efficient at run-time.

$$\Delta U(k) = K_P(E(K) + K_I \sum_{j=0}^{k} E(j)) \tag{4.12}$$

$$\Delta U(k) = \Delta U(k-1) + K_P((K_I + 1)E(k) - E(k-1)) \tag{4.13}$$

The z-transform of equation (4.13) is given by:

$$C(z) = \frac{g(z-r)}{z-1} \qquad g = (K_P(K_I+1)), r = \frac{1}{K_I+1}$$

The parameters that need to be tuned for each controller are K_P and K_I. The tuning, based on system model and QoS requirements, is given in section 4.5.6.

4.5.6 Controller Tuning and Performance

To tune the miss percentage controllers, the model parameters G_M^M and G_M^O have to be tuned under the worst case set-up to support a certain QoS guarantee [20]. Tuning of the model parameters is done by profiling the system under the worst case setup.[3]

We have tuned G_M^M and G_M^O by measuring average miss percentage for mandatory and optional subtransactions respectively, under loads of 50-200%, by steps of 10% increase. A graph illustrating the miss percentage is given in Figure 4.4. The vertical bars indicate 95% confidence intervals.

[3]In our work, we have used a simulator to evaluate our algorithms. Detailed information about system related issues is given in chapter 5.

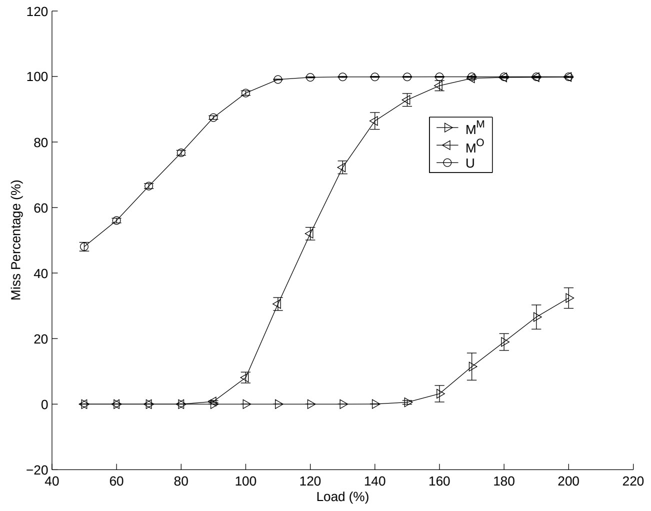

Figure 4.4: System profiling and model tuning

From the graph we can see that the schedulable utilization threshold for optional subtransactions ($U_{th}^{O} \approx 80\%$) is significantly lower than the threshold for mandatory subtransactions ($U_{th}^{M} \approx 150\%$). This is in line with our discussions in section 4.4.

We have turned off AC during system profiling. This allows us to derive the relationship between actual load and the miss percentages without the intervention of an AC. Further, in this profiling we assume that mandatory subtransactions have higher priority than optional subtransactions. Since we must consider the worst-case setting, we have for the profiling scheduled the update transactions before user transactions. This is due to the case when all pending update transactions have an earlier deadline than user transactions and thus higher priority.

Another case that decreases the performance of the system occurs as the number of optional subtransactions decreases. Consequently, the percentage of mandatory subtransactions to the total number of subtransactions increases. This leads to a higher competition among mandatory subtransactions and, hence, may lead to a higher M^M. Also, optional subtransactions will receive less CPU power since there are more mandatory subtransactions with higher priority, resulting in a higher M^O. In our performance evaluations we have set the minimum number of optional subtransactions for each

Controller	K_P	K_I	M_p	T_s
M^M	0.60394	0.18	$\approx 18\%$	≈ 55.2 s
M^O	0.22272	0.18	$\approx 18\%$	≈ 55.2 s
Utilization	0.2906	0.1765	$\approx 18\%$	≈ 75 s

Table 4.1: Controller parameters and time domain performance

user transaction to one (i.e. $\#O_i \geq 1$). In order to assume the worst-case setup. we have for the profiling, set the number of optional user subtransactions to one (i.e. $\#O_i = 1$). From the above we can derive $G_M^M = 0.83$ and $G_M^O = 2.23$, when the load increases from 160% to 170% and 110% to 120%, respectively.

For tuning purposes, we consider the QoS specification given in section 4.2. Having a tuned model of the system and a QoS specification, the next step is to tune the controllers based on the closed loop of the system. As in section 4.4, we use the same symbol $C(z)$ to denote the mandatory and optional subtransaction miss percentage and utilization controllers. Further, we use $M(z)$ and $M_r(z)$ to denote the z-transform of the miss percentage and the reference miss percentage of the mandatory and optional subtransactions, respectively.

Given the model of the controlled system, i.e. P_U and P_M, the transfer function of each feedback control loop,

$$H_M(z) = \frac{C(z)P_M(z)}{1 + C(z)P_M(z)} \qquad H_U(z) = \frac{C(z)P_U(z)}{1 + C(z)P_U(z)} \tag{4.14}$$

can be established. The z-transform of the miss percentages and the utilization,

$$M(z) = \frac{M_r z}{z-1} H_M(z) \qquad U(z) = \frac{U_r z}{z-1} H_U(z)$$

can be derived by using equation (4.14).

We have applied the root locus method in Matlab [1] to tune the controller parameters such that the performance specifications (as given by the QoS specification in section 4.2) are satisfied. Assuming the system workload having a worst-case utilization ratio of two (i.e. $G_A = 2$), we set K_P and K_I according to Table 4.1. Here, we have set the sampling period to 5 seconds.

Below we evaluate the performance of the controllers with regard to overshoot, settling time, stability, and steady-state error.

Time Domain Performance

Given a unit step in Matlab, the transient performance of the tuned feedback controllers are established. As we can see from Table 4.1, the overshoot and the settling time for the miss percentage controllers are 18% and 55.2s, respectively, which meet the requirements given in the QoS specification.

Stability

The closed loop systems are stable since all the poles of $H_M(z)$ and $H_U(z)$ are within the unit circle.

Steady-state Error

Further, the controllers in FC-M and FC-UM can achieve zero steady-state error [20]. Since the closed loop systems are stable, the steady-state error of M^M,

$$\begin{aligned} E_{mss} &= M_r^M - \lim_{z \to 1}(z-1)\frac{M_r^M z}{z-1}\frac{1}{1+C(z)P_M^M(z)} = \\ &= M_r^M - \lim_{z \to 1}(z-1)\frac{M_r^M z}{z-1}\frac{z-1}{1+g(z-r)+G_A G_M^M} = 0 \end{aligned}$$

can be computed by applying the final value theorem to the closed loop transfer functions. Similarly, the final value theorem can by applied to find steady-state errors of M^O (E_{oss}) and U (E_{uss}). This yields that the steady-state errors for M^O and U is zero (i.e. $E_{oss} = E_{uss} = 0$).

4.6 Algorithm Specification

Below, two algorithms for managing a QoS specification in terms of data and user transaction quality are introduced. Both are based on adjusting the estimated requested utilization using feedback control. The utilization adjustment is enforced partially by adjusting the QoD, which requires setting MDE according to the utilization adjustment (ΔU), as described in section 4.3.

For now, we assume the existence of a function $f(\Delta U(k))$ that returns, based on $\Delta U(k)$, the corresponding MDE for period $k+1$, i.e. $MDE(k+1)$. The function f holds the following property. If $\Delta U(k)$ is less than zero, then $MDE(k+1)$ is set such that $MDE(k+1)$ is greater than $MDE(k)$ (i.e. QoD is degraded). Similarly, if $\Delta U(k)$ is greater than zero, then $MDE(k+1)$ is set such that $MDE(k+1)$ is less than $MDE(k)$ (i.e. QoD is upgraded). A formal definition of the concepts around f is given in section 4.7.

The algorithms presented in this work are designed to adjust a set of performance metrics such that a given QoS specification can be satisfied. In this work we use M^O to indicate user transaction quality. Similarly, data quality is expressed in terms of MDE. Consider the following "snapshot" of the RTDB state: M^O just below M_r^O and MDE near zero. Here, the transaction quality is quite low while the data quality is high. What we rather would like to observe is that the transaction and the data quality increase and decrease together. Hence, as M^O is increases, so does MDE and vice versa. We consider this property important but not necessary, as our preliminary goal is to design algorithms that satisfy a given QoS specification.

4.6.1 FCS-IC-1

In FCS-IC-1 (Feedback Control Scheduling Imprecise Computation 1) the FC-UM policy is used to manage data and user transaction quality. The integrated signal, ΔU, from the miss percentage and utilization controllers is fed into the QoD manager, which adjusts the QoD based on the signal. The rationale behind using a FC-UM controller is that $\Delta U(k)$ is computed by considering both the miss percentage and the utilization. As described earlier, the utilization reference is increased as long as no miss percentage overshoot is observed. When the miss percentage is higher than the reference, the miss percentage controllers take over and as a result, the utilization reference is reduced exponentially. The utilization controller then tries to enforce the new utilization reference. This technique has two advantages. First, the utilization yielding the target miss percentage can be closely approximated. Second, the exponential reduction decreases the risk for a potential overshoot. The outline of the algorithm is given as follows.

The system monitors the deadline miss percentage and the CPU utilization. At each sampling period, the CPU utilization adjustment, $\Delta U(k)$, is derived. Based on $\Delta U(k)$ we perform one of the following. If $\Delta U(k)$ is greater than zero, upgrade QoD as much as $\Delta U(k)$ allows. However, when $\Delta U(k)$ is less than zero, degrade the data according to ΔU, but not beyond than the highest allowed MDE (i.e. $MDE \times M_p$). Degrading the data further would violate the upper limit of MDE, given by the QoS specification. In the case when $\Delta U(k)$ is less than zero and MDE equal to $MDE \times M_p$, no QoD adjustment can be issued and, hence, the system has to wait until some of the currently running transactions terminate. An outline of FCS-IC-1 is given in Algorithm 1.

Algorithm 1 FCS-IC-1

```
Monitor $M^M(k)$, $M^O(k)$ and $U(k)$
Compute $\Delta U(k)$
**if** ($\Delta U(k) > 0$ and $MDE(k) > 0$) **then**
  Upgrade QoD according to $MDE(k+1) := f(\Delta U(k), MDE(k))$
  Inform AC about the portion of $\Delta U(k)$ not accommodated by QoD upgrade
**else if** ($\Delta U(k) < 0$ and $MDE(k) < MDE_r \times M_p$) **then**
  Downgrade QoD according to $MDE(k+1) = f(\Delta U(k), MDE(k))$
  Inform AC about the portion of $\Delta U(k)$ not accommodated by QoD downgrade
**else if** ($\Delta U(k) < 0$ and $MDE(k) = MDE_r \times M_p$) **then**
  Reject any incoming transactions
**else**
  Inform the AC of $\Delta U(k)$
**end if**
```

The advantage of FCS-IC-1 is that a potential miss percentage overshoot is suppressed by the use of a utilization controller. However, FCS-IC-1 has some drawbacks, as will be discussed in the following. Imagine the case where the RTDB is overloaded and the miss percentages are kept around the references. Using FCS-IC-1, the utilization reference is reduced as soon as M^M or M^O overshoot, resulting in the utilization controller to take over during the next sampling periods (since E_Util will be larger than E_M and E_O). In the consecutive sampling periods, the utilization controller tries to reduce E_Util by setting $\Delta U(k)$ to a negative value. This means that as soon one of the miss percentages overshoot its reference, the utilization of the system and, hence, the miss percentages are reduced significantly. This behavior is repeated every time one of the miss percentages is higher than its reference, resulting in a cycle of miss percentages above and significantly below the references. Hence, the average miss percentages of mandatory and optional subtransactions will be less than the specified references. Further, the QoD will be unnecessarily degraded as the utilization controller tries to reduce the utilization (negative $\Delta U(k)$).

The above is the main motivation for developing an algorithm that controls the miss percentage in a way such that the average miss percentages are kept near their reference (as they should do). Further, in the discussions given earlier, we would like M^O and MDE to increase and decrease together. For the reasons given above, we have developed an algorithm, called *FCS-IC-2* (Feedback Control Scheduling Imprecise Computation 2), which is described in the next section.

4.6.2 FCS-IC-2

FCS-IC-2 also uses feedback control to control data and user transaction quality. It differs from FCS-IC-1 in the way that a FC-M controller is used rather than a FC-UM. Here the utilization controller is removed to keep the miss percentages at the specified references. This may on the other hand yield high miss percentage overshoots.

As described above, we would like M^O and MDE to increase and decrease together. The reader may have noticed that as long as E_M and E_O are positive (i.e. $M^M \leq M_r^M \wedge M^O \leq M_r^O$), the controller output ΔU will be positive.[4] This means that even if the miss percentages are just below the references, $\Delta U(k)$ is positive and, hence, due to the property of f, the QoD manager will upgrade the data. For this reason, in FCS-IC-2, the QoD manager is extended such that MDE is set not only by considering ΔU, but also according to the current transaction quality, given by M^O. When ΔU is less than zero (i.e. at miss percentage overshoot), MDE is set according to f. However, when ΔU is greater or equal to zero, MDE is set according to the moving average of M^O, defined as:

$$M_{MA}^O(k) = \alpha M^O(k) + (1-\alpha) M_{MA}^O(k-1) \qquad 0 \leq \alpha \leq 1.$$

The latter results in the data quality to vary with the transaction quality. When M_{MA}^O is relatively low, MDE is set to a low value relative to MDE_r. As M_{MA}^O increases, MDE is increased too but only to a maximum value of $MDE_r \times M_p$. A further increase will violate the QoS specification. An outline of FCS-IC-2 is given in Algorithm 2.

4.7 QoD Management

4.7.1 Effects of Varying MDE

The data quality is controlled by the QoD manager (as described in the above), which controls $MDE(k)$ depending on the system behavior. An update transaction is rejected if the error of the data object that is to be

[4]At transient oscillation of M^M and M^O, the controller output, ΔU, may temporally stay negative due to the integral operation.

Algorithm 2 FCS-IC-2

```
  Monitor $M^M(k)$ and $M^O(k)$
  Compute $\Delta U(k)$
  **if** $(\Delta U(k) \geq 0)$ **then**
    Adjust $MDE(k)$ according to
              $MDE(k+1) = \min(\frac{M^O_{MA}(k)}{M^O_r} MDE_r, MDE_r \times M_p)$
    **if** $(MDE(k-1) < MDE(k))$ **then**
      Add the utilization gained after QoD degrade to $\Delta U(k)$
    **else**
      Subtract the utilization lost after QoD upgrade from $\Delta U(k)$
    **end if**
    Inform AC of the new $\Delta U(k)$
  **else if** $(\Delta U(k) < 0$ and $MDE(k) < MDE_r \times M_p)$ **then**
    Downgrade QoD according to $MDE(k+1) = f(\Delta U(k), MDE(k))$
    Inform AC about the portion of $\Delta U(k)$ not accommodated by QoD
    downgrade
  **else if** $(\Delta U(k) < 0$ and $MDE(k) = MDE_r \times M_p)$ **then**
    Reject any incoming transactions
  **else**
    Inform the AC of $\Delta U(k)$
  **end if**
```

updated is less or equal to the current maximum data error, as described in section 4.2.

Rejecting an update transaction results in a decrease in CPU utilization. We define *gained utilization*, $GU(k)$, as the utilization gained due to the result of rejecting one or more update transactions during period k. $GU(k)$ is formally defined as,

$$GU(k) = \sum_i \frac{\#RU_i(k)}{\#AU_i(k)} \times EU_i$$

where $\#RU_i(k)$ is the number of rejected update transactions T_i, $\#AU_i(k)$ the number of arrived update transactions T_i, and EU_i is the estimated utilization of the update transactions T_i. Note that $\#AU_i(k)$ can be either monitored at run-time or computed in advance if the periodicity of the update transactions are constant.

An important issue is how to set $MDE(k+1)$ given a certain $\Delta U(k)$. Basically, we want to set $MDE(k+1)$ such that,

$$GU(k) - GU(k+1) = \Delta U(k) \tag{4.15}$$

the difference of gained utilization in period k and $k+1$ equals the change to the requested estimated utilization. This requires that we can predict $GU(k+1)$ induced by $MDE(k+1)$. Note, we can only estimate this factor since our problem is of probabilistic nature. We can in advance predict the arrival times of the update transactions, but it is impossible to know the update values. One possible solution would be to by means of statistics and probability derive estimates of $GU(k)$. If the density function of the update values for each stream (generating update transactions) is known, we can compute the probability of rejecting transactions (T_j) generated by $Stream_j$, during a period k. This is done by considering the probability of $|CV_i - V_j| \leq CV_i \times MDE(k)$ for T_j updating d_i. Knowing the probability, we can go further and estimate $GU(k)$. For example, if the probability of rejecting T_j is 0.25, then the estimated contribution to the gained utilization by T_j is 0.25 times EU_j. By computing the estimated gained utilization contributed by each update transaction, we can derive the total estimated gained utilization, i.e. $GU(k)$. However, the above mention solution would require a high computational overhead. Further, the distributions of the update values may not be available, which in turn introduces uncertainties in the computation.

According to the discussions above we need to predict $GU(k)$, given $MDE(k)$. For now, we skip the notion of time and let GU denote gained utilization, as the result of our discussions below do not depend on time. GU can be viewed as a function of MDE ($\#RU_i$ depends on MDE). One can observe the following properties of GU as MDE varies. If MDE is equal to zero, then GU is equal to zero as well. As MDE increases, GU increases monotonically. The latter is a result of that GU cannot decrease if MDE increases. There is an upper boundary for GU, given by GU_{max}, which is the utilization of all update transactions. Consequently, as MDE grows, GU will grow towards GU_{max} (i.e. $MDE \rightarrow +\infty \Rightarrow GU \rightarrow GU_{max}$).

4.7.2 Modeling and Online Tuning

In this section, we describe our approach to predicting GU, given an MDE. We present a model relating GU and MDE and also a scheme for how the model can be adapted online.

We introduce the notion of *predicted gained utilization* (PGU),

$$PGU = g(MDE)$$

where given an MDE, the corresponding GU can be predicted. We derive g based on system profiling; it can be derived by considering the average of GU for different MDE and then use numerical analysis to derive g (with the use of e.g. interpolation and/or spline). Further, since RTDBs are dynamic systems in a sense that the behavior of the system and environment are changing (e.g. the distributions of the update values may change), the parameters of g have to be profiled and adjusted on-line. Also note that on-line profiling also has the advantage of requiring less accurate parameters obtained from an off-line analysis.

We derive g by first profiling the system off-line and *linearizing* the relationship between GU and MDE. In other words, we consider PGU to be a linear function of MDE. Here, it is possible to linearize the relationship based on different MDE intervals. In our approach, we have linearized the relationship within the bounds given by $[0, MDE_r]$. Hence, we linearize considering the "working-interval" of MDE, i.e. the interval that MDE is allowed to vary within. We refer to the gradient of g as γ and, hence, we get,

$$PGU = g(MDE) = \gamma \times MDE.$$

We have measured average GU for different MDE and the result in shown in Figure 4.5. From the measurements we set γ to 2.27.

The inverse of g, i.e. g^{-1}, is given by

$$MDE = g^{-1}(PGU) = \frac{1}{\gamma}PGU = \mu \times PGU.$$

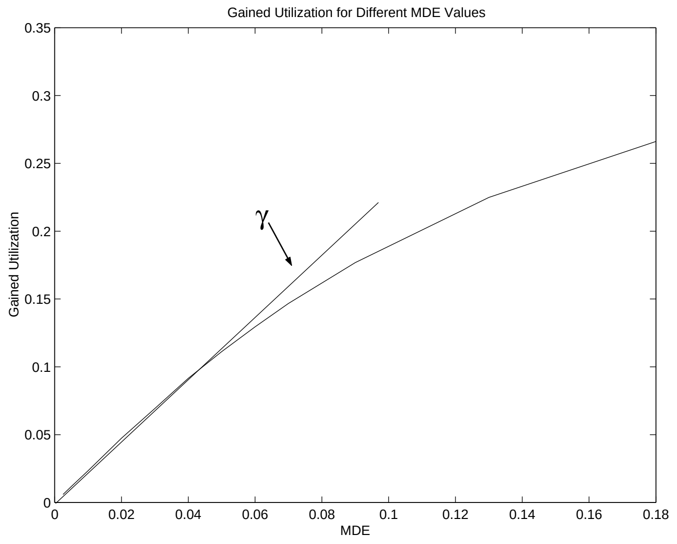

Figure 4.5: Profiling gained utilization with respect to different MDE

Given μ we can compute MDE based on PGU.

Further, an online profiling of μ is applied in order to adjust the model to changes. During each sampling period, $GU(k)$ is monitored and,

$$\mu(k) = \frac{MDE(k)}{GU(k)}$$

is computed. μ is then updated by considering the moving average of the earlier $\mu(k)$. The moving average of μ is defined by,

$$\mu_{MA}(k) = \alpha \times \mu(k) + (1 - \alpha) \times \mu_{MA}(k-1) \qquad 0 \leq \alpha \leq 1 \tag{4.16}$$

and is used to smoothen out large deviations from one sampling period to another. Updating μ according to equation (4.16) has the advantage of linearizing based on "locality". For example, consider the case where the system is at steady state and $MDE(k)$ is set to a certain value, v. Using equation (4.16), results in g being linearized at v. Hence, this technique adapts μ based on the current value of $MDE(k)$.

Finally, $MDE(k+1)$ can be computed by,

$$MDE(k+1) = \mu_{MA}(k) \times PGU(k+1) \tag{4.17}$$

4.7.3 Derivation of f

When f is used to compute $MDE(k+1)$ (as in FCM-IMP1 and some cases in FCS-IC-2), the following scheme is used.

Assume that we want to compute $MDE(k+1)$. It is desired that during period $k+1$, the gained utilization to equal the predicted gained utilization, i.e.,

$$GU(k+1) = PGU(k+1). \tag{4.18}$$

Note, in period k, $GU(k)$ does not necessary have to be equal to $PGU(k)$, as our predicted gained utilization may deviate from the actual gained utilization. From equation (4.15) and equation (4.18) we derive,

$$PGU(k+1) = GU(k) - \Delta U(k) \tag{4.19}$$

as the difference between the gained utilization and the estimated requested utilization adjustment. Inserting equation (4.19) in equation (4.17), yields:

$$MDE(k+1) = \mu_{MA}(k) \times (GU(k) - \Delta U(k)) \tag{4.20}$$

Since, $MDE(k+1)$ may not be greater than $MDE_r \times M_p$ (as given by a QoS specification), we take the minimum of the value generated by equation (4.20) and $MDE_r \times M_p$, and we get,

$$\begin{aligned} MDE(k+1) &= f(\Delta U(k))) = \\ &= \min(\mu_{MA}(k) \times (GU(k) - \Delta U(k)), MDE_r \times M_p). \end{aligned} \tag{4.21}$$

If $\Delta U(k)$ is less than zero, the result will be an increase in MDE, where as if $\Delta U(k)$ is greater than zero, the new MDE will be smaller. $\Delta U(k)$ equals zero results in no change in the current MDE.

Chapter 5

Performance Evaluation

In the following chapter a detailed description of the performed experiments is given. The goal and the background of the experiments are discussed, and finally the results are presented.

5.1 Experimental Goals

The main objective of the experiments is to show whether the presented algorithms can provide guarantees based on a QoS specification. We have for this reason studied and evaluated the behavior of the algorithms according to a set of performance metrics. The performance evaluation is undertaken by a set of simulation experiments, where a set of parameters have been varied. These are:

- **Load** (*Load*). Computational systems may show different behaviors for different loads, especially when the system is overloaded. For this reason, we measure the performance when applying different loads to the system.

- **Execution Time Estimation Error** (*EstErr*). Often exact execution time estimates of transactions are not known. To study how runtime error affects the algorithms, we measure the performance considering different execution time estimation errors.

- **QoS specifications**. It is important that an algorithm can manage different QoS specifications. Here we compare the results of the presented algorithms with regards to different QoS specifications.

5.2 Baselines

To the best of our knowledge, there has been no earlier work on techniques for managing data and transaction impreciseness, satisfying QoS and QoD requirements. Further, previous work within imprecise computing applied to tasks focus on maximizing or minimizing performance metrics (e.g. total error). The latter cannot be applied to our problem, since in our case we want to control a set of performance metrics such that they converge towards a set of references given by a QoS specification.

For this reason, we have developed two baselines that can manage data impreciseness based on miss percentage of optional subtransactions. We use the baselines to study the impact of the workload on the system. Here, we can establish the efficiency of FCS-IC-1 and FCS-IC-2 by comparing the operational envelope of the algorithms, i.e. we can compare the resistance to failure of the algorithms with regard to applied load and/or run-time estimation errors. The baselines are given below.

5.2.1 Baseline-1

In Baseline-1, the preciseness of the data is adjusted based on the relative miss percentage of optional subtransaction. Basically, MDE increases as M^O increases. No feedback control scheduling is used here as the adjustment of MDE is not based on a given requested utilization adjustment. MDE is set as follows:

$$MDE(k+1) = \min(\frac{M^O(k)}{M_r^O} MDE_r, MDE_r \times M_p) \tag{5.1}$$

A simple AC is applied, where a transaction (T_i) is admitted if the estimated utilization of admitted subtransactions and EET_i is less or equal to 80%.

5.2.2 Baseline-2

In Baseline-1, a significant change in MDE may introduce oscillations since a large $M^O(k)$ results in a large $MDE(k+1)$, which in turn lowers $M^O(k+1)$, resulting in a low $MDE(k+2)$ and so on. Baseline-2 is similar to Baseline-1, but here MDE is increased and decreased linearly. The outline of the algorithm is given below:

1. If $M^O(k)$ is greater than zero, increase $MDE(k)$ by a step (MDE_{step}) until $MDE_r \times M_p$ is reached (i.e. $MDE(k+1) = \min(MDE(k) + MDE_{step}, MDE_r \times M_p)$).

Parameter	Value
$\#DataObjects$	1000
P_i	$U:(100ms, 50s)$
EET_i	$U:(1ms, 8ms)$
AV_i	$U:(0, 100)$
AET_i	$N:(EET_i, \sqrt{EET_i})$
V_i	$N:(AV_i, AV_i \times VarFactor)$
$VarFactor$	$U:(0, 1)$

Table 5.1: Workload settings for data and update transactions

2. If $M^O(k)$ is equal to zero, decrease $MDE(k)$ by a step (MDE_{step}) until zero is reached (i.e. $MDE(k+1) = \max(MDE(k) - MDE_{step}, 0)$)

The same AC as in Baseline-1 is used here.

5.3 Simulation Setup

In our simulator, the workload consists of update and user transactions, which access data and perform virtual arithmetic/logical operations on the data. Update transactions occupy approximately 50% of the total workload. The workload model of the update and user transactions are given in Tables 5.1 and 5.2, and described as follows.

5.3.1 Data and Update Transactions

The simulated DB holds 1000 temporal data objects (O_i) where each data object is updated by a stream ($Stream_i$, $1 \leq i \leq 1000$). The period (P_i) is uniformly distributed in the range (100ms,50s) (i.e. $U:(100ms, 50s)$) and estimated execution time (EET_i) is given by $U:(1ms, 8ms)$. The average value (AV_i) is given by $U:(0, 100)$. Upon a periodic generation of an update, $Stream_i$ gives the update an actual execution time (AET_i) given by the normal distribution $N:(EET_i, \sqrt{EET_i})$ and a value (V_i) according to $N:(AV_i, AV_i \times VarFactor)$, where $VarFactor$ is uniformly distributed in (0,1). The deadline is set according to $D_i = AT_i + P_i$.

5.3.2 User Transactions

Each $Source_i$ generates a transaction T_i, consisting of one mandatory, and $\#O_i$ ($1 \leq \#O_i \leq 3$) optional subtransaction(s). The probability of $\#O_i =$

Parameter	**Value**
$\#O_i$	$1 \leq \#O_i \leq 3$, $P(\#O_i = 1) = \ldots = P(\#O_i = 3) = \frac{1}{3}$
$EET_i[t_i]$	$U : (10ms, 20ms)$
$AET_i[t_i]$	$AET_i[t_i] = (1 + EstErr) \times EET_i[t_i]$
Act Exec. Time	$N : (AET_i[t_i], \sqrt{AET_i[t_i]})$
D_i	$D_i = AT_i + EET_i \times SlackFactor$
$SlackFactor$	$U : (20, 40)$
Act. Data Accesses	$N : (\#DA_i[t_i], \sqrt{\#DA_i[t_i]})$

Table 5.2: Workload settings for user transactions

m, $1 \leq m \leq 3$, is $\frac{1}{3}$ (i.e. $P(\#O_i = 1) = P(\#O_i = 2) = P(\#O_i = 3) = \frac{1}{3}$).

The estimated execution time of the subtransactions ($EET_i[t_i]$) is given by U : $(10ms, 20ms)$. The estimation error $EstErr$ is used to introduce execution time estimation error in the average execution time given by $AET_i[t_i] = (1 + EstErr) \times EET_i[t_i]$. Further, upon generation of a transaction, $Source_i$ associates an actual execution time to each subtransaction t_i, which is given by $N : (AET_i[t_i], \sqrt{AET_i[t_i]})$. The deadline is set according to $D_i = AT_i + EET_i \times SlackFactor$. The slackfactor is uniformly distributed according to $U : (20, 40)$.

It is assumed that the number of data accesses ($\#DA_i[t_i]$) for each subtransaction (t_i) is proportional to $EET_i[t_i]$. Hence, longer subtransactions (or transactions) access more data. Upon a transaction generation, $Source_i$ associates an actual number of data accesses given by N : $(\#DA_i[t_i], \sqrt{\#DA_i[t_i]})$ to each subtransaction of T_i. The data set accessed by a transaction is partitioned among the subtransactions such that the partitions are mutually disjoint. However, the data sets accessed by transactions may overlap.

5.3.3 QoS Specifications

We consider the following QoS specifications for simulation purposes. They are referred to as *QoSSpec1* and *QoSSpec2* and are listed in Table 5.3.

Note, M_r^M, M_r^O, and MDE_r are steady-state specifications, whereas transient specifications for respective metric can be obtained by considering the overshoot M_p (e.g. $M^M \leq M_r^M \times M_p$). Note that by changing the target variables, we do not need to retune the controllers.

QoS Metric	**QoSSpec1**	**QoSSpec2**
M_r^M	1%	1 %
M_r^O	10%	5 %
MDE_r	2.5%	5 %
M_p	30%	30 %
T_s	60 s	60 s
U_r	$\leq$ 80%	$\leq$ 80%

Table 5.3: QoS specifications used for performance evaluation

5.4 Experiments and Results

The results of our experiments are given below. As described in section 5.1, we perform experiments based on varying load and varying execution time estimation error. Further, we compare the results between different QoS specifications.

In our experiments, one simulation run lasts for 10 minutes of simulated time. For all the performance data, we have taken the average of 10 simulation runs and derived 95% confidence interval, denoted as vertical lines in the graphs.

5.4.1 Varying Load

We first compare the performance of our approach with the baselines described in section 5.2 for increasing loads. The load is based on submitted user and update transactions. The tested approaches may reduce the applied load by applying admission control. The setup of the experiment is given below followed by the presentation of the results. Figure 5.1 shows the average M^O and MDE.

Experimental setup

We assume that a uniform access pattern is used. We measure M^M, M^O, MDE and U. The experiment setup is as follows:

- Apply loads from 50% to 200%.
- Set the execution time estimation error to zero (i.e. $EstErr = 0$).
- QoSSpec1 is used.

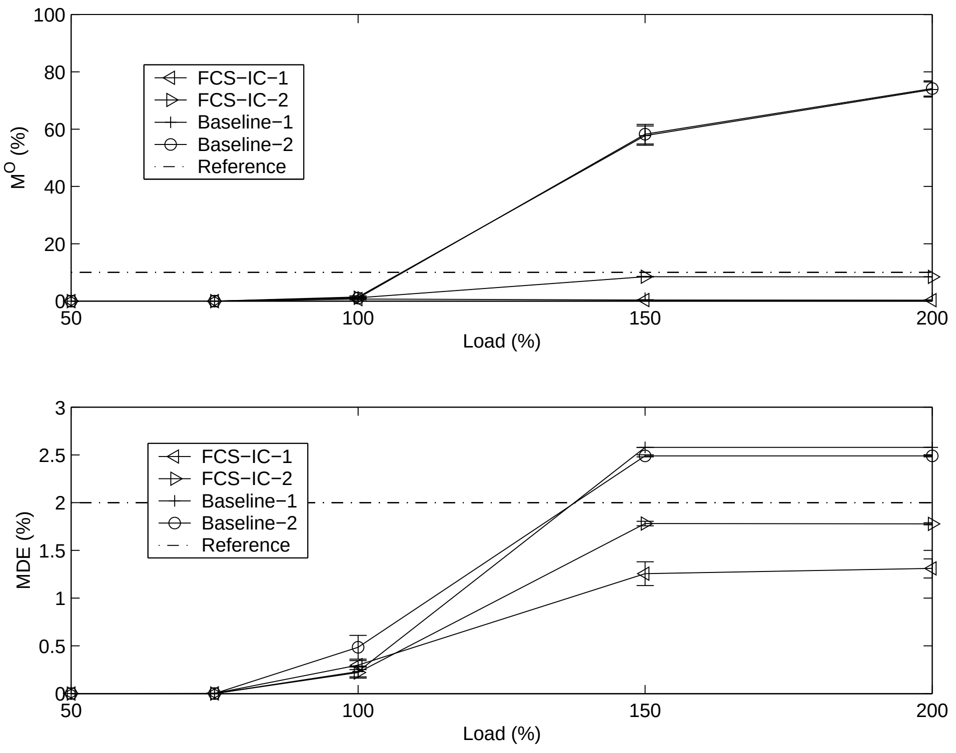

Figure 5.1: Average performance for $Load$ = 50, 75, 100, 150 and 200%, $EstErr = 0$, QoSSpec1

Average Miss Percentage of Mandatory Subtransactions

Miss percentage of mandatory subtransactions (M^M) has been observed to be zero for all four algorithms and, therefore, this has not been included in Figure 5.1. The specified M^M reference (M_r^M), has been set to 1% and this is not satisfied. The careful reader has observed that according to Figure 4.4 on page 36, M^M is increased when M^O reaches a large value. This is of course due to higher priority of mandatory subtransactions compared to optional subtransactions. Further, in section 4.5.6, we assumed that $\#O_i = 1$ for all transactions, i, which yields a lower U_{th}^M than in the assumed simulation setup for user transactions (i.e. $1 \leq \#O_i \leq 3$). Hence, for our simulation setup M^M will remain at zero even if M^O is quite large.

Average Miss Percentage of Optional Subtransactions

For Baseline-1 and Baseline-2, the miss percentage of optional subtransactions (M^O) increases as the load increases, violating the reference miss percentage, M_r^O, at loads exceeding 150%. Miss percentage for Baseline-1 and Baseline-2 at loads 150% and 200% are $57.7 \pm 3.4\%$ and $73.9 \pm 2.6\%$, respectively. In the case of FCS-IC-1, M^O is near zero at loads 150% and 200% ($0.35 \pm 0.1\%$ and $0.33 \pm 0.1\%$, respectively). Even though the miss percentage is low, it does not satisfy the QoS specification. This is in line with our earlier discussions regarding the behavior of FCS-IC-1. The low miss percentage is due to the utilization controller, since it attempts to reduce potential overshoots by reducing the utilization, which in turn decreases the miss percentage. FCS-IC-2 on the other hand shows a better performance. The average M^O at loads 150% and 200% is $8.5 \pm 0.1\%$, which is fairly close to M_r^O. In section 4.5, we tuned the controllers assuming that the utilization ratio is set to two (i.e. $G_A = 2$). In this experiment we have set $EstErr$ to zero (meaning that the utilization ratio is equal to one, i.e. $G_A = 1$), resulting in a certain model error. If $EstErr$ is set to one, i.e. $G_A = 2$, there is no model error and we can observe a M^O close to M_r^O. This is shown in section 5.4.2.

Average MDE

The average MDE for Baseline-1 and Baseline-2 violates the reference MDE set to 2%. This is due to the high M^O, and since in both algorithms MDE is set directly based on M^O, this yields a large average MDE (in both approaches, MDE is less than the transient performance specification given by $MDE < MDE_r \times M_p = 2.6\%$). In contrast, in the case of FCS-IC-1,

MDE is much lower than MDE_r. Since the miss percentages are kept low at all times, they are not likely to overshoot. Consequently, the control signal from the miss percentage controllers (i.e. ΔU_{MP}) is likely to be positive, which is interpreted by the QoD manager as an QoD upgrade and, hence, MDE will not reach the level of MDE_r. This will be further explained in section 5.4.3, where the transient performance of the algorithms is discussed. FCS-IC-2 provides an average MDE closer to MDE_r, given by 1.78±0.024% at loads exceeding 150%. Since in FCS-IC-2, MDE is set according to the M^O at steady state, MDE cannot reach MDE_r (since M^O does not reach the level of M_r^O).

Average Utilization

For all approaches, the utilization is above the specified of 80% for loads between 100-200%, reaching almost 100% at 200% applied load.

5.4.2 Varying EstErr

In this experiment, we compare the performance of our approach with the baselines described in section 5.2 for increasing execution time estimation error. The setup of the experiment is given below followed by the presentation of the results. Figure 5.2 shows the average M^O and MDE.

Experimental setup

We assume that a uniform access pattern is used. We measure M^M, M^O, MDE, and U. The experiment setup is as follows:

- Apply 200 % load.
- Set the execution time estimation error ranging from zero to one, increased by 0.25.
- QoSSpec1 is used.

Average Miss Percentage of Mandatory Subtransactions

As in section 5.4.1, M^M is zero for all approaches and all $EstErr$. The discussions in section 5.4.1 also apply here and are not further discussed.

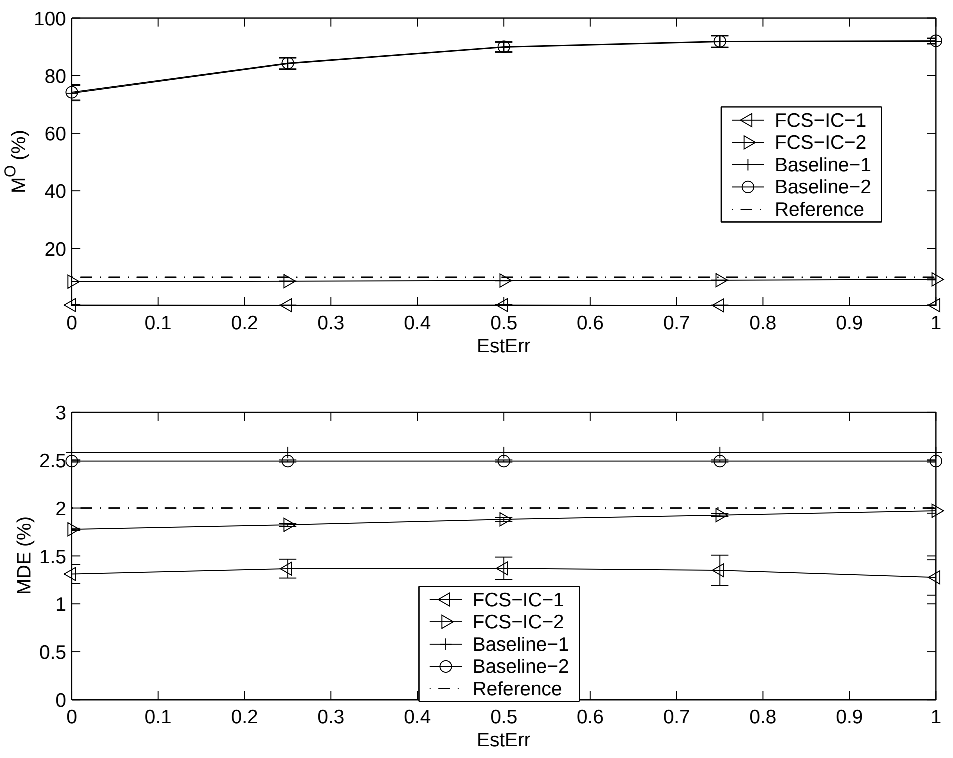

Figure 5.2: Average performance for $EstErr$ = 0, 0.25, 0.50, 0.75 and 1.0, $Load$ = 200%, QoSSpec1

Average Miss Percentage of Optional Subtransactions

As expected, Baseline-1 and Baseline-2 do not satisfy the QoS specification. In fact, as *EstErr* increases, M^O increases, reaching a value close to 90% for both algorithms. As we can see, FCS-IC-1 and FCS-IC-2 are more robust against varying *EstErr*. An interesting issue arises when considering the behavior of FCS-IC-1 as U and M^O actually decrease as *EstErr* increase (the utilization is not shown in Figure 5.2). As *EstErr* increases the more M^O is prone to overshoot. If this is the case, the utilization controller attempts to decrease the utilization more often, resulting in a lower average U and, hence, lower average M^O. FCS-IC-2 shows a somewhat different trend; M^O grows towards M_r^O as *EstErr* increases. M^O for *EstErr* set to zero is $8.47 \pm 0.036\%$, and for *EstErr* set to one is $9.23 \pm 0.17\%$. This is the result of the discussions given in section 5.4.1. As *EstErr* increases, the model error decreases and, hence, the controlled system becomes closer to the model.[1] This gives a more accurate picture of the system and the controllers are able to control the system in a more correct way.

Average MDE

Baseline-1 and Baseline-2 violate the specification $MDE_r = 2\%$. For FCS-IC-1, average MDE does not change considerably for different *EstErr*. In the case of FCS-IC-2, average MDE grows towards MDE_r, with increasing *EstErr*. The adjustment of MDE depends on M^O. As we noticed above, as *EstErr* increases, the average M^O grows toward M_r^O, resulting in MDE growing towards MDE_r, reaching a value of $1.97 \pm 0.03\%$ when *EstErr* is set to one.

5.4.3 Transient Performance

Studying the average performance is often not enough when dealing with dynamic systems. Therefore we study the transient performance of FCS-IC-1 and FCS-IC-2 when *Load* is set to 200%, *EstErr* is set to one, and QoSSpec1 is assumed. Figures 5.3 and 5.4 show the transient behavior of FCS-IC-1 and FCS-IC-2. The dash-dotted line indicates maximum overshoot.

Starting with FCS-IC-1, we note that M^O is kept low at all times. This is expected since the average M^O was shown to be low. Since both miss percentage and utilization controllers are used, one cannot explain the behavior of MDE considering the miss percentages only. The reader may have

[1]By model error we mean the deviation of the model compared with the actual system being controlled.

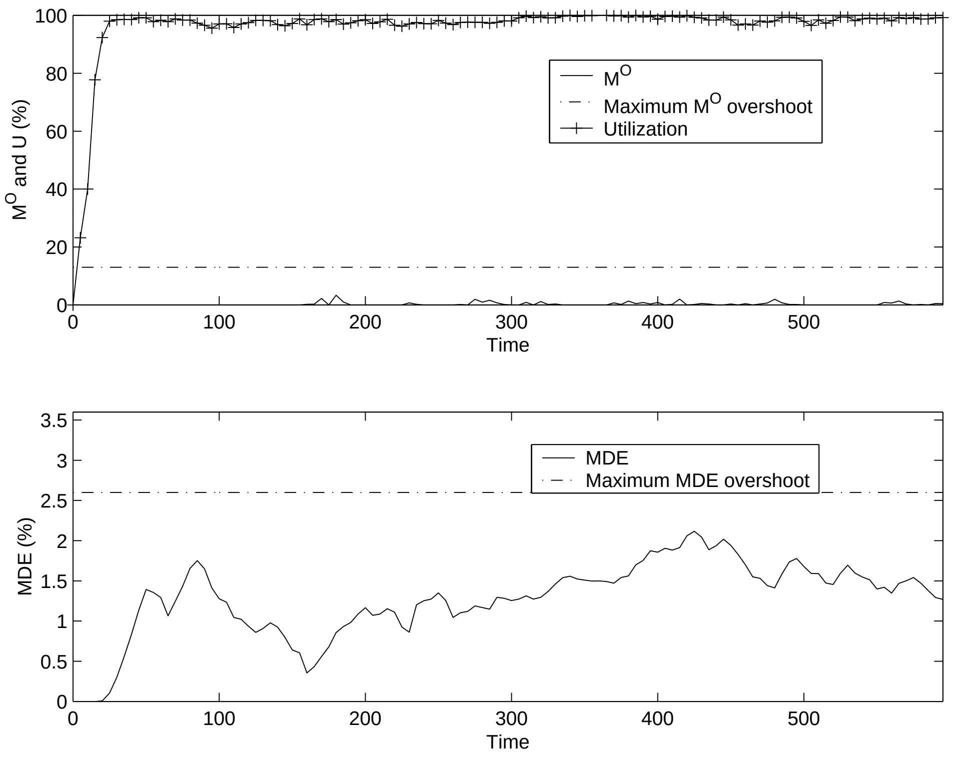

Figure 5.3: Transient performance for FCS-IC-1. $EstErr = 1.0$, $Load = 200\%$, QoSSpec1

noticed that MDE is greater than zero in the interval 20-150, where M^O is zero. Since MDE is greater than zero, it is clear that ΔU may become negative during that period. Further, since M^O is zero, the negative control signal must have originated from the utilization controller (we have traced a set of performance variables and they confirm this behavior). Initially, the utilization is below the reference (U_r). As the utilization increases and no miss percentage overshoots are observed, U_r increases linearly until a miss percentage is observed (one of the miss percentage controllers takes over) and U_r is reduced exponentially. In the current setting, U_r, is only increased if the utilization controller has taken over. Our investigations show that the utilization controller takes over once the utilization overshoots its reference, resulting in a negative ΔU and, hence, U_r being increased too late (one can observe that MDE starts increasing at time 20, which is exactly the time where the U overshoots U_r, initially set to 80%). Consequently, the negative ΔU leads to an increase in MDE.

In the case of FCS-IC-2, since no utilization controller is used, the behavior of MDE can be explained using M^O. FCS-IC-2 shows a more satisfying result as both M^O and MDE increase and decrease together (this was one of the desired effects discussed in section 4.6). Both M^O and MDE are kept around M^O_r and MDE_r, respectively. Although the average M^O is close to M^O_r, we can see that M^O often overshoots its reference. The highest M^O has been noted to 25.7%. This is much higher than the specified maximum miss percentage for optional subtransactions of 13% ($M^O \leq 13\%$). One cause to such overshoot is the various disturbances like data conflicts, resulting in restarts or abort of transactions. Further, we have assumed that $EstErr$ is equal to one, which yields a higher overshoot than in the case when $EstErr$ is zero. The results of setting $EstErr$ to zero is shown in Figure 5.5. Here we can see that the miss percentage variance is much smaller than in the case of $EstErr$ is equal to one. The maximum overshoot here has been noted to 20.6% at $t = 135$.

5.4.4 Varying QoS specification

We also compare the performance of our approaches considering other QoS specifications. The idea is to see how the algorithms can adapt to different QoS specifications. Below, we apply similar load pattern as in in the first experiment. Figure 5.6 shows the average M^O and MDE.

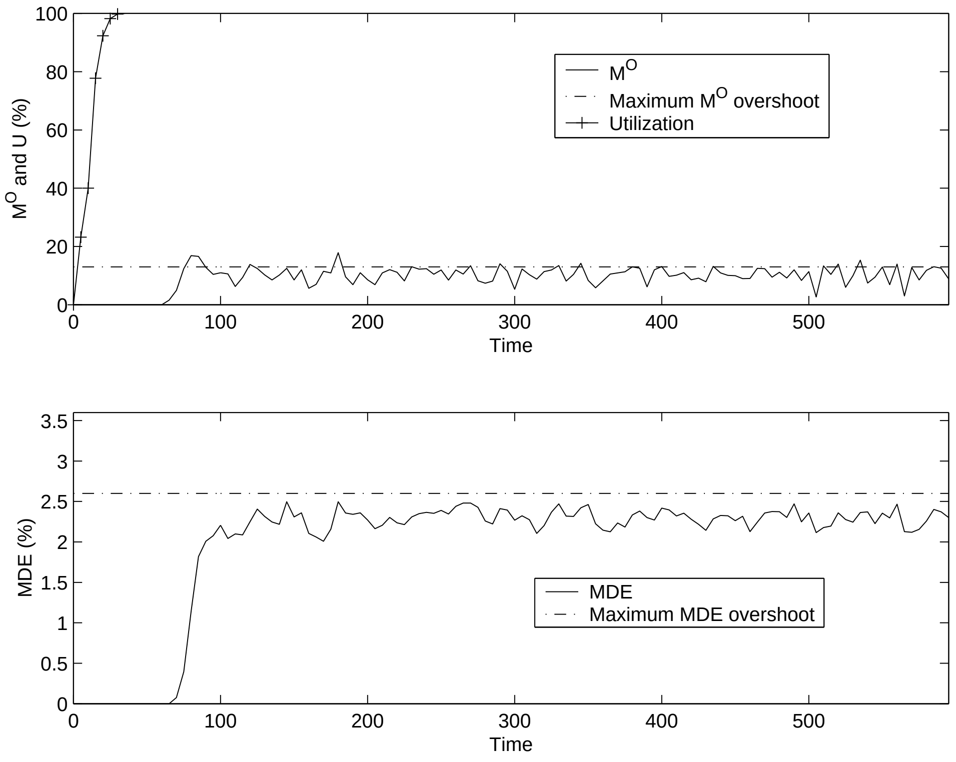

Figure 5.4: Transient performance for FCS-IC-2. $EstErr = 1.0$, $Load = 200\%$, QoSSpec1

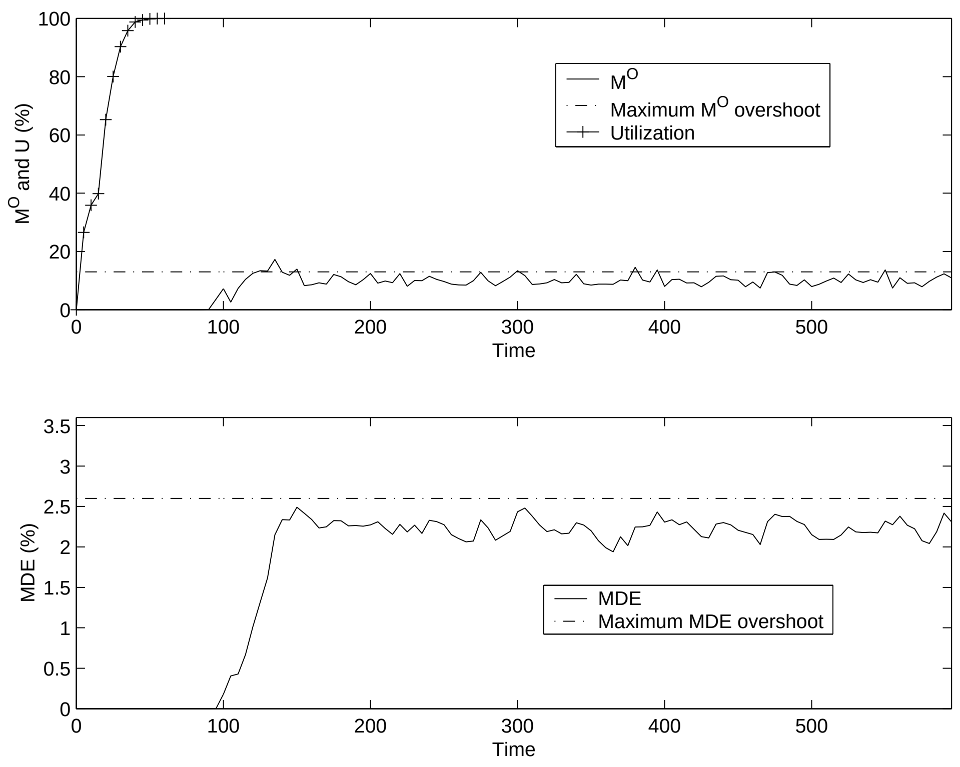

Figure 5.5: Transient performance for FCS-IC-2. $EstErr = 0.0$, $Load = 200\%$, QoSSpec1

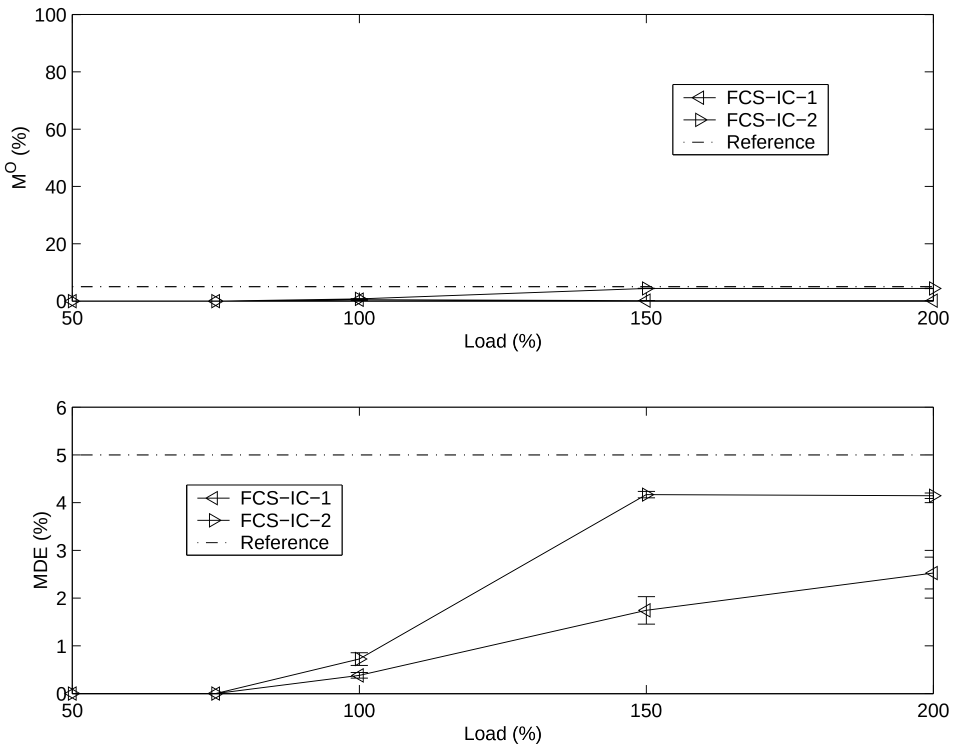

Figure 5.6: Average performance for $Load$ = 50, 75, 100, 150 and 200%, $EstErr = 0$, QoSSpec2

Experimental setup

Uniform access pattern is used. We measure M^M, M^O, MDE, and U. The experiment setup is as follows:

- Apply loads from 50% to 200%.
- Set the execution time estimation error to zero (i.e. $EstErr = 0$).
- QoSSpec2 is used.

Results

We do not present the results of the baselines, since they have showed poor results in earlier experiments. As shown in Figure 5.6, FCS-IC-1 is able to manage average miss percentage in a controlled manner. However, it does not comply with the given QoS specifications, i.e. average M^O is much lower than M_r^O and similarly MDE is lower than MDE_r. FCS-IC-2 on the other hand manages to provide better performance. Average M^O and MDE for FCS-IC-2 is $4.4 \pm 0.04\%$ and $4.14 \pm 0.06\%$, respectively.

5.4.5 Summary of Results and Discussions

It has been shown that FCS-IC-1 and FCS-IC-2 are robust against load variations and inaccurate execution time estimations.

FCS-IC-1 can manage to provide near zero miss percentage for optional subtransactions. We have also seen that FCS-IC-1 can efficiently suppress miss percentage overshoots. However, the average performance of FCS-IC-1, does not fully comply with a given QoS specification. Miss percentages and MDE are kept significantly lower than the references, violating the QoS specification. This is due to the exponential decrease in utilization every time M^O overshoots its reference. Although this policy is an efficient way of reducing number of potential overshoots, it often lowers the utilization unnecessarily, and as a result, data quality is degraded even though user transactions quality is high.

The average performance of FCS-IC-2 has shown to satisfy a given QoS specifications. Miss percentage of optional subtransactions, M^O, and MDE are consistent with the specified references. In addition, we have seen that the data and user transaction quality increase and decrease together. FCS-IC-2, however, produces overshoots above the maximum allowed overshoot, as given by the QoS specifications. Overshoots occur from various disturbances, such as restart and blocking of transactions, and cannot be suppressed when M^O is kept near the reference. This is the disadvantage of FCS-IC-2.

We conclude that FCS-IC-1 performs better than FCS-IC-2 for suppressing overshoots. FCS-IC-1 should be applied to RTDBs where overshoots cannot be tolerated, but where consistency between the controlled variables and their references is relaxed, i.e. we do not require the system to produce the desired miss percentages and MDE. The experiments show that FCS-IC-2 is particularly useful when consistency between the controlled variables and their references is emphasized, but some overshoots higher than the maximum allowed can be accepted.

Chapter 6

Related Work

We have identified three topics closely related to our work. Our research has been focusing on RTDBs where imprecise computation can be applied. We employ FCS in order to provide robustness against varying workload and inaccurate run-time estimates. Below, we present various works within FCS, QoS management in RTDBs and imprecise computation applied on databases.

6.1 Feedback Control Scheduling

In the past few years, feedback control scheduling has been receiving special attention [20, 19, 21, 8, 16, 6].

In [16], a framework is proposed for controlling the application requests for system resources using a PID controller. It has been shown that feedback control scheduling can be used to bound the resource usage in a stable and fair way.

Parekh et al. use feedback control scheduling to control the length of a queue of remote procedure calls (RPCs) arriving at a server [21]. In contrast to [20], they have used statistical models to tune their controllers. However, in their work, they have used I controllers (only integral part), whereas PI controllers (proportional integral parts), in general, have shown to provide better performance.

Changing the periodicity of a set of tasks in response to load variations has been suggested in [6]. If the estimated load is found to be greater than a threshold, task periods are enlarged to find the desired load. Aperiodic tasks are not considered in their model.

In the work of Cervin et al., a feedback-feedforward scheduling algorithm for optimizing the performance of a set of control tasks is presented [8]. The rates of the control tasks are adjusted, such that the utilization is kept close to a reference. Further, execution times of individual tasks are monitored and estimates are derived online. They also use a feedforward structure to make the feedback scheduler more reactive to workload changes.

The approaches above do not address imprecise computation. Lu et al. have presented a feedback control scheduling framework [20], where each task has several QoS-levels giving results of varying quality. Each QoS-level is characterized by a set of attributes, such as period, deadline and utilization. Miss percentage and utilization are monitored and controlled by changing the QoS-level for a set of tasks. They have modeled the system using an analytical method and tuned the model parameters via profiling. They have proposed three algorithms for managing the miss percentage and/or utilization. Simulation studies show that their algorithms provide performance guarantees for periodic and aperiodic tasks even when execution varies considerably from the estimate. Our work is based on the feedback control scheduling methodology given in [20].

6.2 QoS Management in RTDBs

Despite the abundance of QoS research, QoS-related work has been relatively scarce in database systems. The Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between freshness and transaction timeliness [3]. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions and their performance was compared.

The notion of QoD was introduced in [15]. In their work an update scheduling policy was proposed in the context of the web server. Here, the web pages are cached at the server and the back-end database continuously updates them. Their proposed update scheduling policy can significantly improve data freshness compared to FIFO scheduling.

In the work by Kang et al., a feedback control scheduling architecture is used to control the miss percentage and utilization by dynamically balancing update policies (immediate or on-demand) of a set of data [14]. In a similar work, specified miss percentage and utilization levels can be satisfied by modifying the period of the update transactions [13].

Although the approaches described above have addressed QoS or QoD, they have not considered QoS or QoD in terms of imprecision. In our work, we have defined QoS in terms of transactions and data impreciseness.

6.3 Imprecise Computation

Liu et al. proposed an imprecise computation model [18]. They presented a set of imprecise scheduling problems associated with imprecise computing and also gave an algorithm for minimizing the total error of a set of tasks. However, a task set may have more than one optimal schedule with the minimum total error. Consequently, for an optimal schedule the total error may be unevenly distributed among the tasks. Shih et al. addressed this problem by presenting two algorithms for minimizing the maximum error for a schedule that minimizes the total error [24]. Hansson et al. proposes an algorithm, OR-ULD, for minimizing total error and total weighted error (where there is an weight associated with each task) [10]. Further, it has been shown that OR-ULD can handle other types of workloads, including firm and hard tasks.

However, the algorithms presented by Liu, Shih, and Hansson require the knowledge of accurate processing times of the tasks. The same assumption cannot be made for transactions in a RTDB, as the execution time depends on the data needs of the users. In addition, overheads caused by data conflicts and concurrency control must be taken into account as well.

The reader may have noticed that the algorithms proposed by Liu, Shih, and Hansson focus on maximizing or minimizing a performance metric (e.g. total error). The latter cannot be applied to our problem, since in our case we want to control a set of performance metrics such that they converge towards a set of references given by a QoS specification.

Imprecise computation applied to RTDBs has been addressed in the literature. Here the correctness of answers to databases queries can be traded off to enhance timeliness. A query processor, APPROXIMATE [31], produces approximate answer if there is not enough time available. The accuracy of the improved answer increases monotonically as the computation time increases.

A relational database system proposed in [11], can produce approximate answers to queries within certain deadlines. Approximate answers are provided processing a segment of the database by sampling. Given a time quota, the sizes of the samples are computed and an estimation of the query is provided. The correctness of the estimation is improved as the number of samples increases. Further consistency can be traded off for shorter response time. In [22], queries can be executed in spite of concurrent updates to the data used by the transaction (which in the traditional sense violates consistency).

In the approaches above, impreciseness has been applied to only transactions. Further they have not addressed the notion of QoS. In our work, we have introduced impreciseness at data object level and presented QoS in terms of transactions and data impreciseness.

Chapter 7

Summary

Below a short summary is given. We discuss the suggested approaches, performance evaluations and then we identify topics/issues for future work.

7.1 Conclusion

The need for real-time data services has increased during the last years, e.g. sensor fusion support, web based applications and telecommunications. Here timely processing of user transactions using fresh data is important. As the run-time environment of such applications tends to be dynamic, it is imperative to handle transient overloads effectively. It has been shown that feedback control scheduling is quite adaptive to errors in run-time estimates (e.g. changes in workload and estimated execution time). Further, imprecise computation techniques have shown to be useful in many areas where timely processing of tasks or services is emphasized. In this work, we combine the advantages from feedback control scheduling and imprecise computation techniques, forming a framework where a database administrator can specify a set of requirements on the database performance and service quality. We introduce two algorithms, FCS-IC-1 and FCS-IC-2, for managing steady state and transient state performance in terms of data and transaction error. FCS-IC-1 and FCS-IC-2 give a robust and controlled behavior of RTDBs, in terms of transaction and data quality, even during transient overloads and when we have inaccurate run-time estimates of the transactions.

7.2 Future Work

We have identified some possible research problems that are listed below. They are based on feedback scheduling and different aspects and approaches to QoS management.

7.2.1 System Modeling and Feedback Control Scheduling

In this work, we have used an analytical approach to system modeling. Another way of modeling the controlled system is by means of statistical models. In the work by Parekh et al., a system model based on an ARMA model has been tuned by system profiling [21]. Since, RTDBs are quite complex, it is hard to derive accurate system models based on analytical methods. It is then more feasible to derive models based on statistical approaches. As the relative performance of the two approaches has not been established yet, it would be interesting to perform detailed comparisons between the two approaches.

Further, it is known that in real-time databases various factors such as non-uniform arrival pattern of transactions, varying execution time of transactions and unpredicted data conflicts (resulting in restarts or aborts) can introduce disturbances in the feedback control scheduling loop. The feedback control community has produced an impressive body of knowledge dealing with reduction of disturbances with various methods.

One possible way of rejecting disturbances is by describing them using stochastic models and designing the feedback control loop with regards to the "improved" model. Other ways include the structure of the control loop itself. By using various feedback and feedforward structures, the disturbances can be further eliminated. A third method mentioned in the literature is based on prediction of disturbances [29] with the use of e.g. Kalman filter.

7.2.2 Management of Derived Data

In this work, we have not addressed derived data management. A derived data object is derived from a set of base or derived data objects, R. A user transaction may miss its deadline if the derived data objects they read are out-of-date. If it is possible to update the derived data objects before they are read, temporal consistency is improved. However, derived data computation may be relatively expensive, including complex logical and/or arithmetic computations. Hence, it may not be acceptable to recompute a derived data object every time a data object in R is changed. Another option is to block transactions trying to access derived data objects that are out-of-date. However, if it takes too long for an update transaction to arrive, the waiting transactions may miss their deadline. To address the issues mentioned above, a QoS management scheme, similar to the following, need to be applied. Here, a derived data objects are kept updated when the system is underutilized. When the system is overloaded, the derived data objects are updated on-demand, e.g. with the use of triggered updates [5] or with the notion of forced delay [4] (i.e. delay the derived data recomputations during overloads).

Further, in this work we have applied imprecise data and transaction techniques to RTDBs. Hence, we allow a certain degree of data error in compensation for decreased resource requirements. This leads to the issue of error propagation in a database. Consider the case where a set of base data objects are read to compute a derived data object. If each of the base data objects have a certain error, then the derived data object will also have an error, induced by the error of the base data set. If the error of the base data is known and arithmetic operations are used to compute the derived data object, then the error of the derived data object can be computed. We may then be able to use the error propagation in our derived data management scheme. Here, recomputation of a derived data object might not be necessary if the base data error can be disregarded or can be tolerated. This approach requires that equations describing the influence of base data error on the derived data object is available. For most numerical computations, functions describing error propagation can be derived.

7.2.3 Service Differentiation

Service differentiation has been receiving more attention during the last years. In the case of RTDBs, one can classify transactions according to a set of classes with different QoS specifications. Classes can be issued depending on their criticalness, security needs etc.

A user transaction (T_A) belonging to a class (A), will in general produce service of higher quality (e.g. lower deadline miss percentage of user transactions) as compared to a user transaction (T_B) in a lower class (B). The QoS provided by user transactions is directly in connection with perceived freshness (i.e., the ratio of number of fresh data objects accessed by timely transactions to the total number of data objects accessed by timely transactions in a RTDB). It seems feasible that T_A should receive higher perceived freshness than T_B. If this is the case, the RTDB should be able to provide or satisfy the specified QoS for T_A to a greater extent. Since, the database freshness is in turn related to the update transactions, one can imagine classifying the update transactions into several classes.

To the best of our knowledge, there has been no work on classification on update transactions. Here an update transaction (T_j) updating a data object d_i can be classified into different classes depending on the class of user transactions accessing d_i. Conceptually, if d_i is frequently accessed by T_A, then T_j is moved into a higher class, resulting in T_j receiving higher priority among other update transactions. Hence, we will increase the perceived freshness for T_A. Now, an issue is how to classify update transactions. If T_A accesses d_i often (or other transactions belonging to class A), then clearly T_j should be moved to a higher class. The situation is somewhat different when d_i is rarely accessed by transactions in class A, but often by transactions in class B. The question here is if T_j should be moved to a higher class or not. It would be interesting to develop a scheme where update transactions can be classified according to the access patterns of the user transactions.

7.2.4 QoS Specification

A DBA may want to give a QoS specification in terms of various performance metrics. In this work, we have introduced metrics based on transaction and data quality. In [14], the DBA can specify the minimum perceived freshness that a database should have. A QoS could further include guarantees on response time. In the case of imprecise computation, if it is possible to describe transaction quality in terms of error, then one may specify transactions quality by steady-state and transient error.

In this work, we have not addressed maintainance of QoS *per* transaction basis. For example, in our work one can specify RTDB QoS in terms of miss percentage. Here, it is not possible to provide certain QoS guarantees for each admitted optional subtransaction. One transaction may have all its optional subtransactions executed and completed, while another one may only have one optional subtransactions completed before the transaction deadline. Hence, there will be a great variance in the QoS provided by the RTDB. For this reason, we would like to be able to specify a QoS, such that it is guaranteed for each admitted transaction. Also, it is desired that the variance of QoS provided by the RTDB is minimized.

7.2.5 Managing Data Impreciseness

As discussed in section 4.7, an issue regarding the data preciseness management was how much to degrade or upgrade QoD given an estimated requested utilization adjustment. Given the estimated utilization adjustment, $\Delta U(k)$, we would like to set $MDE(k+1)$ such that the difference in gained utilization in period k and $k+1$ is equal the estimated utilization adjustment, i.e., $GU(k) - GU(k+1) = \Delta U(k)$. Our approach to this problem was to profile the system and linearize the relation between $MDE(k)$ and gained utilization $GU(k)$ for different $MDE(k)$. From this we could derive estimates of $MDE(k+1)$, given $\Delta U(k)$.

Now, one can imagine other approaches to this problem. As discussed in section 4.7, we can apply an analytical method, based on probability and statistics. Given an $MDE(k+1)$, if it is possible to compute the probability of rejecting a set of update transactions, then the gained utilization for the specified $MDE(k+1)$ may be derived. It was mentioned that this method might require high computational overhead, which was one of the drawbacks. However, this method is still interesting from a theoretical point of view.

Bibliography

[1] *Control Systems Toolbox User's Guide.* The Mathworks Inc., 1996.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database System*, 17:513–560, 1992.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In ACM SIGMOD.

[4] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Extending Database Technology*, pages 223–240, 1996.

[5] Q. N. Ahmed and S. V. Vrbsky. Triggered updates for temporal consistency in real-time databases. *Journal of Real-time Systems*, 19(3):209–243, November 2000.

[6] G. Buttazzo and L. Abeni. Adaptive workload managment through elastic scheduling. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[7] G. C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[8] A. Cervin, J. Eker, B. Bernhardsson, and K. Årzén. Feedback-feedforward scheduling of control tasks. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[9] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems.* Addison-Wesley, third edition, 1994.

[10] J. Hansson, M. Thuresson, and Sang H. Son. Imprecise task scheduling and overload managment using or-uld. In *Proceedings of the 7th Conference in Real-Time Computing Systems and Applications*, pages 307–314. IEEE Computer Press, 2000.

[11] W. Hou, G. Ozsoyoglu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 68–77. ACM Press, 1989.

[12] J. Huang, J. A. Stankovic, D. F. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. pages 144–155. IEEE Real-Time Systems Symposium, 1989.

[13] K. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. Flute: A flexible real-time data management architecture for peformance guarantees.

[14] K. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A qos-sensitive approach for timeliness and freshness guarantees in real-time databases. 14th Euromicro Conference on Real-time Systems, June 19-21 2002.

[15] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *The VLDB Journal*, pages 391–400, 2001.

[16] B. Li and K. Nahrstedt. A control theoretical model for quality of service adaptations. Technical report, 1998.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[18] J. W.S. Liu, K. Lin, W. Shin, and A. Chuang shi Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5), May 1991.

[19] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptived real-time systems. IEEE Real-Time Systems Symposium, Orlando, FL, December 2000.

[20] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[21] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance managment. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[22] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the ACM SIGMOD International Conference of Managment of Data*, pages 377–386, 1991.

[23] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, (1), 1993.

[24] W. K. Shih and J. W. S. Liu. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Transactions on Computers*, 44(3):466–471, 1995.

[25] J. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. EuroMicro Conference on Real-Time Systems, June 1999.

[26] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[27] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling for Real-time Systems.* Kluwer Academic Publishers, 1998.

[28] K. J. Åström and T. Hägglund. *PID Controllers: Theory, Design and Tuning.* Instrument Society of America, second edition, 1995.

[29] K. J. Åström and T. Hägglund. *Computer Controlled Systems.* Prentice Hall, third edition, 1997.

[30] K. J. Åström and B. Wittenmark. *Adaptive Control.* Addion-Wesley, second edition, 1995.

[31] S. V. Vrbsky and W. S. Liu. Approximate-a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, December 1993.

Table of Abbreviations

ARMA	Autoregressive Moving Average
AC	Admission Controller
2PL-HP	2 Phase Locking with Highest Priority
BS	Basic Scheduler
CC	Concurrency Control
DBA	Database Administrator
FCS	Feedback Control Scheduling
FCS-IC-1	Feedback Control Scheduling Imprecise Computation 1
FCS-IC-2	Feedback Control Scheduling Imprecise Computation 2
FM	Freshness Manager
PI	Proportional Integral
QoD	Quality of Data
QoS	Quality of Service
RTDB	Real-Time Database

Table of Variables

Variable	Description	Page
AET_i	average execution time of transaction T_i	20
$AET_i[t_i]$	average execution time of subtransaction t_i	20
AIT_i	average inter-arrival time of T_i	21
$AIT_i[t_i]$	average inter-arrival time of subtransaction t_i	21
AT_i	arrival time of T_i	20
$AT_i[t_i]$	arrival time of subtransaction t_i	20
AU_i	average utilization of T_i	21
$AU_i[t_i]$	average utilization of subtransaction t_i	21
AV_i	average update value of T_i	20
AVI_i	absolute validity interval of data object d_i	22
CV_i	current value of data object d_i	22
D_i	relative deadline of T_i	20
$D_i[t_i]$	relative deadline of subtransaction t_i	20
DE_i	data error of d_i	22
EET_i	estimated execution time of transaction T_i	20
$EET_i[t_i]$	estimated execution time of subtransaction t_i	20
EIT_i	estimated inter-arrival time of T_i	21
$EIT_i[t_i]$	estimated inter-arrival time of subtransaction t_i	21
E_M	difference between M_r^M and M^M	32
E_O	difference between M_r^O and M^O	32
EU_i	estimated utilization of T_i	21
$EU_i[t_i]$	estimated utilization of subtransaction t_i	21

Variable	**Description**	**Page**
G_M^M	miss percentage factor of mandatory subtransactions	29
G_M^O	miss percentage factor of optional subtransactions	29
H_M	miss percentage closed loop transfer function	37
H_U	utilization closed loop transfer function	37
M^M	miss percentage of mandatory subtransactions	23
M^O	miss percentage of optional subtransactions	23
M_p	overshoot	24
MDE	maximum data error	23
P_i	period of T_i	20
$P_i[t_i]$	period of subtransaction t_i	20
P_M^M	transfer function of M^M	30
P_M^O	transfer function of M^O	30
P_U	utilization transfer function	30
T_s	settling time	24
TS_i	time stamp of data object d_i	22
U	utilization	24
U_{ActReq}	actual requested utilization	29
U_{EstReq}	estimated requested utilization	28
U_{th}^M	mandatory subtransaction schedulable threshold	29
U_{th}^O	optional subtransaction schedulable threshold	29
$\#DA_i[t_i]$	number of data accesses of subtransaction t_i	50
$\#O_i$	number of optional subtransactions in T_i	19

Index

2PL-HP, 23

absolute consistency, 4
absolute validity interval, 4, 22, 26
actual requested utilization, 28
actuator, 7
admission control, 22
arrival time, 20
average execution time, 20
average inter-arrival time, 21
average utilization, 21

base data, 4
baseline, 48
Baseline-1, 48
Baseline-2, 48
basic scheduler, 8, 26
BIBO, 11

closed loop, 7
concurrency control, 6, 23, 26
conditional integration, 34
controlled system, 7
controlled variable, 7
controlled variables, 24
controller, 7
    design, 10
    miss percentage, 32
    utilization, 32

data error, 22
deadline, 5, 20, 21
deadline miss percentage, 23
derived data, 4
dynamic model, 8

earliest deadline first, 6
error, 11, 22, 25
estimated execution time, 20
estimated inter arrival time, 21
estimated utilization, 21

FC-M, 31
FC-U, 31
FC-UM, 31
FCS-IC-1, 39
FCS-IC-2, 41
feedback control scheduling, 7
freshness manager, 26

gained utilization, 43

imprecise computation, 12
integrator antiwindup, 34

manipulated variable, 7, 25
maximum data error, 17
milestone approach, 13
miss percentage factor, 29
model
    controlled system, 8
    RTDB, 19
modeling, 10
monitor, 8
multiple version, 13

open loop, 7

overshoot, 9

period, 20
PI controller, 10
PI regulator, 30
precision control, 27
precision controller, 27
predicted gained utilization, 44
processing plant, 7
proportional gain, 11

QoD
  definition, 24
  manager, 25
QoS, 23

real-time database, 3
reference, 7
relative consistency, 4
relative consistency set, 4
relative validity interval, 4
robustness, 9
root locus, 10

saturation, 29
schedulable threshold, 29
scheduling, 6, 22
sensitivity, 9
settling time, 9
sieve function, 13
source, 25
specification, 8
stability, 9, 11
steady-state error, 9, 12
stream, 25
subtransaction, 19
system modeling, 27

temporal consistency, 4
temporal data, 3
time stamp, 4, 22, 27
time-domain specification, 9
transaction, 5, 19
transaction handler, 26
transfer function, 28
transient specification, 9

utilization ratio, 28

value, 4, 20, 22
value revision, 17