Report Number: WUCS-2006-39

2006-08-01

# Adaptive Quality of Service Control in Distributed Real-Time Embedded Systems

Xiaorui Wang

An increasing number of distributed real-time embedded systems face the critical challenge of providing Quality of Service (QoS) guarantees in open and unpredictable environments. For example, such systems often need to enforce CPU utilization bounds on multiple processors in order to avoid overload and meet end-to-end dead-lines, even when task execution times deviate significantly from their estimated values or change dynamically at run-time. This dissertation presents an adaptive QoS control framework which includes a set of control design methodologies to provide robust QoS assurance for systems at different scales. To demonstrate its effectiveness, we have applied the framework to... Read complete abstract on page 2.

# Adaptive Quality of Service Control in Distributed Real-Time Embedded Systems

Xiaorui Wang

Complete Abstract:

An increasing number of distributed real-time embedded systems face the critical challenge of providing Quality of Service (QoS) guarantees in open and unpredictable environments. For example, such systems often need to enforce CPU utilization bounds on multiple processors in order to avoid overload and meet end-to-end dead-lines, even when task execution times deviate significantly from their estimated values or change dynamically at run-time. This dissertation presents an adaptive QoS control framework which includes a set of control design methodologies to provide robust QoS assurance for systems at different scales. To demonstrate its effectiveness, we have applied the framework to the end-to-end CPU utilization control problem for a common class of distributed real-time embedded systems with end-to-end tasks. We formulate the utilization control problem as a constrained multi-input-multi-output control model. We then present a centralized control algorithm for small or medium size systems, and a decentralized control algorithm for large-scale systems. Both algorithms are designed systematically based on model predictive control theory to dynamically enforce desired utilizations. We also introduce novel task allocation algorithms to ensure that the system is controllable and feasible for utilization control. Furthermore, we integrate our control algorithms with fault-tolerance mechanisms as an effective way to develop robust middleware systems, which maintain both system reliability and real-time performance even when the system is in face of malicious external resource contentions and permanent processor failures. Both control analysis and extensive experiments demonstrate that our control algorithms and middleware systems can achieve robust utilization guarantees. The control framework has also been successfully applied to other distributed real-time applications such as end-to-end delay control in real-time image transmission. Our results show that adaptive QoS control middleware is a step towards self-managing, self-healing and self-tuning distributed computing platforms

2006-39

# Adaptive Quality of Service Control in Distributed Real-Time Embedded Systems, Doctoral Dissertation, August 2006

Authors: Xiaorui Wang

Corresponding Author: xiaoruiwang@gmail.com

Abstract: An increasing number of distributed real-time embedded systems face the critical challenge of providing Quality of Service (QoS) guarantees in open and unpredictable environments. For example, such systems often need to enforce CPU utilization bounds on multiple processors in order to avoid overload and meet end-to-end deadlines, even when task execution times deviate signi¯cantly from their estimated values or change dynamically at run-time.
This dissertation presents an adaptive QoS control framework which includes a set of control design methodologies to provide robust QoS assurance for systems at different scales. To demonstrate its e®ectiveness, we have applied the framework to the end-to-end CPU utilization control problem for a common class of distributed real-time embedded systems with end-to-end tasks. We formulate the utilization control problem as a constrained multi-input-multi-output control model. We then present a centralized control algorithm for small or medium size systems, and a decentralized control algorithm for large-scale systems. Both algorithms are designed systematically based on model predictive control theory to dynamically enforce desired utilizations.
We also introduce novel task allocation algorithms to ensure that the system is controllable and feasible for

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

# ADAPTIVE QUALITY OF SERVICE CONTROL IN DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

by

Xiaorui Wang

Prepared under the direction of Professor Chenyang Lu

---

A dissertation presented to the Henry Edwin Sever Graduate School of Washington University in partial fulfillment of the requirements for the degree of

DOCTOR OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

ABSTRACT

---

# ADAPTIVE QUALITY OF SERVICE CONTROL IN DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

by

Xiaorui Wang

---

ADVISOR: Professor Chenyang Lu

---

August 2006

Saint Louis, Missouri

---

An increasing number of distributed real-time embedded systems face the critical challenge of providing Quality of Service (QoS) guarantees in open and unpredictable environments. For example, such systems often need to enforce CPU utilization bounds on multiple processors in order to avoid overload and meet end-to-end deadlines, even when task execution times deviate significantly from their estimated values or change dynamically at run-time.

This dissertation presents an adaptive QoS control framework which includes a set of control design methodologies to provide robust QoS assurance for systems at different scales. To demonstrate its effectiveness, we have applied the framework to the end-to-end CPU utilization control problem for a common class of distributed real-time embedded systems with end-to-end tasks. We formulate the utilization control problem as a constrained multi-input-multi-output control model. We then present a centralized control algorithm for small or medium size systems, and a decentralized control algorithm for large-scale systems. Both algorithms are designed systematically based on model predictive control theory to dynamically enforce desired utilizations. We also introduce novel task allocation algorithms to ensure that the system is controllable and feasible for utilization control. Furthermore, we integrate our control algorithms with fault-tolerance mechanisms as an effective way to develop robust middleware systems, which maintain both system reliability and real-time performance even when the system is in face of malicious external resource contentions

and permanent processor failures. Both control analysis and extensive experiments demonstrate that our control algorithms and middleware systems can achieve robust utilization guarantees. The control framework has also been successfully applied to other distributed real-time applications such as end-to-end delay control in real-time image transmission. Our results show that adaptive QoS control middleware is a step towards self-managing, self-healing and self-tuning distributed computing platforms.

To my wife and my parents

# Contents

# List of Tables

# List of Figures

# Acknowledgments

First, I would like to thank my research advisor, Dr. Chenyang Lu for his guidance and detailed advice on this dissertation work. I would like to thank Dr. Christopher Gill and Dr. Xenofon Koutsoukos for their valuable suggestions and significant help. Most of this dissertation work is the result of my collaboration with them.

I would like to thank Mr. Malcolm Ware and Dr. Charles Lefurgy at IBM Austin Research Lab for being great mentors when I was a research intern there last summer. I also thank all members of the power-aware systems department. It was a pleasure working there with those great people.

My thanks once again go to Dr. Chenyang Lu, Dr. Christopher Gill, Mr. Malcolm Ware, Dr. Charles Lefurgy and Dr. Xenofon Koutsoukos for writing letters of recommendation for me. I really appreciated their help.

I am indebted to Dr. Kenneth Goldman and Dr. Shirley Dyke for serving on my dissertation committee, and for their insightful comments and feedback on this dissertation. I also thank Dr. Bijoy Ghosh for serving on my proposal committee.

I would like to acknowledge that parts of this work are the result of joint work with Dong Jia, Yingming Chen, Huang-Ming Huang and Venkita Subramonian. I thank them for interesting discussions and enjoyable collaborations.

Finally, my greatest thanks go to my wife for the tremendous support and constant encouragement she gave me during my doctoral studies. Without her support, it would be impossible for me to finish this dissertation work. I would also like to thank my parents for their patience and understanding during the past few years.

Xiaorui Wang

*Washington University in Saint Louis*
*August 2006*

# Chapter 1

# Introduction

## 1.1 Motivation

Classical real-time theory assumes that the characterization of workload and systems is known *a priori* in order to do schedulability analysis and provide performance guarantees in predictable environments (e.g., embedded process control and avionic applications). For example, classical scheduling algorithms such as Rate Monotonic (RM) [54] and Earliest Deadline First (EDF) [54][55] require complete knowledge of real-time tasks such as execution times, task periods, precedence constraints, and future arrival times. In real-time systems where accurate workload characteristics like execution times are not available, worst-case estimations are commonly used to guarantee desired real-time performance. However, this pessimistic solution often causes resource over-provisioning which may significantly increase the system cost because the computing resource (e.g. processors) may be severely underutilized. In addition, in many *soft* real-time systems like web servers, e-business applications, and audio and video processing, worst-case real-time analysis may not be applicable because those systems operate in open environments where both workload and available resources are difficult to predict. The increasing unpredictability is also due to several important trends in real-time systems, such as the increasing use of Commercial-Off-The-Shelf (COTS) components, the migration of real-time applications to plug-and-play open systems, and the proliferation of event-driven applications whose execution times are influenced heavily by input data. As such systems running in unpredictable environments become increasingly important to our society, system-wide adaptive solutions

are needed to meet their Quality of Service (QoS) requirements, such as real-time deadline, resource utilization and throughput.

### Adaptive Quality of Service Control

In recent years, a new paradigm of real-time computing based on *Adaptive QoS Control (AQC)* has received significant attention (e.g., [5] [19] [57] [89]). In contrast to traditional approaches to real-time systems that rely on accurate knowledge about system workload, AQC can provide robust QoS guarantees in unpredictable environments by adapting to workload variations based on dynamic feedback. Different from traditional adaptive solutions which rely on heuristics, a key advantage of AQC is that it adopts a control-theoretic framework for systematically developing adaptation strategies. The benefit of having control theory as a theoretic foundation is that we can have (i) standard approaches to choosing the right control parameters so that exhaustive iterations of tuning and testing are avoided; (ii) theoretically guaranteed control performance such as accuracy, stability, short settling time, small overshoot; and (iii) quantitative control analysis when the system is suffering unpredictable workload variations. This rigorous design methodology is in sharp contrast to heuristic-based adaptive solutions that rely on extensive empirical evaluation and manual tuning.

In this dissertation, we focus on an important instance of AQC called *utilization control* for soft real-time systems. The goal of utilization control is to enforce desired CPU utilization on a processor despite significant uncertainties in system workload. Utilization control is crucial to real-time systems because all tasks on a processor are guaranteed to meet their real-time deadlines if the utilization of the processor is equal to or lower than an appropriate schedulable bound [55]. For example, when the RMS scheduling algorithm is used on a processor, the schedulable bound can be calculated as a function of the total number of tasks on the processor [55]. As long as the real utilization of the processor is lower than the bound, it has been proved in real-time theory that all periodic tasks[1] on the processor can meet their

[1]Schedulable utilization bound also exists for systems with aperiodic tasks [3].

deadlines [54]. Utilization control provides us an effective way to guarantee all real-time deadlines without the detailed knowledge of the workload. It can also enhance system survivability by providing overload protection against workload fluctuation.

Several other projects have applied control theory to real-time systems. For example, Steere et al. and Goel et al. developed feedback-based schedulers [31] [89] that guarantee desired progress rates for real-time applications. Abeni et al. presented control analysis of a reservation-based feedback scheduler [5]. Lu et al. developed feedback control scheduling algorithms that control the CPU utilization and deadline miss ratio [57]. However, all these projects focused on controlling the QoS of *single*-processor systems. As a result, they are not applicable to a major category of real-time systems called distributed real-time systems that have end-to-end tasks running on multiple processors. This dissertation is different from those related projects because we focus on developing multi-input-multi-output (MIMO) control algorithms to control multiple processors simultaneously in distributed real-time systems.

### AQC in Distributed Real-Time Systems

Traditional approaches to handling end-to-end tasks such as end-to-end scheduling [91] and distributed priority ceiling [73] rely on schedulability analysis, which requires *a priori* knowledge about worst-case execution times. When task execution times are highly unpredictable, such open-loop approaches may severely underutilize the system. Recent years have seen rapid growth of *Distributed Real-time Embedded (DRE)* applications executing in *unpredictable* environments in which workloads are unknown and vary significantly at run-time. For example, task execution times in vision-based feedback control systems depend on the content of live camera images of changing environments [34]. Likewise, the supervisory control and data acquisition (SCADA) systems for power grid control may experience dramatic load increase during a cascade power failure [17]. Furthermore, as DRE systems become connected to the Internet, they are exposed to load disturbances due to variable user requests and even cyber attacks [17][13][38][101]. Hence, it is crucial to develop AQC algorithms for DRE systems.

In this dissertation, we focus on end-to-end utilization control on all processors to guarantee the end-to-end deadlines of all periodic real-time tasks. In real-time theory, a distributed real-time system is commonly abstracted as an *end-to-end task model* [55], where an end-to-end task may comprise of a chain of subtasks executing on multiple processors. The end-to-end deadline of each task is commonly divided into a set of subdeadlines for its subtasks. Then an appropriate schedulable utilization bound is enforced on each processor, so all (sub)tasks on the processor can meet their (sub)deadlines. As a consequence of end-to-end utilization control, the end-to-end deadlines of all tasks in the system can be guaranteed. Utilization control in such DRE systems introduces many new research challenges that have not been addressed in earlier work on single-processor systems.

1. Utilization control in DRE systems is a *multi-input-multi-output (MIMO)* control problem where the CPU utilization of all processors in the system must be guaranteed simultaneously. The multiple control inputs may be the invocation rates of all end-to-end tasks.

2. In DRE systems, the CPU utilization of each processor *cannot* be controlled independently from others. For example, changing the rate of an end-to-end task will affect the CPU utilizations of all the processors where its subtasks are located. Therefore, the *coupling* among processors must be modeled and addressed in the design of control algorithms.

3. Control model is subject to constraints. For example, the task rates usually can only be adapted within allowed ranges specified by application developers. Those constraints have to be systematically modeled in control algorithms to provide optimized online solution. Manual constraint maintenance may severely affect the control performance.

4. Different control algorithms are needed for DRE systems at different scales. A centralized controller may be more preferred for small or medium size DRE systems due to considerations in security and efficiency. However, large DRE systems (e.g. power grid management and smart spaces) usually require highly scalable control algorithms, since the communication and computation overhead of a centralized controller usually depends on the size of the *entire* system.

In this dissertation, we first present *EUCON* (End-to-end Utilization CONtrol) [59], the first control-theoretic utilization control algorithm designed for DRE systems with end-to-end tasks. EUCON can maintain desired CPU utilizations on multiple processors in a DRE system despite uncertainties in task execution times and coupling among processors. It employs a *centralized* MIMO model predictive controller to manage and coordinate the adaptation of multiple processors, subject to the constraints on task rates. While it is well suitable for small-scale DRE systems, this centralized control scheme has several limitations. Since its computation and communication overhead depends on the size of an *entire* DRE system, it is not scalable for large-scale systems (e.g. wide-area power grid management and ubiquitous smart spaces). Furthermore, the processor executing the controller is a single point of failure because the entire system will lose the capability of QoS adaptation if it fails.

To address the drawbacks of centralized control, we then present a more scalable control solution called *DEUCON* (Decentralized End-to-end Utilization CONtrol) [94] that can dynamically enforce desired utilizations on multiple processors in large-scale DRE systems. In contrast to centralized control schemes, DEUCON features a novel decentralized control structure that requires only localized coordination among neighbor processors. DEUCON is systematically designed based on recent advances in distributed model predictive control theory. Both control-theoretic analysis and simulations show that DEUCON can provide robust utilization guarantees and maintain global system stability despite severe variations in task execution times. Furthermore, DEUCON can effectively distribute the computation and communication cost to different processors and tolerate considerable communication delay between local controllers. Our results indicate that DEUCON can provide scalable and robust utilization control for large-scale distributed real-time systems executing in unpredictable environments.

While EUCON and DEUCON have shown promise, a fundamental problem of end-to-end utilization control is guaranteeing system *controllability* and *feasibility*. Both controllability and feasibility are important properties of DRE systems. No control algorithm (including EUCON, DEUCON or any other algorithms) can control a system if the system itself is uncontrollable. It may still be infeasible for a controllable system to achieve the desired utilization set points due to the task rate constraints. In this dissertation, we prove that controllability and feasibility depend crucially on the

end-to-end task configuration of a DRE system. We then present novel allocation algorithms for deploying end-to-end tasks to ensure that the system is controllable and robustly feasible. Furthermore, we develop runtime algorithms that maintain controllability and feasibility by reallocating subtasks dynamically in response to task termination and arrival. Our results demonstrate that our task allocation algorithms improve the robustness of utilization guarantees in DRE systems.

**Adaptive Real-Time Middleware**

While novel control algorithms have to be designed specifically for DRE systems, another important challenge is the implementation platform of the control algorithms. Modern DRE systems increasingly rely on middleware (e.g., Real-Time CORBA [67]) to meet QoS requirements on Commercial Off-The-Shelf (COTS) platforms. A key benefit of middleware is that it supports functional portability across different operating system platforms so that an application does not need to be reimplemented for different platforms. For QoS-critical applications, however, DRE middleware must support QoS portability [2][58] in addition to functional portability. A DRE middleware should allow applications to run on different platforms with the same critical QoS guarantees (e.g., CPU utilization) without the need for manual performance tuning. DRE middleware is an ideal platform to implement control algorithms because (i) adaptive middleware equipped with QoS control is an effective way to achieve both functional portability and QoS portability, (ii) DRE middleware operates at a distributed scope unlike stand-alone operating systems, so it is a particularly suitable layer for *end-to-end* control, and (iii) the integration of QoS control and established fault-tolerance mechanisms in DRE middleware provides double guarantees in terms of both system reliability and real-time QoS. Traditional fault-tolerant middleware exclusively focuses on reliability and failover strategies. In DRE systems, however, an end-to-end application that violates its real-time properties is equivalent to (or sometimes even worse than) an application that does not perform its computation. Therefore, it is extremely important to maintain real-time guarantees while recovering from failures.

In this dissertation, we first develope a single processor QoS control middleware called *FCS/nORB* as a starting point. FCS/nORB integrates a Feedback Control real-time

Scheduling (FCS) service with nORB, a small-footprint real-time Object Request Broker (ORB) designed for networked embedded systems [90]. FCS/nORB features feedback control loops that provide real-time performance guarantees by automatically adjusting the rate of remote method invocations transparently to an application. FCS/nORB thus enables real-time applications to be truly portable in terms of real-time performance as well as functionality, without the need for hand tuning. Chapter 6 presents the design, implementation, and evaluation of FCS/nORB. Our extensive experiments on a Linux testbed demonstrate that FCS/nORB can provide deadline miss ratio and utilization guarantees in face of changes in the platform and task execution times, while introducing only a small amount of overhead.

We then present *FC-ORB* (Feedback Controlled ORB) [96], a real-time Object Request Broker (ORB) middleware that employs end-to-end utilization control to handle fluctuations in application workload and system resources. The novelty of FC-ORB is the integration of end-to-end scheduling, adaptive QoS control, and fault-tolerance mechanisms that are optimized for unpredictable environments. FC-ORB implements a distributed utilization control loop that enforces desired CPU utilization bounds on multiple processors by adapting the rates of end-to-end tasks within user-specified ranges. The core of FC-ORB is a set of middleware-level mechanisms designed to support end-to-end tasks and distributed multi-processor utilization control in a real-time ORB. Extensive experimental results show that FC-ORB can maintain desired utilizations in face of uncertainties and variations in task execution times, resource contentions from external workloads, and permanent processor failure. FC-ORB demonstrates that the integration of utilization control, end-to-end scheduling and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

At the end of the dissertation, we introduce *CAMRIT* [93], a Control-based Adaptive Middleware framework for Real-time Image Transmission. Real-time image transmission is important to an emerging class of DRE systems operating in open network environments. Examples include avionics mission re-planning over Link-16 [20], security systems based on wireless camera networks, and online collaboration using camera phones. Meeting image transmission deadlines is a key challenge in such systems due to unpredictable network conditions. CAMRIT features a distributed feedback control loop that meets image transmission deadlines by dynamically adjusting the

quality of image tiles. We derive an analytic model that captures the dynamics of a distributed middleware architecture. A control theoretic methodology is applied to systematically design a control algorithm with analytic assurance of system stability and performance, despite uncertainties in network bandwidth. Experimental results demonstrate that CAMRIT can provide robust real-time guarantees for a representative application scenario.

## 1.2 Research Contributions

Specifically, this dissertation research makes the following major contributions.

- **Formulation of end-to-end utilization control** as a constrained least squares optimization problem. We derive a dynamic control model that captures the coupling among different processors and the constraints in DRE systems executing end-to-end tasks.
- **Design and analysis of an MPC controller**. We develop a Model Predictive Control (MPC) approach for the constrained MIMO control problem in DRE systems. We design and analyze a MIMO feedback control loop that provides robust utilization guarantees based on control theory when task execution times deviate from their estimation or vary significantly at run-time.
- **Design and analysis of a decentralized control algorithm**. We propose a new approach for decomposing the global MIMO utilization control problem into local subproblems to facilitate the design of decentralized control solutions. We design the DEUCON algorithm featuring a novel peer-to-peer control structure that enforces desired utilizations of multiple processors through localized coordination among controllers. We present control analysis based on the *distributed model predictive control* (DMPC) theory [16] which establishes the stability properties of the DEUCON algorithm in face of uncertain task execution times.
- **Evaluation and comparison of control algorithms**. We develop event-driven simulators to evaluate EUCON and DEUCON, respectively. Our results

indicate that both EUCON and DEUCON can provide robust utilization control for distributed real-time systems executing in unpredictable environments. While DECUON scales much better in large systems, it requires more complicated control analysis and has slightly worse control performance due to the lack of global information.

- **Architectural design of feedback control middleware**. We document the design of a utilization control service at the ORB middleware layer, which provides real-time performance portability and robust performance guarantees in face of workload variations. We implement a feedback control loop in a distributed ORB middleware that dynamically adjusts the rates of remote method invocations. We also address the design challenges of real-time middleware introduced by the integration of QoS control strategies such as continuous rate adaptation.

- **Design of end-to-end real-time ORB architecture**. We design an ORB architecture to support end-to-end real-time tasks based on the end-to-end scheduling framework [55]. We specialize the FC-ORB architecture to facilitate efficient end-to-end adaptation in memory-constrained DRE systems. We implement a distributed feedback control loop that provides the utilization control service and coordinates adaptations on multiple interdependent processors.

- **Integration of QoS control with fault tolerance mechanisms**. We integrate FC-ORB with fault tolerance mechanisms to handle processor failures with an adaptive strategy that combines reconfigurable utilization control and task migration. A unique feature of our fault tolerance approach is that it can maintain *real-time* properties for DRE applications after a processor failure.

- **Design of subtask allocation algorithms** to guarantee controllability and feasibility in end-to-end utilization control. We transform the controllability and feasibility problem to an end-to-end task allocation problem. We design allocation algorithms to preprocess the system workload before its deployment so we can ensure that the system is controllable and robustly feasible. We evaluate the algorithms with a large number of randomly generated workloads.

- **Investigation of controllability and feasibility** when workloads vary at runtime. We prove that dynamic task termination affects controllability while

dynamic task arrival affects feasibility. We maintain controllability and feasibility by reallocating subtasks dynamically at a small cost of runtime overhead. We implement and empirically evaluate the algorithms in the FC-ORB middleware system.

- **Design and analysis of a control middleware** for real-time image transmission. We derive an analytic model that captures the dynamics of a distributed middleware architecture. We systematically design a control algorithm with analytic assurance of system stability and performance, despite uncertainties in network bandwidth.

- **System developments and public release**. This dissertation research has produced three real-time middleware systems and two event-driven simulators. All the software is open-source and is publicly released at: `http://deuce.doc.wustl.edu/FCS_nORB/`. Based on the number of inquiry/question emails we received, the source code and executable files of those systems have been downloaded by many researchers at different universities and research institutes. The specific information of each system is as follows:

  - **FC-ORB:** implemented in 7017 lines of C++ code. The controller is implemented in 2089 lines of C++ code. Software is released at `http://deuce.doc.wustl.edu/FCS_nORB/FC-ORB/`.
  - **FCS/nORB:** implemented in 7898 lines of C++ code. Software is released at `http://deuce.doc.wustl.edu/FCS_nORB/FCS_nORB/`.
  - **CAMRIT:** implemented in 12835 lines of C++ code. Software is released at `http://deuce.doc.wustl.edu/FCS_nORB/CAMRIT/`.
  - **EUCON Simulator:** implemented in 2185 lines of C++ code. Software is released at `http://deuce.doc.wustl.edu/FCS_nORB/EUCON/`.
  - **DEUCON Simulator:** implemented in 2430 lines of C++ code. Software is released at `http://deuce.doc.wustl.edu/FCS_nORB/DEUCON/`.

## 1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 reviews the related work. Chapter 3 formulates the end-to-end utilization control problem. Chapter 4 and Chapter 5 present the design, analysis and evaluation of the centralized EUCON algorithm and the decentralized DEUCON algorithm, respectively. Chapter 6 introduces and empirically evaluates the FCS/nORB middleware system which provides QoS control service for single processor real-time system. Chapter 7 presents the architecture design, the control loop, the fault-tolerance mechanisms and the experimental results of the end-to-end FC-ORB middleware. Chapter 8 investigates the controllability and feasibility of end-to-end utilization control in DRE systems. Chapter 9 presents the application of feedback control to real-time image transmission. Chapter 10 concludes the dissertation and outlines future research directions.

# Chapter 2

# Related Work

In this chapter, we survey related work on end-to-end real-time scheduling, the applications of feedback control theory to different real-time computing systems, and adaptive real-time middleware.

## 2.1 QoS Control in Real-Time Systems

Traditional approaches for handling end-to-end tasks such as end-to-end scheduling [91] and distributed priority ceiling [73] rely on schedulability analysis, which requires *a priori* knowledge about worst-case execution times. When task execution times are highly unpredictable, such open-loop approaches may severely underutilize the system. An approach for dealing with unpredictable task execution times is resource reclaiming [14][85]. A drawback of existing resource reclaiming techniques is that they often require modifications to low-level scheduling mechanisms in operating systems. In contrast, the feedback control approach and rate adaptation techniques adopted in this dissertation can be easily implemented at the application or middleware layer on top of the Commercial Off-The-Shelf (COTS) platforms [58].

Several projects that applied control theory to real-time scheduling and utilization control are directly related to this dissertation. For example, Steere, et al., developed a feedback based CPU scheduler [89] that coordinates allocation of CPU cycles to consumer and supplier threads in a modified Linux kernel. Goel et al. developed feedback-based scheduler [31] that guarantees desired progress rates for real-time applications. Abeni et al. presented control analysis of a reservation-based feedback

scheduler [5]. Authors of [57] developed feedback control scheduling algorithms that controlled the CPU utilization and deadline miss ratio. These algorithms have been implemented as a middleware service [58]. For systems requiring discrete control adaptation strategies, hybrid control theory has been adopted to control state transitions among different system configurations [1][48].

Control theoretic approaches have also been applied to a number of other computing systems. For example, recently, control theory has been successfully used to do power or thermal control for processors [106][87][102] and computing servers [95][84]. Feedback control technique has also been applied to digital control applications [19] [84], networks [45][9][35], data service and storage system [61][44][7], and Internet servers [62][97][21]. A survey of feedback performance control in computing systems is presented in [2].

All the aforementioned projects focused on controlling the performance of *single*-processor systems. Their algorithms are based on single-input-single-output (SISO) linear control techniques which are not applicable to distributed systems with multiple processors. This dissertation is in sharp contrast to those related projects because we focus on developing multi-input-multi-output (MIMO) control algorithms to control multiple processors simultaneously in distributed real-time systems. Another key difference between the work presented in this dissertation and the related work is that we integrate our utilization control algorithms into ORB middleware systems, while the related work is based on either simulations or kernel implementations. ORB middleware is a particularly suitable layer for managing *end-to-end* adaptation in distributed systems since it operates at a broader (distributed) scope than stand-alone operating systems.

Two recent papers [88][53] proposed feedback control scheduling algorithms for distributed real-time systems with *independent* tasks. For example, Stankovic et al. proposed a distributed feedback control real-time scheduling algorithm designed for distributed systems [88]. These algorithms do not address the dependencies among processors caused by end-to-end tasks, which are commonly available in DRE systems. Instead, they assume that tasks on different processors are independent from each other, so they cannot handle the interrelationship between different controlled variables (i.e. processors). In contrast, our control algorithms are specially designed

to handle multiple processors that are coupled due to end-to-end tasks in DRE systems.

Diao et al. developed MIMO control algorithms to control the processor and memory utilizations for Apache web servers [21] and to do load balancing for data servers [22]. However, their algorithms were designed based on simpler linear control theory so they cannot handle actuation constraints. While those linear control algorithms may be sufficient for general data servers, attention has to be given in real-time systems because constraints are strictly enforced there. In this dissertation, we have innovatively developed the control algorithms based on the model predictive control (MPC) theory which can deal with constraints naturally. In addition, our decentralized control algorithm is the first designed to provide highly scalable control solution for large-scale DRE systems.

Another important distinction between our work and the aforementioned work is that our work is the first one which addresses the controllability and feasibility problem for DRE systems. Both controllability and feasibility are important properties of distributed systems where MIMO control is necessary. A recent paper [43] raised the problem of designing controllable systems. However, that paper focused only on some practical issues regarding how to get better control performance for SISO systems. In contrast, our work investigates the fundamental issues defined in control theory such as whether it is possible to control a DRE system and how to make an uncontrollable system controllable. Feasibility is another important issue. While the feasibility of scheduling tasks [8] has been addressed before in real-time community, in this dissertation, we focus on the feasibility of controlling DRE systems. We transform the controllability and feasibility problem to a task allocation problem in DRE system. Task allocation is a classical problem which has been discussed by several existing projects [36][27][6]. The difference between our work and those related projects is that we are trying to guarantee system controllability and maximize the probability of system remaining feasible, instead of minimizing communication cost or ensuring load balancing.

## 2.2 Real-Time Middleware

Adaptive middleware is emerging as a core building block for DRE systems. For example, TAO [81], dynamicTAO [47], ZEN [46], and nORB [90] are adaptive middleware frameworks that can (re)configure various properties of ORB middleware at design- and run-time. Higher-level adaptive resource management frameworks, such as QuO [107], Kokyu [29] and RT-ARM [41], leverage lower-level mechanisms provided by ORB middleware to (re)configure scheduling, dispatching, and other QoS mechanisms in higher-level middleware. ORB services such as the TAO Real-Time Event Service [33] and the TAO Scheduling Service [29] offer high-level services for managing reliability and real-time properties of interactions between application components.

Our middleware systems have several important features that distinguish them from the aforementioned earlier work on adaptive middleware. First, our work integrates the end-to-end scheduling service with a utilization control service. This integrated approach enables the middleware to meet end-to-end deadlines by dynamically controlling the utilizations on individual processors. Second, in contrast to earlier works that rely on heuristics-based adaptive techniques, our middleware service implements control algorithms that have been rigorously designed and analyzed based on a control-theoretic approach. Finally, our work enhances traditional fault-tolerance mechanisms with utilization control techniques to handle processor failures.

Agilos [52] was an earlier effort on control-based middleware framework for QoS adaptation in distributed multimedia applications. The work presented in this dissertation is different from Agilos in two important aspects. First, our work provides a general framework which is applicable to various real-time applications, whereas Agilos only supports adaptation strategies (e.g., image operations) specific to client-server multimedia applications (e.g., visual tracking). Second, our work employs advanced control-theoretic techniques to handle the complex couplings and constraints in large-scale DRE systems, whereas Agilos is based on control schemes such as linear control and fuzzy control, which cannot handle coupling or constraints in the controlled systems.

Another project that is closely related to our work is ControlWare [104], which is an incarnation of software performance control at the middleware layer. The difference is that ControlWare embodies adaptation mechanisms (such as server process allocation in the Apache server) that are tailored for Quality of Service provisioning on Internet servers, while our work integrates feedback control loop with remote method invocation mechanisms for distributed real-time embedded systems.

WSOA [20] gave a large-scale demonstration of adaptive resource management at multiple architectural levels in a realistic distributed avionics mission computing environment. The WSOA image transmission application is in essence a networked ad hoc control system, with adaptation of image tile compression to meet download deadlines. Based on the WSOA application, a real-time system computing model and theoretical controller has been developed in [93]. The work presented in this dissertation also seeks to add rigor to middleware-based resource management by applying control theory *within the middleware itself*. In doing so, we seek to complement other middleware projects for DRE systems, and increase the capabilities offered by DRE middleware as a whole.

Another difference between our work and the above three projects on control-based middleware is that our system is built upon the real-time Object Request Broker (ORB) middleware architecture which is a more general commercial-of-the-shelf (COTS) platform for DRE systems. In addition, our work provides an *end-to-end* utilization control service in a peer-to-peer architecture for DRE systems. A key feature of our work is that it can effectively coordinate the adaptation on multiple interdependent processors through a distributed feedback control loop.

# Chapter 3

# End-to-End Utilization Control

In this chapter, we formulate the end-to-end utilization control problem for DRE systems. The (CPU) utilization of a processor is the percentage of time when its CPU performs useful computation. Our utilization control strategy ensures that the utilizations of all processors in a DRE system remain below their set points specified by the user. Utilization control is important not only for preventing system crash due to CPU saturation but also for meeting end-to-end deadlines of distributed real-time tasks by enforcing an appropriate schedulable utilization bound on each host [55][59][94]. Adaptation is essential for a utilization control strategy to handle workload uncertainties and variations. A utilization control strategy may use different adaptation mechanisms as actuators to dynamically control the utilizations of processors. In this dissertation we focus on controlling the utilizations by dynamically adjusting task rates, i.e., the rates at which periodic tasks are released. Note that the algorithms developed in this dissertation can also be applied to other adaptation mechanisms.

## 3.1 Task Model

We adopt an end-to-end task model [55] implemented by many DRE applications. A system is comprised of $m$ periodic tasks $\{T_i|1 \leq i \leq m\}$ executing on $n$ processors $\{P_i|1 \leq i \leq n\}$. Task $T_i$ is composed of a chain of sub-tasks $\{T_{ij}|1 \leq j \leq n_i\}$ located on different processors. The release of subtasks is subject to precedence constraints, i.e., subtask $T_{ij}(1 < j \leq n_i)$ cannot be released for execution until

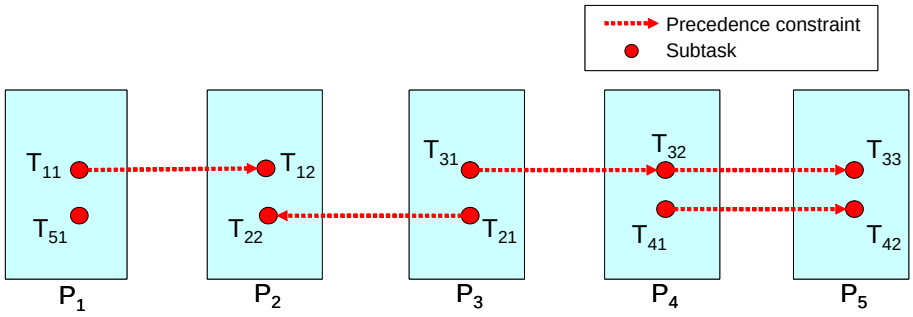



Figure 3.1: An example DRE application

its predecessor subtask $T_{ij-1}$ is completed. In a DRE middleware, the release of a subtask $T_{ij}(1 < j \leq n_i)$ is usually triggered by its predecessor $T_{ij-1}$ through a remote operation invocation or an event. If a non-greedy synchronization protocol (e.g., release guard [91]) is used to enforce the precedence constraints, all the subtasks of a periodic task share the same rate as the first subtask. Therefore, the rate of a task (and all its subtasks) can be adjusted by changing the rate of its first subtask. In this proposal, the processor $P_j$ hosting the first subtask of a task $T_i$ is called $T_i$'s *master processor* and we say $P_j$ *masters* $T_i$. Only a task's master processor can change its rate. An example DRE application with five end-to-end tasks running on five processors is shown in Figure 3.1.

Our task model has two important properties. First, while each subtask $T_{ij}$ has an *estimated* execution time $c_{ij}$ available at design time, its *actual* execution time may be different from its estimation and vary at run time. Modeling such uncertainty is important to DRE systems operating in unpredictable environments. Second, the rate of a task $T_i$ may be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. This assumption is based on the fact that the task rates in many applications (e.g., digital control [64][83], sensor update, and multimedia [10][12]) can be dynamically adjusted without causing system failure. A task running at a higher rate contributes a higher value to the application at the cost of higher utilizations.

We assume that each task $T_i$ has a *soft* end-to-end deadline related to its period. In an end-to-end scheduling approach [91], the deadline of an end-to-end task is divided into subdeadlines of its subtasks [42][66]. When the release guard protocol [91]) is used to synchronize the execution of subtasks, each subtask can be modeled as a periodic task. Hence the problem of meeting the deadline can be transformed to the problem

of meeting the subdeadline of each subtask. A well known approach for meeting the subdeadlines on a processor is to ensure its utilization remains below its schedulable utilization bound [50][54]. Therefore, the end-to-end scheduling approach provides a way to meet end-to-end deadlines by controlling the utilizations of all processors in the system.

## 3.2 Problem Formulation

Utilization control can be formulated as a dynamic constrained optimization problem. We first introduce several notations. $T_s$, the sampling period, is selected so that multiple instances of each task may be released during a sampling period. $u_i(k)$ is the CPU utilization of processor $P_i$ in the $k^{th}$ sampling period, i.e., the fraction of time that $P_i$ is not idle during time interval $[(k-1)T_s, kT_s)$. $B_i$ is the desired utilization set point on $P_i$. $r_j(k)$ is the invocation rate of task $T_j$ in the $(k+1)^{th}$ sampling period.

Given the utilization set point vector, $\mathbf{B} = [B_1 \dots B_n]^T$ and the rate constraints $[R_{min,j}, R_{max,j}]$ for each task $T_j$, the control goal at $k^{th}$ sampling point (time $kT_s$) is to dynamically choose task rates $\{r_j(k) | 1 \leq j \leq m\}$ to minimize the difference between $B_i$ and $u_i(k+1)$ for all processors:

$$\min_{\{r_j(k)|1\leq j\leq m\}} \sum_{i=1}^{n} (B_i - u_i(k+1))^2 \tag{3.1}$$

subject to constraints

$$u_i(k+1) \leq B_i \qquad (1 \leq i \leq n) \tag{3.2}$$

$$R_{min,j} \leq r_j(k) \leq R_{max,j} \qquad (1 \leq j \leq m) \tag{3.3}$$

The utilization constraints ensure that no processor exceeds its set point, and the rate constraints ensure all tasks remain within their acceptable rate ranges. The optimization formulation maximizes task rates by making the utilization of each processor as close to its set point as allowed by the constraints. The design goal is to ensure that all processors quickly converge to their utilization set points after a workload variation,

whenever it is feasible under the rate constraints. Therefore, to guarantee end-to-end deadlines, a user only needs to specify the set point of each processor to be a value below its schedulable utilization bound. In most real systems, the set point is usually configured to be the bound for maximum utilization, so hereinafter we use the words bound and set point interchangeably. Utilization control algorithms can be used to meet all the end-to-end deadlines by enforcing the set points of all the processors in a DRE system.

The control design faces three key challenges: (1) the utilization on multiple processors is coupled because changing the rate of one end-to-end task may affect the utilization of multiple processors. Therefore, a MIMO controller must be designed to control multiple processors simultaneously by adapting multiple task rates, (2) the control is subject to constraints including the upper bounds on utilizations and limits on acceptable task rates, and (3) the control algorithm must be able to handle unknown and varying task execution times.

## 3.3 Applications

End-to-end utilization control has several important applications over a broad range of QoS-critical systems.

**Meeting end-to-end deadlines:** Real-time tasks must meet their end-to-end deadlines in DRE systems. In the end-to-end scheduling approach [91], the deadline of an end-to-end task is divided into subdeadlines for its subtasks, and the problem of meeting the deadline is transformed to the problem of meeting the subdeadline of each subtask. A well known approach for meeting the subdeadlines on a processor is by enforcing the schedulable utilization bound [54]. The subdeadlines of all the subtasks on a processor are guaranteed if the utilization of the processor remains below its schedulable utilization bound. To guarantee end-to-end deadlines, a user only needs to specify the utilization set point of each processor to be a value below its schedulable utilization bound. This method can work with various subdeadline

assignment algorithms [42][66] and schedulable utilization bounds for different task models [50][54] presented in the literature.

**QoS portability:** End-to-end utilization control can also be deployed in a middleware to support QoS portability [60]. When an application is deployed on a faster platform, the task rates will be automatically increased to take advantage of the additional resource. On the other hand, when an application is deployed to a slower platform, task rates will be automatically reduced to maintain the same CPU utilization guarantees. This kind of self-tuning capability can significantly reduce the cost of porting DRE software across platforms.

**Overload protection:** Many distributed systems (including non-real-time systems) must avoid saturation of processors, which may cause system crash or severe service degradation [4]. On COTS operating systems that support real-time priorities, high utilization by real-time threads may cause kernel starvation [60]. End-to-end utilization control allows a user to enforce desired utilization bounds for all the processors in a distributed system. Moreover, the utilization set point can be changed online. For example, a user may lower the utilization set point on a particular processor in anticipation of additional workload, and the utilization controller will dynamically readjust task rates to enforce the new set point.

DRE systems span a wide spectrum in terms of scale and network support, so different control algorithms have to be designed for different systems. In this dissertation, we first present a centralized QoS control algorithm that is usually sufficient to many small-scale DRE systems (e.g., avionics systems, shipboard computing, and process control systems) running on server clusters, in which several processors connected through a high speed communication interface (e.g., a VME bus backplane). A decentralized control algorithm is then presented to provide scalable QoS guarantees for large-scale DRE systems (e.g., wide-area power grid management).

# Chapter 4

# EUCON: Centralized Control

As a step toward QoS control for the end-to-end task model, this chapter proposes the *End-to-end Utilization CONtrol (EUCON)* algorithm. EUCON can maintain desired CPU utilization in distributed systems with end-to-end tasks in unpredictable environments through online adaptation. The primary contributions of this chapter are four-fold:

- We derive a dynamic model that captures the coupling among processors and constraints in DRE systems executing end-to-end tasks.
- We develop a Model Predictive Control (MPC) approach for utilization control in DRE systems.
- We design and analyze a distributed multi-input-multi-output (MIMO) feedback control loop that provides robust utilization guarantees based on control theory when task execution times deviate from their estimation and vary significantly at run-time.
- We present extensive simulation results that demonstrate the effectiveness of EUCON and validate our control design and analysis.

In this chapter, we first give an overview of EUCON in Section 4.1. We then derive a dynamic system model for control design in Section 4.2. The detailed design and analysis are presented in Section 4.3. Section 4.4 evaluates EUCON with simulations. Section 4.5 summaries this chapter.

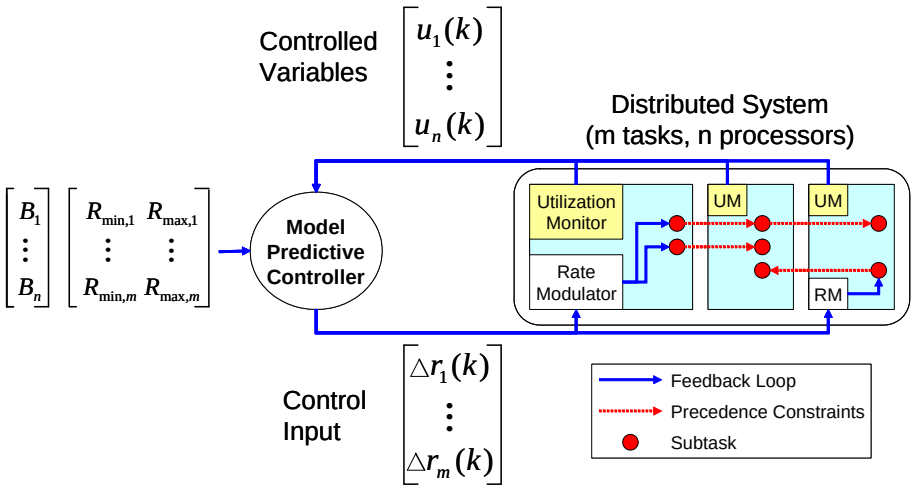



Figure 4.1: The MIMO feedback control loop in EUCON

## 4.1 EUCON Overview

As shown in Figure 4.1, EUCON features a MIMO feedback control loop composed of a centralized controller, a utilization monitor and a rate modulator on each processor. EUCON is invoked periodically, and its invocation period $T_s$ is selected so that multiple instances of each task may be released during a sampling period. The controlled variables are the utilizations of all processors, $\mathbf{u(k)} = [u_1(k)...u_n(k)]^T$. The control inputs from the controller are the changes in task rates $\mathbf{\Delta r(k)} = [\Delta r_1(k) \dots r_m(k)]^T$, where $\Delta r_i(k) = r_i(k) - r_i(k-1)(1 \leq i \leq m)$.

The feedback control loop works as follows:

1. The utilization monitor on each processor $P_i$ sends its utilization $u_i(k)$ in the last sampling period $[(k-1)T_s, kT_s)$ to the controller.

2. The controller collects the utilization vector $\mathbf{u(k)} = [u_1(k) \dots u_n(k)]^T$, computes a new rate change vector $\mathbf{\Delta r(k)} = [\Delta r_1(k) \dots r_m(k)]^T$, and sends the new task rates $\mathbf{r(k)} = \mathbf{r(k-1)} + \mathbf{\Delta r(k)}$ to the rate modulators on master processors (i.e., processors that master at least one task).

3. Then the rate modulators on master processors change the rates of tasks according to $\mathbf{r(k)}$.

## 4.2 Dynamic System Model

Following a control theoretic methodology, we must establish a dynamic model that characterizes the relationship between the control input $\mathbf{\Delta r}(k)$ and the controlled variable $\mathbf{u}(k)$. First, we model the utilization $u_i(k)$ of one processor $P_i$. Let $\Delta r_j(k)$ denote the change to task rate, $\Delta r_j(k) = r_j(k) - r_j(k-1)$. We define the *estimated change to utilization*, $\Delta b_i(k)$, as

$$\Delta b_i(k) = \sum_{T_{jl} \in S_i} c_{jl} \Delta r_j(k) \tag{4.1}$$

where $S_i$ represents the set of subtasks located at processor $P_i$. Note $\Delta b_i(k)$ is based on the estimated execution time. Since the actual execution times may be different from their estimation, we model the utilization $u_i(k)$ as the following difference equation.

$$u_i(k) = u_i(k-1) + g_i \Delta b_i(k-1) \tag{4.2}$$

where the utilization gain $g_i$ represents the ratio between the change to the actual utilization and its estimation $\Delta b_i(k-1)$. For example, $g_i = 2$ means that the actual change to utilization is twice of the estimated change. Note that the exact value of $g_i$ is *unknown* due to the unpredictability of subtasks' execution times. Equation (4.2) models a single processor. A system with $m$ processors is described by the following MIMO model.

$$\mathbf{u}(k) = \mathbf{u}(k-1) + \mathbf{G}\mathbf{\Delta b}(k-1) \tag{4.3}$$

where $\mathbf{\Delta b}(k)$ is a vector including the estimated change to utilization of each processor, and $\mathbf{G}$ is a diagonal matrix where $g_{ii} = g_i$ $(1 \leq i \leq n)$ and $g_{ij} = 0$ $(i \neq j)$. Note that $\mathbf{G}$ describes the effect of uncertainty in workload on the utilization of a DRE system. The relationship between the utilization and task rates is characterized as follows.

$$\mathbf{\Delta b}(k) = \mathbf{F}\mathbf{\Delta r}(k) \tag{4.4}$$

The *subtask allocation matrix*, $\mathbf{F}$, is an $n \times m$-order matrix, where $f_{ij} = c_{jl}$ if a subtask $T_{jl}$ of task $T_j$ is allocated to processor $P_i$, and $f_{ij} = 0$ if no subtask of task $T_j$ is allocated to processor $P_i$. $F$ captures the coupling among processors due to end-to-end tasks. Equations (4.3-4.4) give a dynamic model of a distributed system with $m$ tasks and $n$ processors.

**Example:** Consider a system with two processors and three tasks. $T_1$ has only one subtask $T_{11}$ on processor $P_1$. $T_2$ has two subtasks $T_{21}$ and $T_{22}$ on processors $P_1$ and $P_2$, respectively. $T_3$ has one subtask $T_{31}$ allocated to processors $P_2$. We have

$$u(k) = \begin{bmatrix} u_1(k) \\ u_2(k) \end{bmatrix}, G = \begin{bmatrix} g_1 & 0 \\ 0 & g_2 \end{bmatrix}, F = \begin{bmatrix} c_{11} & c_{21} & 0 \\ 0 & c_{22} & c_{31} \end{bmatrix}, \Delta r(k) = \begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \\ \Delta r_3(k) \end{bmatrix}$$

From (4.3), the system model is

$$u_1(k+1) = u_1(k) + g_1(c_{11}\Delta r_1(k) + c_{21}\Delta r_2(k))$$
$$u_2(k+1) = u_2(k) + g_2(c_{22}\Delta r_2(k) + c_{31}\Delta r_3(k))$$

## 4.3 Design and Analysis of A Model Predictive Controller

We present the design and analysis of a model predictive controller for EUCON. We first derive a mathematical formulation of EUCON in the model predictive control framework. Next this formulation is transformed to a constrained least-squares problem, which allows us to design the control algorithm based on an existing least squares solver. Finally, we prove the stability of our controller through control analysis.

### 4.3.1 Formulation for Model Predictive Control

Based on the system model, a MIMO predictive controller can be designed to guarantee the utilization set points on multiple processors. The single-input-single-output (SISO), linear control approach adopted in earlier works on feedback control real-time scheduling [57][88] is not suitable for DRE systems due to the coupling among multiple processors and the constraints. To solve this control problem, we adopt a *Model Predictive Control* (MPC) [63] approach. MPC is an advanced control technique used extensively in industrial process control. Its major advantage is that it can deal with coupled MIMO control problems with constraints on the plant and the actuators. This characteristic makes MPC very suitable for end-to-end utilization control in DRE systems where the performance measures and the coupling between processors can be expressed by constraints and MIMO system models.

The basic idea of MPC is to optimize an appropriate cost function defined over a time interval in the future. The controller employs a model of the system which is used to predict the behavior over P sampling periods called the prediction horizon. The control objective is to select an input trajectory that minimizes the cost while satisfying the constraints. An input trajectory includes the control inputs in the following $M$ sampling periods, e.g., $\mathbf{\Delta r(k)}$, $\mathbf{\Delta r(k+1|k)}$, ... $\mathbf{\Delta r(k+M-1|k)}$, where $M$ is called the control horizon. The notation $x(k+i|k)$ means that the vector signal $x$ depends on the conditions at time $k$. Once the input trajectory is computed, only the first element $\mathbf{\Delta r(k)}$ is applied as the input signal to the system. In the next step, the prediction horizon slides one sampling period and the input is computed again as a solution to a constrained optimization problem based on performance feedbacks $\mathbf{u(k)}$. MPC combines performance prediction, optimization, constraint satisfaction, and feedback control into a single algorithm. Details of MPC can be found in [63].

We now design a controller for EUCON. As illustrated in Figure 4.2, our model predictive controller includes a least squares solver, a cost function, a reference trajectory, and an approximate system model under the rate constraints. In the end of every sampling period, the controller computes the control input $\mathbf{\Delta r(k)}$ that minimizes the cost function under the rate constraints based on an approximate system model 4.7.

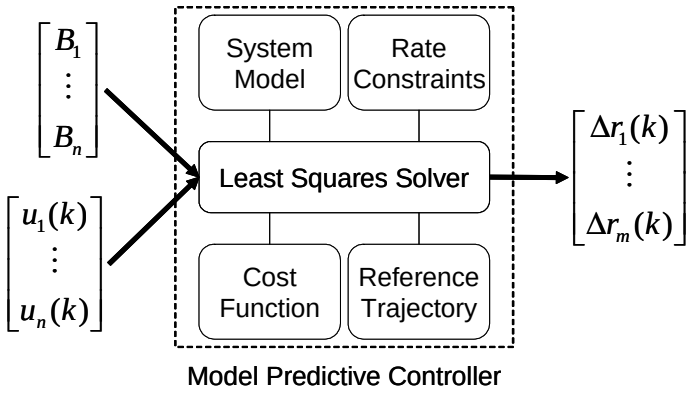



Figure 4.2: The model predictive controller

The cost function to be minimized by our controller is

$$V(k) = \sum_{i=1}^{P} \|\mathbf{u(k+i|k)} - \mathbf{ref(k+i|k)}\|^2_{\mathbf{Q(i)}} + \sum_{i=0}^{M-1} \|\mathbf{\Delta r(k+i|k)} - \mathbf{\Delta r(k+i-1|k)}\|^2_{\mathbf{R(i)}} \qquad (4.5)$$

where $P$ is the prediction horizon, $M$ is the control horizon, $\mathbf{Q(i)}$ is the *tracking error weight*, and $\mathbf{R(i)}$ is the *control penalty weight*. The first term in the cost function represents the *tracking error*, i.e., the difference between the utilization vector $\mathbf{u(k+i|k)}$ and a *reference trajectory* $\mathbf{ref(k+i|k)}$. The reference trajectory defines an ideal trajectory along which the utilization vector $\mathbf{u(k+i|k)}$ should change from the current utilizations $\mathbf{u(k)}$ to the utilization set points $\mathbf{B}$. Our controller is designed to track the following exponential reference trajectory so that the closed-loop system will behave as a linear system.

$$\mathbf{ref(k+i|k)} \;=\; \mathbf{B} - e^{-\frac{T_s}{T_{ref}} i}(\mathbf{B} - \mathbf{u(k)}) \qquad (1 \le i \le P) \qquad (4.6)$$

$T_{ref}$ is the time constant that specifies the speed of system response. A higher $T_{ref}$ causes the system to converge faster to the set points. By minimizing the tracking error, the closed-loop system will converge to the utilization set points if the system is stable. The weight matrix $\mathbf{Q(i)}$ can be tuned to represent preferences between processors. For example, we can assign a higher weight to a processor if it executes more important applications. The second term in the cost function represents the

*control penalty*. The control penalty term ensures that the controller will minimize the changes in the control input.

We have established a system model for DRE systems in Section 4.2. However, the model cannot be directly used by the controller because the system gains $\mathbf{G}$ are unknown. Therefore the controller must use an approximate model. Our controller assumes $\mathbf{G} = diag[1 \cdots 1]$ in (4.3), i.e., the controller assumes the actual utilization will be the same as the utilization predicted based on estimated ones. Hence our controller solves the constrained optimization based on an approximate system model as

$$\mathbf{u(k+1)} = \mathbf{u(k)} + \mathbf{F\Delta r(k)} \tag{4.7}$$

Although this approximate model may behave differently from the real system. However, as we prove in 4.3.2, the closed loop system under our controller can still maintain stability and guarantee desired utilization set points as long as $\mathbf{G}$ is within a certain range. Furthermore, this range can be established using stability analysis [60].

The controller must minimize the cost function (4.5) under the utilization and rate constraints (3.2) and (3.3) based on the approximate system model described by (4.4) and (4.7). This constrained optimization problem can be transformed to a standard constrained least-squares problem (the detailed transformation is available in Appendix A). The controller then uses a standard least-squares solver to solve the problem on-line.

In our system, we implement the controller based on the `lsqlin` solver in Matlab. `lsqlin` uses an active set method similar to that described in [30]. The worst-case computation complexity of the solver is polynomial in the numbers of tasks and processors in the system model (4.3). More specifically, our constrained least-square optimization is a convex nonlinear optimization, for which interior point methods require $O(n)$ Newton iterations [103], where $n$ is the number of optimization variables. Since each Newton iteration requires $O(n^3)$ algebraic operations, the worst-case computation complexity of the solver is cubic in the number of tasks and processors in the

system model. A preliminary overhead measurement in the MATLAB environment is presented in Section 4.4.6.

### 4.3.2 Stability Analysis

In MPC, a system is called *stable* iff for any initial condition it will converge to the equilibrium point [63]. In our case, the equilibrium points of the system are the utilization set points B. Hence a stable DRE system guarantees that the utilization of every processor converges to its set point. We now outline a general approach for analyzing the stability for a DRE system controlled by our controller.

1. Derive the control inputs $\mathbf{\Delta r}(k)$ that minimize the cost function based on the *approximate* system model described by (4.4) and (4.7).

2. Derive the closed-loop system model by substituting the derived control inputs $\mathbf{\Delta r}(k)$ into the actual system model described by Equations (4.3-4.4). The closed-loop system model is in the form

$$\mathbf{u}(k) = \mathbf{A}\mathbf{u}(k-1) + \mathbf{C} \tag{4.8}$$

   where $\mathbf{A}$ is a matrix whose eigenvalues depend on the utilization gains $\{g_i | 1 \leq i \leq n\}$.

3. Derive the stability condition of the closed-loop system described by (A.2). According to control theory, the closed-loop system is stable if all the eigenvalues of matrix A locate inside the unit circle in the complex space. Solving this stability condition will give the range of $g_i (1 \leq i \leq n)$ where the system will guarantee stability.

In our stability analysis, we assume the constrained optimization problem is *feasible*, i.e., there exists a set of task rates within their acceptable ranges that can make the utilization on every processor equal to its set point. If the problem is infeasible, no controller can guarantee the set point through rate adaptation. In Chapter 8, we address the issues of system feasibility and controllability by adjusting end-to-end task allocation.

**Example:** We now apply the stability analysis approach to the example system described in the end of Section 4.2. The system has 3 tasks and 2 processors. We set the prediction horizon $P = 2$ and the control horizon $M = 1$. According to the MPC theory, the system is also stable with any longer prediction horizon and control horizon if it is stable with shorter horizons. The time constant of the reference trajectory is $T_{ref}/T_s = 4$. The weights on all terms are 1. The cost function can be transformed to the following formula in scalar form

$$V(k) = \sum_{j=1}^{2}\sum_{i=1}^{2} \|\mathbf{u_j(k+i|k)} - \mathbf{ref_j(k+i|k)}\|^2 + \sum_{j=1}^{3} \|\mathbf{\Delta r_j(k)} - \mathbf{\Delta r(k-1)}\|^2 \quad (4.9)$$

Substituting the model parameters of the example system to (4.4) and (4.7), we have

$$\begin{bmatrix} u_1(k+1) \\ u_2(k+1) \\ u_1(k+2) \\ u_2(k+2) \end{bmatrix} = \begin{bmatrix} u_1(k) \\ u_2(k) \\ u_1(k+1) \\ u_2(k+1) \end{bmatrix} + \begin{bmatrix} c_{11} & c_{21} & 0 \\ 0 & c_{22} & 0 \\ c_{11} & c_{21} & 0 \\ 0 & c_{22} & c_{31} \end{bmatrix} \begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \\ \Delta r_3(k) \end{bmatrix} \quad (4.10)$$

Substitute (A.3) and the reference trajectory in (4.6) to (A.2), the cost function becomes a function of $\mathbf{\Delta r(k)}$. We then derive the control input vector $\mathbf{\Delta r(k)}$ that minimize the cost function (4.9) through partial differentiation. Following Step 2, we establish the closed-loop model by substituting $\mathbf{\Delta r(k)}$ derived in the last step into the actual system model (4.3-4.4). The closed-loop model is a function of the system gains $(g_1, g_2)$. Following Step 3, we can establish a stability region for $(g_1, g_2)$ in which the closed-loop system will remain stable. For example, in the special case when $g_1 = g_2$, the example system is guaranteed to be stable if $0 < g_1 = g_2 < 5.95$. That is, EUCON can maintain stability even if the execution time of every subtask becomes as high as 5.95 times its estimated one. The details of the stability analysis on this example are available at Appendix B. Note this approach is also applicable to more complex systems following the same steps.

### 4.3.3 Control Tuning

For a stable system, controller tuning involves a trade-off between utilization oscillation and the speed of convergence. Severe oscillation in utilization is undesirable even if the average utilization remains close to the set point. In practice, this may lead to oscillation in application performance such as video frame rate and the frequency of control in process control systems. The speed of converge is also important because it represents how quickly a system can recover from utilization variations and regain the desired utilization. If the gains used in the controller (1 in EUCON) is lower than the actual one ($g_i$), the real effect of the control input is going to be larger than what the controller has predicted and the system will oscillate. Using pessimistic estimation on execution times will reduce system oscillation because the system gains are less than 1 when execution times are overestimated. It should be noted that using pessimistic estimated execution times under EUCON does not cause underutilization. This key difference from open-loop scheduling is because EUCON dynamically adjusts rates based on measured utilization rather than the estimated execution times. However, more pessimistic estimation on execution times leads to smaller gains, which cause slower convergence to the set points.

The choice of the sampling period must balance convergence time, overhead, and oscillation. A short sampling period speeds up convergence by enabling the system to adapt to variations at a higher frequency. However, a short sampling period also increases the run-time overhead of EUCON because its feedback control loop is invoked once per sampling period. Moreover, since EUCON measures the average utilization over a sampling period, a longer sampling period may filter out noise in the utilization input to the controller and hence reduce oscillation.

## 4.4 Experimentation

### 4.4.1 Experimental Setup

Our simulation environment is composed of an event-driven simulator implemented in C++ and a controller implemented in MATLAB (R12). The simulator implements the distributed real-time system controlled by EUCON, the utilization monitor and the rate modulator. The sub-tasks on each processor are scheduled by the Rate Monotonic (RMS) scheduling algorithm [54]. The precedence constraints among subtasks are enforced by the release guard protocol [91]. The controller is based on the `lsqlin` least squares solver in MATLAB. The simulator opens a MATLAB process and initializes the controller at start time. In the end of each sampling period, the simulator collects the CPU utilization on each processor from the utilization monitors, and calls the controller in MATLAB with the utilization vector $\mathbf{u}(\mathbf{k})$ as parameters. The controller computes the control input, $\mathbf{\Delta r}(\mathbf{k})$, and return it to the simulator. The simulator then calls the rate modulator on each processor to adjust the task rates.

Each task's end-to-end deadline $d_i = n_i/r_i(k)$, where $n_i$ is the number of subtasks in task $T_i$. Each end-to-end deadline is evenly divided into subdeadlines for its subtasks. The resultant subdeadline of each subtask $T_{ij}$ equals its period, $1/r_i(k)$. Hence the schedulable utilization bound of RMS [54] is used as the utilization set point on each processor:

$$B_i = m_i(2^{1/m_i} - 1) \tag{4.11}$$

where $m_i$ is the number of subtasks on $P_i$. All (sub)tasks meet their (sub)deadlines if the utilization set point on every processor is enforced. As discussed in Section 3.3, other subdeadline assignment algorithms [42] and utilization bounds [50] may also be used with EUCON. Network delay is ignored in the simulations.

Two different workload/system configurations were used in our experiments. SIMPLE (see Table 4.1) is the example used in the stability analysis in Section 4.3.2. The second configuration, MEDIUM, simulates a more complex workload. MEDIUM

Table 4.1: Task parameters in SIMPLE (*Proc* represents the processor where a subtask is located)

| $\mathbf{T_{ij}}$ | **Proc** | $\mathbf{c_{ij}}$ | $\mathbf{1/R_{max,i}}$ | $\mathbf{1/R_{min,i}}$ | $\mathbf{1/r_i(0)}$ |
|---|---|---|---|---|---|
| $T_{11}$ | $P_1$ | 35 | 35 | 700 | 60 |
| $T_{21}$ | $P_1$ | 35 | 35 | 700 | 90 |
| $T_{22}$ | $P_2$ | 35 | | | |
| $T_{31}$ | $P_2$ | 45 | 45 | 900 | 100 |

Table 4.2: Controller parameters

| **System** | **P** | **M** | $\mathbf{T_{ref}/T_s}$ | $\mathbf{T_s}$ |
|---|---|---|---|---|
| SIMPLE | 2 | 1 | 4 | 1000 time unit |
| MEDIUM | 4 | 2 | | |

includes 12 tasks (with a total of 25 subtasks) executing on 4 processors. There are eight end-to-end tasks running on multiple processors and four local tasks (tasks $T_8$ to $T_{12}$). The execution time of every subtask $T_{ij}$ in MEDIUM follows a uniform distribution.

To evaluate the robustness of EUCON when execution times deviate from the estimation, the average execution time of each subtask $T_{ij}$ can be changed by tuning a parameter called the execution-time factor, $etf_{ij}(k) = a_{ij}(k)/c_{ij}$, where $a_{ij}$ is the average execution time of $T_{ij}$. The execution time factor represents how much the actual execution time of a subtask deviates from the estimated one. The execution-time factor (and hence the average execution times) may be kept constant or changed dynamically in a run. When all subtasks share a same constant execution time factor *etf*, etf equals to the system gain on every processor in the model, i.e., $etf = g_i(1 \leq i \leq m)$. The controller parameters are listed in Table 4.2. The controller for MEDIUM has higher control and prediction horizons to guarantee stability in a larger system.

### 4.4.2 Baselines

We compare EUCON against two baseline algorithms, OPEN and FC-U-E2E. OPEN is an open-loop algorithm that uses fixed task rates. It assigns task rates a priori

based on estimated execution times so that $\mathbf{B} = \mathbf{Fr}'$, where $\mathbf{F}$ is the subtask allocation matrix defined in Section 4.2, and $\mathbf{r}'$ is the vector of task rates assigned by OPEN. From the definition of $etf(k)$ we have

$$\mathbf{u}(k) = etf(k)\mathbf{B} \tag{4.12}$$

Although OPEN can result in desired utilization when estimated execution times are accurate (i.e., $etf(k) = 1$), it causes underutilization when execution times are overestimated (i.e., $etf(k) < 1$), and CPU over-utilization when execution times are underestimated (i.e., $etf(k) > 1$). Unfortunately, it is often difficult to establish tight bound on task execution times - especially in open and unpredictable environments where task execution times are heavily influenced by the value of sensor data or user input at run time.

FC-U-E2E is an extension of the FC-U [57] algorithm. Similar to EUCON, FC-U features a feedback control loop that controls utilization by dynamically adjusting task rates. However, FC-U is a single-processor algorithm, i.e., it only controls the utilization of a single processor. It uses a single-input-single-output (SISO) Proportional controller to compute the changes to task rates based on measured utilization. A simple approach for utilization control in a distributed system is executing a FC-U algorithm on each processor. Each FC-U algorithm controls the utilization of its own processor by computing task rates independently from others. However, this approach cannot handle the end-to-end task model due to its constraint that all the subtasks of an end-to-end task must execute at the same rate. In contrast, FC-U algorithms on those processors may decide to assign different rates to the same task based on the states of their own processors. For example, the FC-U controller on a heavily loaded processor may assign a lower rate to a task than that assigned by a lightly loaded processor that shares the same task. Therefore conflicts among the desired rates by multiple processors must be resolved. To guarantee the utilization bound constraints on all processors, a conservative approach can be adopted to assign the lowest rate given by any processors to a task. This mechanism can be implemented by adding a min component to the rate modulator on each processor. In the end of every sampling period, the rate modulator on each processor $P_i$ receives the rates assigned to each of its tasks from all the FC-U controllers on processors that share

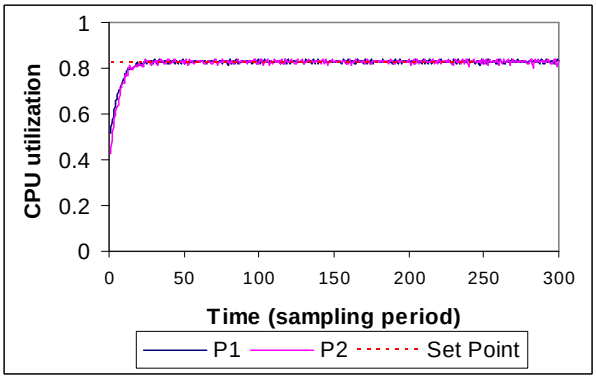
tasks with $P_i$, and change the rate of each of its task to the minimum one among all the received rates for this task. We refer to this extended algorithm FC-U-E2E. A fundamental difference between EUCON and FC-U-E2E is that EUCON explicitly incorporates the inter-processor coupling in a distributed system in its the design of a MIMO MPC, while FC-U-E2E implicitly handles the coupling by resolving the conflict among multiple SISO Proportional controllers through a min operator. As a baseline FC-U-E2E allows us to study the benefit of MPC compared to simple linear control.

In the following, we present three sets of simulations. In Experiment I, execution times are steady but deviate from the estimation. In Experiment II, task execution times vary dynamically at run-time. Experiment III compares EUCON with FC-U-E2E.

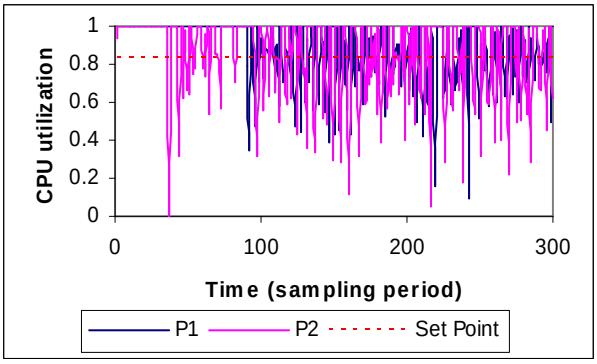
### 4.4.3 Experiment I: Steady Execution Times

In this set of experiments, all subtasks share a constant execution-time factor in each run. Since the system gains $g_1$ and $g_2$ equal the execution-time factor under this setup, we can compare the results of our stability analysis to the simulation results through these experiments. Figure 4.3(a) shows the system performance when the average execution time of every subtask is only half of the estimated one. In the beginning of the run, both processors are underutilized. EUCON then increases the task rates until the utilization of both processors converges to the utilization set points. As predicted by our control analysis, the system remains stable in this case. In contrast, Figure 4.3(b) shows the situation when the average execution time of every subtask is seven times its estimation. In the beginning, the processors were fully utilized because of the long task execution times. At around time $30T_s$, the utilization drops sharply to almost zero and starts to oscillate. The utilization on $P_2$ also oscillates significantly. The system fails to converge to the utilization set point. This result is also consistent with our stability analysis that predicts the system will be unstable when the system gains exceed 5.95.

We plot the mean and standard deviation of utilization on $P_1$ during each run in Figure 4.4(a). Every data point is based on the measured utilization $u(k)$ from time



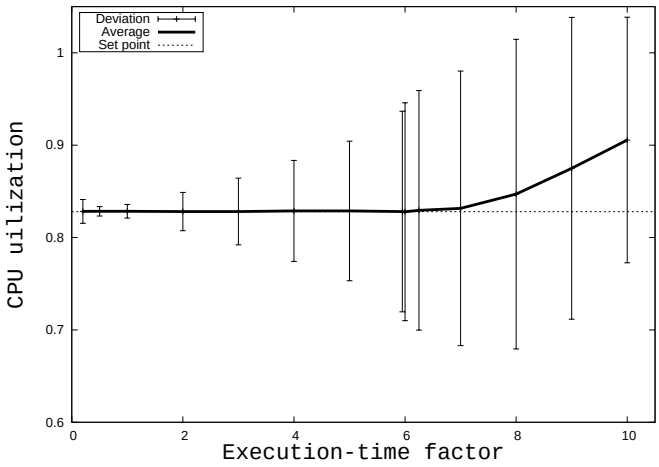

(a) Execution-time factor = 0.5



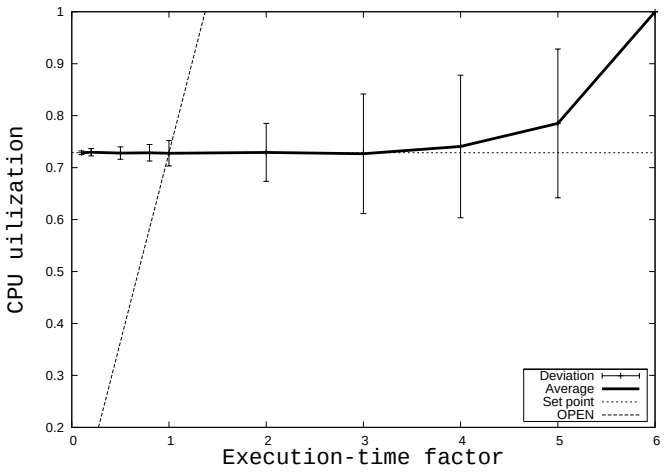

(b) Execution-time factor = 7

Figure 4.3: Utilization under different execution time factors (SIMPLE)

$100T_s$ to $300T_s$ to exclude the transient response in the beginning of each run. The system performance is considered acceptable if the average utilization is within 0.02 to the utilization set point, and the standard deviation is less than 0.05. Satisfying the requirement on average utilization ensures that the system achieves the desired utilization. Satisfying the requirement on standard deviation ensures that the utilization does not oscillate significantly. While the thresholds for acceptable performance depend on specific applications, the general conclusions drawn in this section are applicable to many applications. As shown in Figure 4.4(a), the average utilization remains close to the set point for execution-time factors between 0.20 and 5.95, and it starts deviating from the set point and increases linearly when the execution-time




(a) SIMPLE




(b) MEDIUM

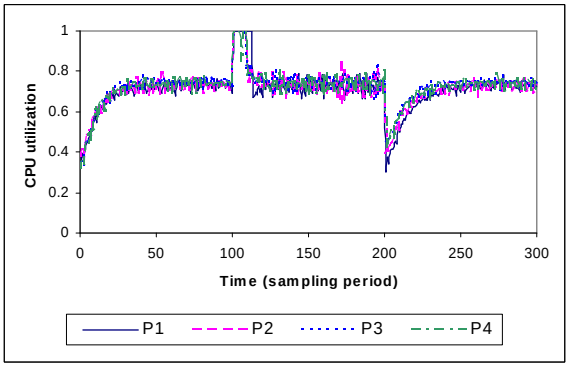
Figure 4.4: Average utilization on $P_1$

factor exceeds 6.00. When execution-time factor = 5.95, the average utilizations on $P_1$ and $P_2$ are 0.828 and 0.829, respectively. When execution-time factor increases to 6.00, however, the average utilization on $P_1$ and $P_2$ become 0.828 and 0.833, respectively. Based on the set point of 0.828 on both processors, the system becomes unstable (on $P_2$) when execution-time factor is in the range [5.95, 6.00] in the run.

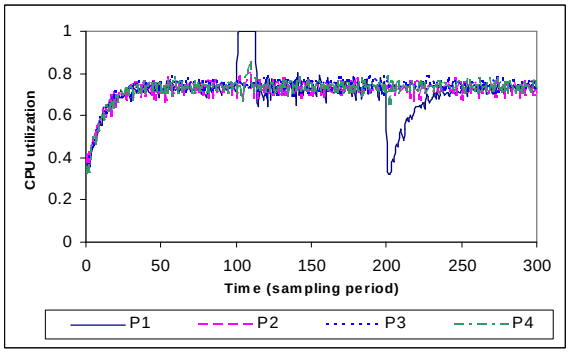
This empirical result is close to the analysis which shows the system should remain stable when the gain is below 5.95 (see Section 4.3.2).

The standard deviation of utilization indicates the intensity of oscillation. As the execution-time factor increases from 0.2 to 3, the standard deviation remains less than 0.05 and the average utilization remains within 0.02 to the set point. These results demonstrate that EUCON can enforce the same utilization guarantees when execution times deviate from the estimates as long as the execution-time factor remains below 3. However, the standard deviation is higher than 0.05 for execution-time factors between 4 and 6, although the system is analytically stable in this range. This result is consistent with our analysis in Section 4.3.2 that pessimistic estimation on execution times will reduce oscillation without underutilizing the CPUs.

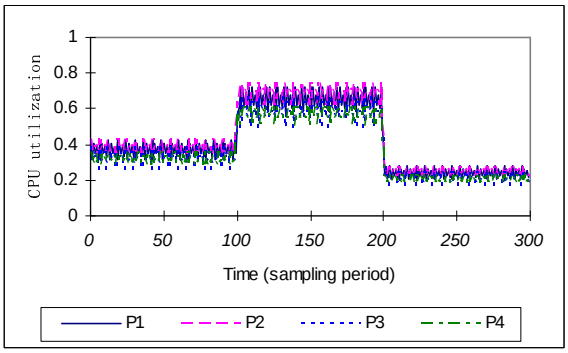
We then repeat our experiments under MEDIUM in order to evaluate the system performance under more complex settings. Figure 4.4(b) plots the mean and standard deviation of utilization on processor $P_1$ under different execution-time factors (the performance on other processors is similar to $P_1$ and is not shown due to space limit). For comparison, the expected utilization under OPEN (computed based on (4.12)) is also plotted. OPEN causes underutilization when execution times are overestimated ($etf < 1$), and causes overload when execution times are underestimated ($etf > 1$). In contrast, EUCON provides acceptable utilization guarantees for any tested execution-time factor within the range [0.1, 1]. In this range, the average utilization under EUCON remains within 0.02 to the utilization set point and the standard deviation remains below 0.05. For example, when etf = 0.1, the utilization under OPEN is only 0.073, while the average utilization under EUCON is 0.729 - the same as the utilization set point - with an standard deviation of 0.003. This result demonstrates EUCON can achieve desired utilization even when execution times are significantly overestimated. Similar to SIMPLE, the oscillation of utilization under MEDIUM also increases as execution times are underestimated. This result confirms our observation that pessimistic estimation of execution times should be used in the predictive controller in EUCON.



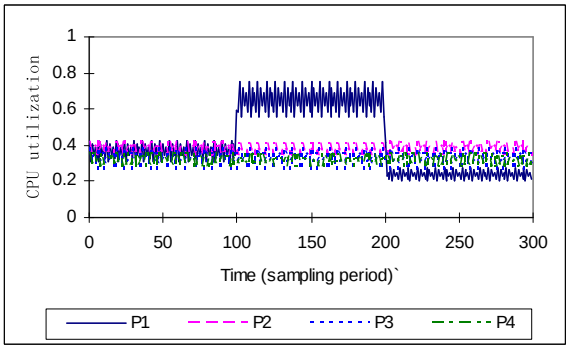

(a) Global fluctuation (EUCON)




(b) Local fluctuation on $P_1$ (EUCON)




(c) Global fluctuation (OPEN)




(d) Local fluctuation on $P_1$ (OPEN)

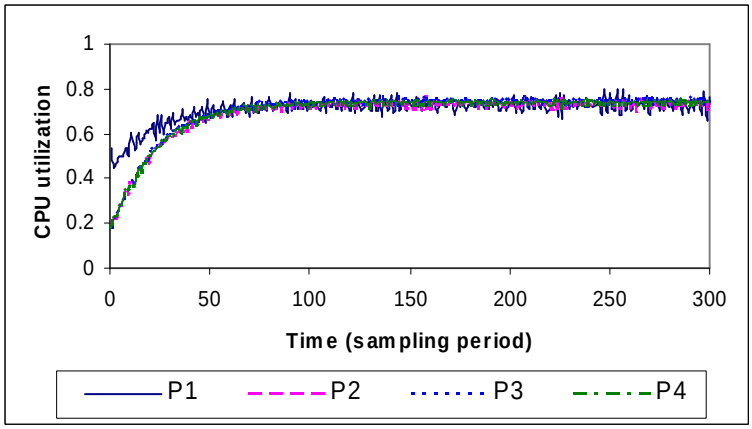
Figure 4.5: Utilization and task rates when execution times fluctuate at run time

### 4.4.4 Experiment II: Varying Execution Times

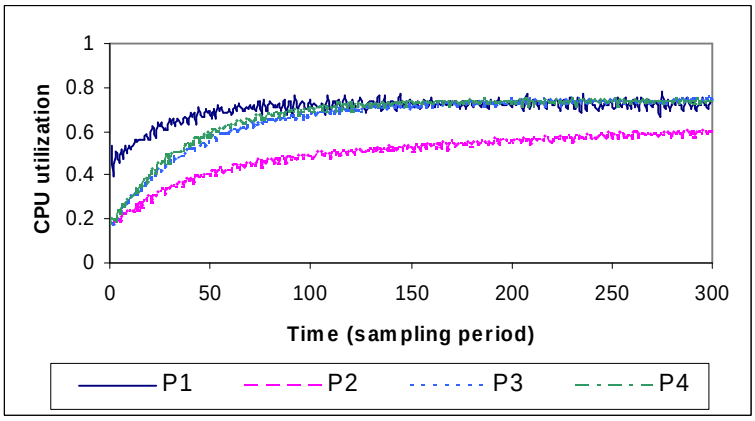
In Experiment II, execution times vary dynamically at run-time under the MEDIUM configuration. To investigate the robustness of EUCON we tested two scenarios of workload fluctuation. In the first set of runs, the average execution times on all processors change uniformly. In the second set of runs, only the average execution times on $P_1$ change dynamically. The first scenario represents global load fluctuation in the whole system, while the second scenario represents local fluctuation on a part of the system.

In each run with global workload fluctuation, the execution time factor is initially 0.5. At time $100T_s$, it increases to 0.9 causing an 80% increase in the execution times of all subtasks. At time $200T_s$, the execution-time factor drops to 0.33 causing a 67% decrease in execution times. Such instantaneous variation in workload stress tests the system capability of handling workload fluctuations. As shown in Figure 4.5(a), EUCON enforces the utilization set points on all processors despite significant variations in execution times. At time $100T_s$, all processors are suddenly overloaded due to the increase in execution times. EUCON responds to the deviation from the utilization set points by decreasing task rates. The utilization on all processors re-converges to their set points within $20T_s$. At time $200T_s$, the utilization dropped dramatically causing EUCON to increase task rates until the utilization on all processors regain to their set points. The system settling time after $200T_s$ is longer than that follows $100T_s$. As discussed in Section V this is because the system gain is smaller during interval [$200T_s$, $300T_s$] than [$100T_s$, $200T_s$]. The system maintains stability and avoids significant oscillation throughout the run despite variations in execution times. In contrast, Figure 4.5(c) shows that the utilization under OPEN fluctuates significantly because it cannot adapt to the workload variations.

In each run with local workload fluctuation, the execution-time factor on $P_1$ follows the same variation as that in global fluctuation, but all the other processors have a fixed execution-time factor of 0.5. As shown in Figure 4.5(b), the utilization of $P_1$ converges to its set point after the significant variation of execution times at $120T_s$ and $250T_s$, respectively. The settling times under local workload fluctuation are close to those under global workload fluctuation. We also observe that the other processors experience only slight utilization fluctuation after the execution times change on $P_1$.




(a) EUCON




(b) FC-U-E2E

Figure 4.6: Utilization under EUCON and FC-U-E2E (etf = 0.2, MEDIUM)

This result demonstrates that EUCON effectively handles the coupling among processors during rate adaptation. In contrast, OPEN fails to maintain steady utilization on $P_1$ in face of local workload fluctuation (as shown in Figure 4.5(d)).

### 4.4.5 Experiment III: Comparison with FC-U-E2E

A premise of this work is that the MIMO approach adopted by EUCON can outperform the SISO control approach. SISO control cannot handle the coupling among

processors effectively - especially when the utilization on different processors are unbalanced. In this situation, the task rates computed by different controllers may become inconsistent with each other due to the unbalanced utilization on different processors. We now compare the performance of EU-CON and FC-U-E2E under an unbalanced workload. The workload used in this experiment is the same as MEDIUM except that the execution times on processor $P_1$ are higher. The execution time factor remains at 0.2 in each run. As shown in Figure 4.6(a), the utilization on all processors converge to their set points despite the difference in initial values when EUCON is used. The performance of FC-U-E2E is shown in Figure 4.6(b). The utilization on $P_1$ follows a similar trajectory as under EUCON. However, all the other three processors suffer from significantly longer settling times. For instance, while it only takes about $60T_s$ for $P_4$ to reach its set point under EUCON, it fails to reach its set point in the end of the run ($300T_s$). Long settling times are undesirable because systems need to quickly recover from load variation.

We now analyze what causes the poor performance of FC-U-E2E. After $P_1$ reaches the set point at time $50T_s$, its Proportional controller stops increasing the rates of all tasks with sub-tasks on this processor. Because all tasks must execute at the lowest rate given by any controllers in FC-U-E2E, their rates will stop increasing, even if the controllers on the other processors need them to do so in order to reach their set points. This effectively slows down the convergence of processors $P_2$-$P_4$ to their set points. Actually, FC-U-E2E can eventually reach the set points only because every processor has a local task whose rate can be changed independently from other processors. P2 has the longest settling time because it shares four end-to-end tasks with $P_1$, while each of $P_3$ and $P_4$ only shares two with $P_1$. Hence the utilization of $P_2$ is particularly affected by the controller on $P_1$. After $P_3$ and $P_4$ both reach their set points, the utilization increase of $P_2$ becomes even slower since only its local task can increase its rate in this case. Compared with FC-U-E2E, a key advantage of EUCON lies in its capability to handle the coupling among multiple processors. Furthermore, MPC provides a theoretic framework to analyze system stability under a wide range of execution-time factors.

### 4.4.6 Overhead

To estimate the run-time overhead of the controller, we measure the execution time of the least squares solver which dominates the computation cost of the controller. In the simulations with the MEDIUM configuration on a 2GHz Pentium 4 PC with 256MB RAM, each invocation of the solver in MATLAB takes less than 9ms (corresponding to less than 1% CPU utilization when the sampling period is 1 sec). This result indicates the overhead of the controller is acceptable for a range of applications. Since this preliminary result is based on the solver in the MATLAB environment, it is not a precise benchmark for a controller implemented in native code. Evaluation of EUCON in a real middleware environment is presented in Chapter 7.

## 4.5 Summary

EUCON features a model predictive controller to handle the coupling among multiple processors and constraints based a mathematical model that characterizes the dynamics of distributed systems with end-to-end tasks. Both stability analysis and simulation results demonstrate that EUCON can maintain desired utilization on multiple processors when task execution times are significantly overestimated and change dynamically at run-time. EUCON also outperforms both open-loop scheduling and a FCS algorithm based on SISO linear control.

# Chapter 5

# DEUCON: Decentralized Control

In this chapter, we present the *Decentralized* End-to-end Utilization CONtrol (DEUCON) algorithm for large DRE systems with end-to-end tasks. In contrast to the centralized control scheme presented in Chapter 4, DEUCON employs a peer-to-peer control structure with a separate local controller $C_i$ on each master processor $P_i$. Each controller only coordinates with a small number of processors called its (logical) *neighbors*. Specifically, the contributions of this chapter are four-fold.

- We propose a new approach for decomposing the global multi-processor utilization control problem into local subproblems to facilitate the design of decentralized control solutions.
- We describe the DEUCON algorithm featuring a novel peer-to-peer control structure that enforces desired utilizations of multiple processors through localized coordination among controllers.
- We give control analysis based on the *distributed model predictive control* (DMPC) theory [16] which establishes the stability properties of the DEUCON algorithm in face of uncertain task execution times.
- We present simulation results showing that DEUCON can provide robust utilization guarantees to multiple processors through task rate adaptation[1], while achieving scalability by effectively distributing the computation and communication overhead to local controllers.

[1]Other control strategies such as task migration, quality level adaptation and possible combinations of them are subjects of our future research.

In this chapter, we first discuss the limitations of centralized control in large-scale DRE systems. We then present an approach for decomposing the global system model into localized control subproblems, as a foundation of our decentralized control design. Based on this foundation, we describe the design and stability analysis of the DEUCON algorithm. Section 5.3 evaluates DEUCON with simulations. Section 5.4 summaries this chapter.

## 5.1 Limitations of Centralized Control

Centralized control algirthms (i.e. the EUCON algorithm described in Chapter 4) rely on a single centralized controller to manage the adaptation of multiple processors. Hence, in large DRE systems, a centralized control scheme has several disadvantages.

1. The run-time overhead depends on the size of an entire DRE system. Specifically, the worst-case computational complexity of a model predictive controller is polynomial in the total number of tasks and the total number of processors in the system. Furthermore, since every processor in the system needs to communicate with the controller in every sampling period, the processor executing the controller can become a communication bottleneck. Therefore, a centralized control scheme cannot scale effectively in large DRE systems.

2. The control design of EUCON assumes that communication delays between the control processor and other processors are negligible compared to the sampling period of the controller. This assumption may not hold in networks with significant delays such as the Internet and wireless sensor networks.

3. The processor executing the controller is a single point of failure. The entire system will lose the capability to adapt to the environment if it fails.

Centralized solutions are therefore not suitable for large-scale DRE systems (e.g., wide-area power grid management). In this section, we develop *decentralized* control algorithms to improve the scalability and reliability of adaptive utilization control in DRE systems.

## 5.2 Design of DEUCON

In contrast to the centralized control scheme adopted by EUCON, DEUCON employs a peer-to-peer control structure with a separate local controller $C_i$ on each master processor $P_i$. Each controller only coordinates with a small number of processors called its (logical) *neighbors*. A fundamental design challenge is to achieve system stability and desired utilizations without global information. In this section, we present the design of DEUCON based on a distributed model predictive control (DMPC) framework. As a foundation of our control design, we first present a dynamic model of the entire system and an approach for decomposing the global system model into localized control subproblems. We then describe the design and control analysis of the DEUCON algorithm based on the dynamic models.

### 5.2.1 Global System Model

In a control-theoretic methodology, a control algorithm should be designed based on a model of the system. As described in Chapter 4, a DRE system can be approximated by the following *global* system model:

$$\mathbf{u(k+1)} = \mathbf{u(k)} + \mathbf{GF\Delta r(k)} \tag{5.1}$$

The vector $\mathbf{\Delta r(k)}$ represents the changes in task rates. The *subtask allocation matrix*, $\mathbf{F}$, is an $n \times m$ matrix, where $f_{ij} = c_{jl}$ if a subtask $T_{jl}$ of task $T_j$ is allocated to processor $P_i$, and $f_{ij} = 0$ if no subtask of task $T_j$ is allocated to processor $P_i$. $\mathbf{F}$ captures the *coupling* among processors due to end-to-end tasks. $\mathbf{G} = diag[g_1 \dots g_n]$ where $g_i$ represents the ratio between the change in the actual utilization and its estimation. The exact value of $g_i$ is *unknown* due to the unpredictability in execution times. Note that $\mathbf{G}$ describes the effect of uncertainty in workload on the utilization of a DRE system. As an example, Figure 5.1 shows a DRE system with five processors and five tasks. It is modeled by (5.1) with the following parameters:

$$\mathbf{u(k)} = \begin{bmatrix} u_1(k) \\ u_2(k) \\ u_3(k) \\ u_4(k) \\ u_5(k) \end{bmatrix}, \mathbf{F} = \begin{bmatrix} c_{11} & 0 & 0 & 0 & c_{51} \\ c_{12} & c_{22} & 0 & 0 & 0 \\ 0 & c_{21} & c_{31} & 0 & 0 \\ 0 & 0 & c_{32} & c_{41} & 0 \\ 0 & 0 & c_{33} & c_{42} & 0 \end{bmatrix},$$

$$\mathbf{G} = \begin{bmatrix} g_1 & 0 & 0 & 0 & 0 \\ 0 & g_2 & 0 & 0 & 0 \\ 0 & 0 & g_3 & 0 & 0 \\ 0 & 0 & 0 & g_4 & 0 \\ 0 & 0 & 0 & 0 & g_5 \end{bmatrix}, \mathbf{\Delta r(k)} = \begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \\ \Delta r_3(k) \\ \Delta r_4(k) \\ \Delta r_5(k) \end{bmatrix}$$

### 5.2.2 Problem Decomposition

Although our previous work showed that the above global system model is sufficient for designing a centralized controller for EUCON [60], it cannot be used for designing decentralized control algorithms because it includes information about the entire system. To address this problem, we propose a new approach to decompose the global utilization control problem into a set of localized subproblems.

From a local controller $C_i$'s perspective, the goal of decomposition is to partition the set of system variables into three subsets, including *local variables* on host processor $P_i$, *neighbor variables* on $P_i$'s neighbors, and all other variables in the system. $C_i$'s subproblem only includes its local and neighbor variables. A key feature of our decomposition scheme is that it balances two conflicting goals. On one hand, the number of neighbor variables should be minimized to improve system scalability. On the other hand, the neighbor variables must capture the coupling among processors so that local controllers can achieve global system stability through coordination in their neighborhoods.

We give several definitions before presenting our decomposition scheme.

**Definition** Processor $P_j$ is $P_i$'s *direct neighbor* if (1) $P_j$ has a subtask belonging to an end-to-end task mastered by $P_i$ and (2) $P_j$ is not $P_i$ itself.

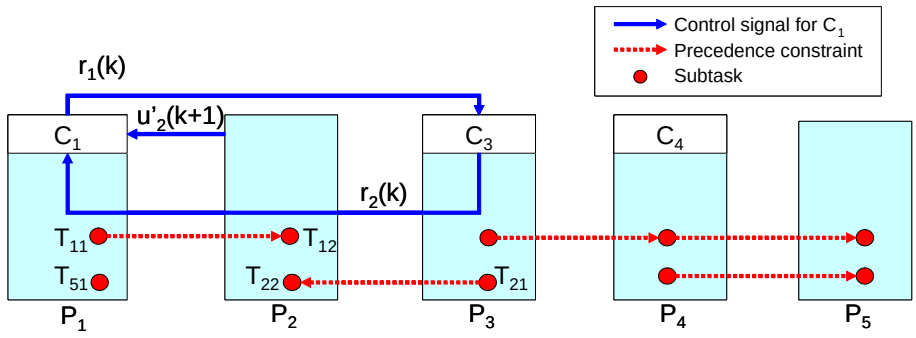



Figure 5.1: Data exchange between $C_1$ and its neighbors (other data exchanges are not shown)

**Definition** The *concerned tasks* of $P_i$ are the tasks which have subtasks located on $P_i$ or $P_i$'s direct neighbors.

**Definition** Processor $P_j$ is $P_i$'s *indirect neighbor* if (1) $P_j$ is the master processor of any of $P_i$'s concerned tasks and (2) $P_j$ is not $P_i$'s direct neighbor or $P_i$ itself.

For example, we consider controller $C_1$ in the system shown in Figure 5.1. $P_1$ has one direct neighbor ($P_2$) due to task $T_1$ mastered by $P_1$. Its concerned tasks include $T_1$, $T_5$ and $T_2$ (which has a subtask on direct neighbor $P_2$). Hence $P_3$, the master processor of $T_2$, is $P_1$'s indirect neighbor.

The subproblem of a controller includes a set of utilizations as *controlled variables*, and a set of task rates as *manipulated variables*. In our decomposition scheme, the controlled variables of controller $C_i$ include $u_i(k)$, the host processor $P_i$'s utilization, and $UD_i(k)$, the set of utilizations of $P_i$'s direct neighbors. $UD_i(k)$ are considered $C_i$'s neighbor variables because they are affected by the rates of tasks mastered by $P_i$. Since each concerned task contributes to the utilizations of $P_i$ and/or its direct neighbors, $C_i$'s manipulated variables include the rates of all of $P_i$'s concerned tasks. Note that a concerned task may be mastered by $P_i$ itself, its direct neighbor, or its indirect neighbor. For example, $C_1$ has two controlled variables, $u_1(k)$ and $u_2(k)$, and three manipulated variables $r_1(k)$, $r_2(k)$ and $r_5(k)$.

Let set $NR_i(k)$ denote the rates of all of $P_i$'s concerned tasks, and set $NU_i(k) = UD_i(k) \cup \{u_i(k)\}$, the subproblem of $C_i$ then becomes the following localized constrained optimization problem within its neighborhood:

$$\min_{NR_i(k)} \sum_{u_l(k) \in NU_i(k)} (B_l - u_l(k+1))^2 \tag{5.2}$$

subject to

$$R_{min,j} \leq r_j(k) \leq R_{max,j} \qquad (r_j(k) \in NR_i(k))$$

In contrast to the global model (5.1) used in EUCON, each controller in DEUCON has a localized model which only includes its local and neighbor variables. This local model of $C_i$ is described as:

$$\mathbf{nu_i}(\mathbf{k+1}) = \mathbf{nu_i}(\mathbf{k}) + \mathbf{G_i F_i \Delta nr_i}(\mathbf{k}) \tag{5.3}$$

where $\mathbf{nu_i}(\mathbf{k})$ and $\mathbf{nr_i}(\mathbf{k})$ are vectors comprised of all elements in $NU_i(k)$ and $NR_i(k)$, respectively. $\mathbf{G_i}$ and $\mathbf{F_i}$ are defined in the same way as $\mathbf{G}$ and $\mathbf{F}$ in (5.1), but include only the processors in $NU_i(k)$ and the task rates in $NR_i(k)$.

For example, the controller $C_1$ shown in Figure 5.1 is modeled with the following parameters.

$$\mathbf{nu_1}(\mathbf{k}) = \begin{bmatrix} u_1(k) \\ u_2(k) \end{bmatrix}, \mathbf{F_1} = \begin{bmatrix} c_{11} & 0 & c_{51} \\ c_{12} & c_{22} & 0 \end{bmatrix}, \mathbf{G_1} = \begin{bmatrix} g_1 & 0 \\ 0 & g_2 \end{bmatrix}, \mathbf{\Delta nr_1}(\mathbf{k}) = \begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \\ \Delta r_5(k) \end{bmatrix}$$

From (5.3), $C_1$'s local model is

$$\begin{aligned} u_1(k+1) &= u_1(k) + g_1(c_{11}\Delta r_1(k) + c_{51}\Delta r_5(k)) \\ u_2(k+1) &= u_2(k) + g_2(c_{12}\Delta r_1(k) + c_{22}\Delta r_2(k)) \end{aligned}$$

### 5.2.3 Localized Feedback Control Loop

We now present DEUCON's localized feedback control loop based on our decomposition scheme. The execution of a controller $C_i$ at each sampling point $k$ includes three steps:

1. *Local control computation:* $C_i$ executes an MPC algorithm to solve its local subproblem. The feedback input to the control algorithm includes (1) $u_i(k)$ from the local utilization monitor, (2) a set of *predicted utilizations* $UD'_i(k)$ of its direct neighbors, and (3) the rates of concerned tasks, $NR_i(k-1)$ in the last sampling period. The output from the controller $C_i$ includes the new rates for concerned tasks, $NR_i(k)$. The details of the control algorithm are presented in Section 5.2.4.

2. *Local actuation:* The rate modulator on $P_i$ changes the rates of the set of tasks mastered by $P_i$ according to the control input from $C_i$. The other task rates in the control input will be ignored because they are not mastered by $P_i$.

3. *Data exchange among neighbors:* $C_i$ sends its *predicted utilization* at the next sampling point, $u'_i(k+1)$, to other controllers of which it serves as a direct neighbor. $C_i$ also sends the rates of tasks mastered by $P_i$ to those controllers which have these tasks as their concerned tasks. In addition, $C_i$ receives new predicted utilizations from its direct neighbors, and the actual rates of the concerned tasks which are not mastered by itself, from its direct and indirect neighbors. They will be used for the local control computation at the next sampling point $(k+1)$.

Compared to centralized control schemes, a fundamental advantage of DEUCON is that both the computation and communication overhead of a controller depends on the size of its neighborhood instead of the entire system. This feature allows DEUCON to scale effectively in many large DRE systems.

Another important advantage of DEUCON is that it can tolerate considerable network delays. Note that in step 1, the *predicted* utilizations $UD'_i(k)$ (instead of $UD_i(k)$) are provided by $C_i$'s direct neighbors in the previous sampling period. This is because

$UD_i(k)$ is not instantaneously available to $C_i$ at time $kT_s$ due to network delays. $UD'_i(k)$ is predicted based on $UD_i(k-1)$ at time $(k-1)T_s$, as a substitute for $UD_i(k)$ to be transmitted over the network during interval $[(k-1)T_s, kT_s)$. Each element $u'_j(k) \in UD'_i(k)$ is calculated using the following reference trajectory from measured utilization $u_j(k-1)$ to its set point $B_j$ over the following $P$ sampling periods.

$$ref_j((k-1)+l|k-1) \;=\; B_j - e^{-\frac{T_s}{T_{ref}}l}(B_j - u_j(k-1)) \qquad (1 \le l \le P) \quad (5.4)$$

where $T_{ref}$ is the time constant that specifies the speed of system response. $P$ is called the *prediction horizon*. The notation $x((k-1)+l|k-1)$ means that the value of variable $x$ at time $((k-1)+l)T_s$ depends on the conditions at time $(k-1)T_s$. The value of $ref_j(k|k-1)$ is assigned to $u'_j(k)$. Since $UD'_i(k)$ can take the entire sampling period to transmit, DEUCON can tolerate much longer communication delays than EUCON which assumes the delays to be negligible.

DEUCON can also improve system fault-tolerance by avoiding a centralized controller which is a single point of failure in the whole system. In DEUCON, even if the system failure of a processor may disable a local controller, the subtasks on the failed processor can be immediately migrated to their backup processors, and then be effectively controlled by other local controllers there. As a result, single processor failures will not cause the system to lose control in DEUCON.

### 5.2.4 Controller Design

DEUCON employs a local controller on each *master* processor. Non-master processors do not need controllers because they cannot change the rate of any task. For the example shown in Figure 5.1, processors $P_1$, $P_3$ and $P_4$ each have a controller, while $P_2$ and $P_5$ do not have controllers because they are not master processors for any tasks. This feature reduces the overhead of DEUCON.

We design a model predictive control algorithm [15] for controller $C_i$. We choose model predictive control because it can deal with coupled MIMO control problems with constraints on the actuators. At every sampling point, the controller computes an input

trajectory in the following $M$ sampling periods, e.g., $\mathbf{\Delta nr_i(k)}, \mathbf{\Delta nr_i(k+1|k)}, \ldots$ $\mathbf{\Delta nr_i(k+M-1|k)}$, that minimizes the following cost function under the rate constraints.

$$\begin{aligned} V_i(k) &= \sum_{l=1}^{P} \|\mathbf{nu_i(k+l|k)} - \mathbf{ref_i(k+l|k)}\|^2 \\ &+ \sum_{l=0}^{M-1} \|\mathbf{\Delta nr_i(k+l|k)} - \mathbf{\Delta nr_i(k+l-1|k)}\|^2 \end{aligned} \tag{5.5}$$

where $P$ is the *prediction horizon*, and $M$ is the *control horizon*. The first term in the cost function represents the *tracking error*, i.e., the difference between the utilization vector $\mathbf{nu_i(k+l|k)}$, which is predicted based on (5.6), and the reference trajectory $\mathbf{ref_i(k+l|k)}$ defined in (5.4). The controller is designed to track the exponential reference trajectory that converges to the set points so that the closed-loop system behaves like a desired linear system. By minimizing the tracking error, the closed-loop system will also converge to the utilization set points. The second term in the cost function represents the *control penalty*. The control penalty term causes the controller to minimize the changes in the control input.

The controller predicts the cost based on the following *approximate* model:

$$\mathbf{nu_i(k+1)} = \mathbf{nu_i'(k)} + \mathbf{F_i \Delta nr_i(k)} \tag{5.6}$$

The above model has two differences from the *actual* system model (5.3). First, the utilizations of direct neighbors are approximated by their predicted utilizations $\mathbf{nu_i'(k)}$, where $\mathbf{nu_i'(k)}$ is a vector comprised of all elements in $NU_i'(k)$. As discussed in Section 5.2.3, this approximation allows DEUCON to tolerate network delays. Second, because the real system gains $\mathbf{G_i}$ in system model (5.3) are unknown in unpredicted environments, our controller assumes $\mathbf{G_i} = diag[1 \ldots 1]$, i.e., the controller assumes that the estimated execution times are accurate. Although this approximate model is not an exact characterization of the real system, the closed-loop system under our controller can still maintain stability and guarantee desired utilization set points as long as $\mathbf{G_i}$ are within a certain range (see analysis and simulation results in Sections 5.2.5 and 5.3.2). This is due to the coordination scheme and the online feedback controls used in our distributed model predictive control algorithm.

The controller computes the input trajectory $\mathbf{\Delta nr_i(k)}, \mathbf{\Delta nr_i(k+1|k)}, \ldots$ $\mathbf{\Delta nr_i(k+M-1|k)}$ that minimizes the cost function subject to the rate constraints. This constrained optimization problem can be transformed to a standard constrained least square problem [63][60]. Controller $C_i$ can then use a standard least-square solver to solve this problem on-line. The detailed transformation can be found in Appendix A. The worst-case computation complexity of the solver is polynomial in the numbers of tasks and processors in the localized model (5.6). More specifically, our constrained least-square optimization is a convex nonlinear optimization, for which interior point methods require $O(n)$ Newton iterations [103], where n is the number of optimization variables. Since each Newton iteration requires $O(n^3)$ algebraic operations, the worst- case computation complexity of the solver is cubic in the number of tasks and processors in the localized model.

Once the input trajectory is computed, only the first element $\mathbf{\Delta nr_i(k)}$ is applied as the control input and sent to the rate modulators. At next sampling point, the prediction horizon slides by one sampling period and the control input is computed again as a solution to the constrained optimization problem based on the utilization feedbacks from its direct neighbors and itself.

### 5.2.5 Stability Analysis

A fundamental benefit of the control-theoretic approach is that it enables us to prove the utilization guarantees provided by DEUCON despite uncertainties in task execution times. We say that a DRE system is *stable* if the utilizations $\mathbf{u}$ converge to the desired set points $\mathbf{B}$, that is, $\lim_{\mathbf{k}\to\infty} \mathbf{u}(\mathbf{k}) = \mathbf{B}$. In this subsection, we present a method that, given a system and a range of variations in task execution times, allows to analytically assess the stability and robustness of DEUCON. To ensure that the system can be stabilized, the constrained optimization problem must be feasible, i.e., there exists a set of task rates within their acceptable ranges that can make the utilization on every processor equal to its set point. If the problem is infeasible, no controller can guarantee the set point through rate adaptation. The system feasibility and controllability issues are addressed in Chapter 8 by adjusting end-to-end task allocation.

In DEUCON, each controller solves a finite horizon optimal tracking problem. Based on optimal control theory [51], the local control decision is a linear function of the current utilization and the set point of the local CPU, the utilizations of its direct neighbors and the previous decisions for its manipulated tasks and concerned tasks. We now outline the process for analyzing the stability of the system controlled by DEUCON.

1. Compute the feedback and feed-forward matrices for each local controller $i$ by solving its local control input $\mathbf{\Delta nr_i}$ based on the local approximate system model (5.6) and reference trajectory (5.4). The solution is in the following form:

$$\mathbf{\Delta nr_i}\,(\mathbf{k}) = \mathbf{K_i nu_i'}\,(\mathbf{k}) + \mathbf{H_i \Delta nr_i}\,(\mathbf{k-1}) + \mathbf{E_i B_i} \tag{5.7}$$

2. Construct the feedback and feed-forward matrices for the whole system (5.1) based on those for local system models derived in Step 1.

$$\mathbf{\Delta r}\,(\mathbf{k}) = \mathbf{K u}\,(\mathbf{k}) + \mathbf{L u}\,(\mathbf{k-1}) + \mathbf{H \Delta r}\,(\mathbf{k-1}) + \mathbf{EB} \tag{5.8}$$

   This is a dynamic controller. The stability analysis needs to consider the composite system consisting of the dynamics of the original system and the controller.

3. Derive the closed-loop model of the composite system by substituting the control inputs derived in Step 2 into the *actual* system model described by (5.1). The closed-loop composite system is in the form

$$\begin{bmatrix} \mathbf{u}\,(\mathbf{k+1}) \\ \mathbf{u}\,(\mathbf{k}) \\ \mathbf{\Delta r}\,(\mathbf{k}) \end{bmatrix} = \begin{bmatrix} \mathbf{I+GFK} & \mathbf{GFL} & \mathbf{GFH} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{K} & \mathbf{L} & \mathbf{H} \end{bmatrix} \begin{bmatrix} \mathbf{u}\,(\mathbf{k}) \\ \mathbf{u}\,(\mathbf{k-1}) \\ \mathbf{\Delta r}\,(\mathbf{k-1}) \end{bmatrix} + \begin{bmatrix} \mathbf{GFE} \\ \mathbf{0} \\ \mathbf{E} \end{bmatrix} \mathbf{B} \tag{5.9}$$

   where $\mathbf{I}$ is the identity matrix. Note that the closed-loop system model is a function of $\mathbf{G}$.

4. Derive the stability condition of the closed-loop system (5.9) given a range of $\mathbf{G}$ values. According to the control theory, if all poles locate inside the unit circle in the complex space and the DC gain matrix from the control input $\mathbf{\Delta r}(\mathbf{k})$ to

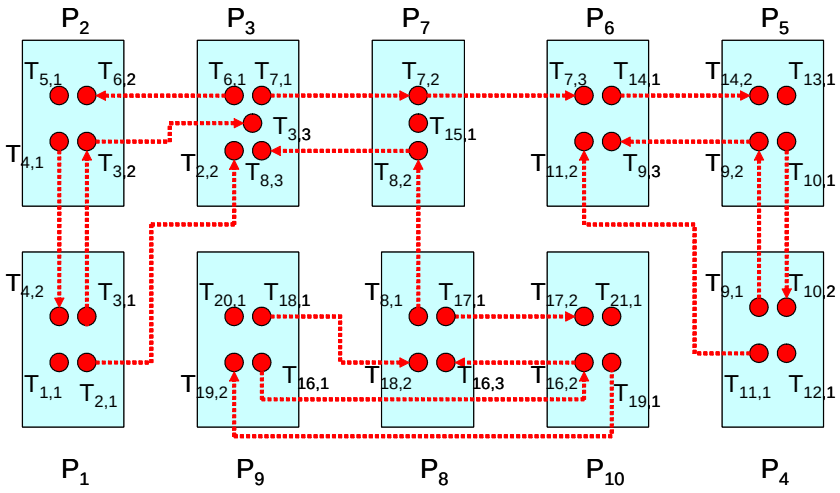



Figure 5.2: A medium size workload

the system state $\mathbf{u}(\mathbf{k})$ is the identity matrix, the utilizations $\mathbf{u}(\mathbf{k})$ will converge to the set point.

The details of the above steps follow the method given in [16]. We have developed a MATLAB program to perform the above stability analysis procedure automatically.

To illustrate our method for stability analysis, we have applied the stability analysis approach to the example system described in Figure 5.2. The detailed analysis is available in Appendix D. The result shows that all poles of the closed-loop system are within the unit circle for $0 < g < 2$. Furthermore, the DC gain of the closed-loop system is the identity matrix for $0 < g < 2$. Therefore, the system is stable. Our analysis proves that DEUCON can provide robust utilization guarantees to the example system even when actual execution times deviate significantly from the estimation. For instance, our results indicate that DEUCON can converge to the desired utilizations on all processors even if the execution time of every task is 90% lower ($g = 0.1$) or 90% higher ($g = 1.9$) than the estimation as long as the range of task rates are not violated. We validate this analysis through simulations presented in Section 5.3.

## 5.3 Simulation Results

In this section, we first describe the simulation settings. We then compare the performance and overhead of DEUCON and EUCON. We choose EUCON as the baseline for performance as it is the only available utilization control algorithm for DRE systems with end-to-end tasks. Previous results showed that EUCON significantly outperformed a common open-loop approach that assigned fixed task rates based on estimated execution times [60]. Finally, we evaluate the scalability of DEUCON in large systems using randomly generated workloads.

### 5.3.1 Simulation Setup

Our simulation environment is composed of an event-driven simulator implemented in C++ and a set of controllers implemented in MATLAB (R12). The simulator implements the utilization monitors, the rate modulators, and the distributed real-time system with an interface to the controllers. The subtasks on each processor are scheduled by the Rate Monotonic Scheduling (RMS) algorithm [54]. The precedence constraints among subtasks are enforced by the release guard protocol [91]. The controllers are based on the `lsqlin` least square solver in MATLAB. The simulator opens a MATLAB process and initializes all the controllers at start time. In the end of each sampling period, the simulator collects the local utilization, the predicted neighborhood utilizations and the concerned task rates for each controller, and then calls the controller in MATLAB. The controllers compute the control input, $\mathbf{\Delta r(k)}$, and return it to the simulator. The simulator then calls the rate modulators on each processor to adjust the rates of its mastered tasks.

Each task has its end-to-end deadline as $d_i = n_i/r_i(k)$, where $n_i$ is the number of subtasks in task $T_i$. Each end-to-end deadline is evenly divided into subdeadlines for its subtasks. The resultant subdeadline of each subtask $T_{ij}$ equals its period, $1/r_i(k)$. The schedulable utilization bound of RMS [54], $B_i = m_i(2^{1/m_i} - 1)$ is used as the utilization set point on each processor, where $m_i$ is the number of subtasks on $P_i$. All

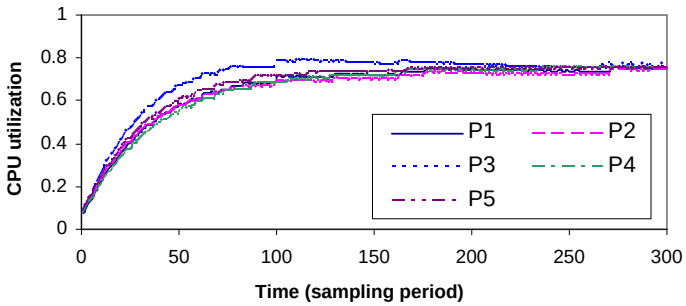
(sub)tasks meet their (sub)deadlines if the utilization set point on every processor is enforced[2].

A medium size workload (as shown in Figure 5.2) is used in our experiments. It includes 21 tasks (with a total of 40 subtasks) executing on 10 processors. There are 14 end-to-end tasks running on multiple processors and 7 local tasks. The controller parameters used for this workload include the prediction horizon as 2 and the control horizon as 1. The control period $T_s = 1000$ time units. The time constant $T_{ref}/T_s$ used in (5.4) is set as 4. Specific parameters of tasks are not shown due to space limitations.

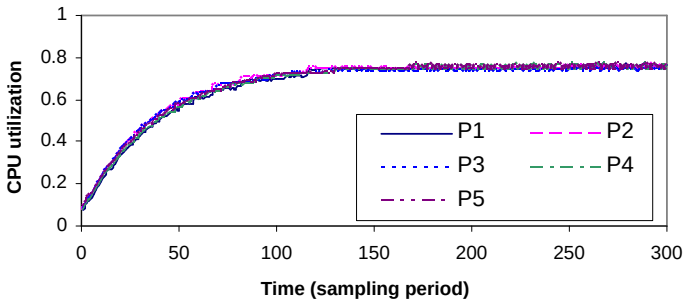
To evaluate the robustness of DEUCON when execution times deviate from the estimation, the execution time of each subtask $T_{ij}$ can be changed by tuning a parameter called the execution-time factor, $etf_{ij}(k) = a_{ij}(k)/c_{ij}$, where $a_{ij}$ is the actual execution time of $T_{ij}$. The execution time factor represents how much the actual execution time of a subtask deviates from the estimated one. The execution time factor (and hence the actual execution times) may be kept constant or changed dynamically in a run. When all subtasks share a same constant *etf*, it equals to the system gain on every processor in the model, i.e., $etf = g_{ii}(1 \leq i \leq m)$. In the following, we use *inversed etf* (*ietf*) defined by $ieft_{ij}(k) = 1/etf_{ij}(k)$ because we are more interested in the situation when execution times are overestimated (i.e. $etf < 1$)[3].

### 5.3.2 System Performance

In this subsection, we present two sets of simulation experiments. The first one evaluates DEUCON's system performance when task execution times deviate from the estimation. The second experiment tests DEUCON's ability to provide robust utilization guarantees when task execution times vary dynamically at run-time.




(a) DEUCON




(b) EUCON

Figure 5.3: CPU utilization of $P_1$ to $P_5$ (ietf=8)

### Steady Execution Times

In this experiment, all subtasks share a fixed execution-time factor (*ietf*) in each run. Since it is often difficult to estimate the execution times of real-time tasks precisely in DRE system, we stress-test DEUCON's performance when real execution time significantly deviate from their estimations. Figures 5.3(a) and (b) show the utilizations of processors $P_1$ to $P_5$ when execution times of tasks are *one-eighth* of their estimations. In this case, we can observe a noticeable difference in the transient state between DEUCON and EUCON. While the utilizations of EUCON follow the same trajectory, utilizations of DEUCON diverge in the middle of the run and then converge to their set points in the end. The reason for this divergence is that each controller in DEUCON only utilizes local information and makes local decision. Despite this slight difference in the transient state, all utilizations converge to their set points

[2]Other utilization bounds [50] can be used by DEUCON when the subdeadlines of subtasks are not equal to their periods

[3]In general, as discussed in [60], algorithms based on model predictive control and distributed model predictive control cause oscillation when the execution times are underestimated (i.e. $etf > 1$).

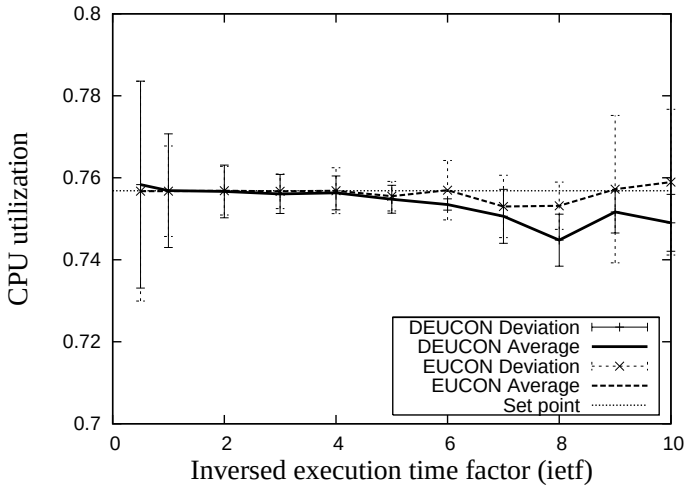



Figure 5.4: The average and deviation of the CPU utilization of $P_1$ with different execution times

within similar settling times. Both DEUCON and EUCON achieve desired utilization guarantees in steady states.

To examine DEUCON's performance under different execution time factors, we plot the mean and standard deviation of utilization on $P_1$ during each run in Figure 5.4. Every data point is based on the measured utilization $u(k)$ from time $200T_s$ to $300T_s$ to exclude the transient response in the beginning of each run. Both EUCON and DEUCON achieve desired utilizations for all tested execution-time factors within the $ietf$ range $[0.5, 10]$. In this range, the average utilizations under EUCON and DEUCON remain within $\pm 0.012$ to the utilization set points and the standard deviations remain below 0.025. However, when $ietf = 8$, DEUCON's performance is slightly worse than that of EUCON, as its average utilization is 0.012 lower than its set point. In addition, EUCON has a high deviation when $ietf = 9$, because $P_1$ has a longer settling time under EUCON. As a result, the system is still in its transient state for part of the interval $[200T_s, 300T_s]$. We also observe that both EUCON and DEUCON suffer a standard deviation of $\pm 0.025$ when $ietf = 0.5$. However, as a key benefit, both EUCON and DEUCON can achieve desired utilizations even when execution times are severely overestimated. This capability is in sharp contrast to open-loop approaches which are based on schedulability analysis. Open-loop underutilizes the processors in such cases.

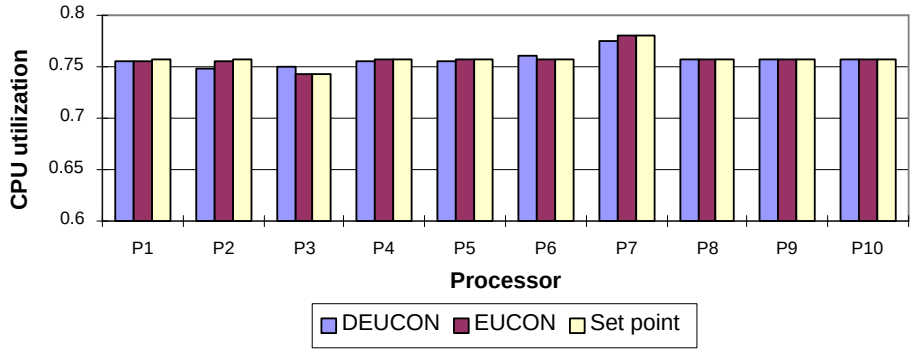


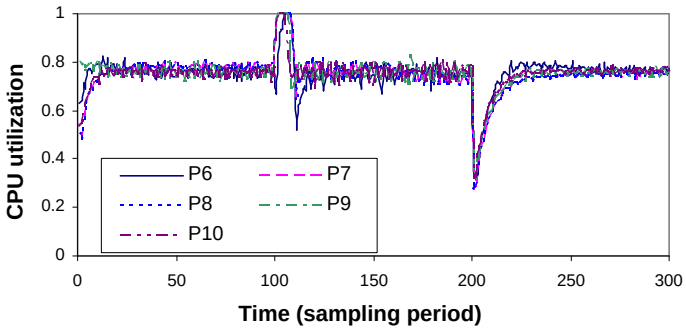

Figure 5.5: Average CPU utilization (ietf=5)

To further investigate the CPU utilizations on other processors, Figure 5.5 plots the average utilizations of all processors when *ietf* is 5. The deviations of all utilizations are less than 0.008. We observe that on $P_2$ to $P_7$, the difference between the utilizations and the set points for DEUCON are slightly larger than that of EUCON. However, all the differences are within the $\pm 0.009$ range. In practice, such small steady-state errors can be handled by setting the set points to slightly lower than the schedulable utilization bounds.

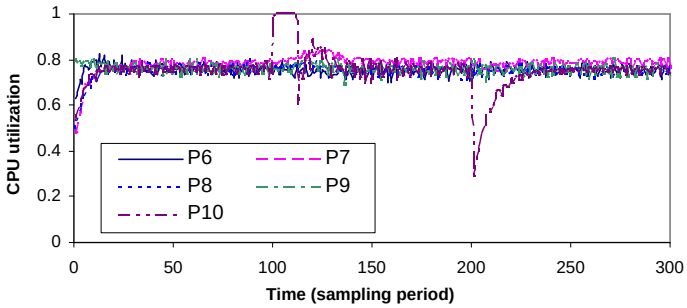
In summary, the simulation results demonstrate that DEUCON can achieve almost the same performance as EUCON, for a wide range of *ietf* ($[0.5, 10]$ in our experiments). We also note that the range of *ietf* corresponds to a system gain $g$ in a range $[0.1, 2]$. Therefore, our simulation results validate the correctness of our stability analysis presented in Section 5.2.5.

### Varying Execution Times

In this experiment, execution times vary *dynamically* at run-time. To investigate the robustness of DEUCON we tested two scenarios of workload fluctuation. In the first set of runs, the average execution times on all processors change simultaneously. In the second set of runs, only the execution times on $P_{10}$ change dynamically, while those on the other processors remain unchanged. The first scenario represents *global* load fluctuation, while the second scenario represents *local* fluctuation on a part of the system.




(a) Global fluctuation



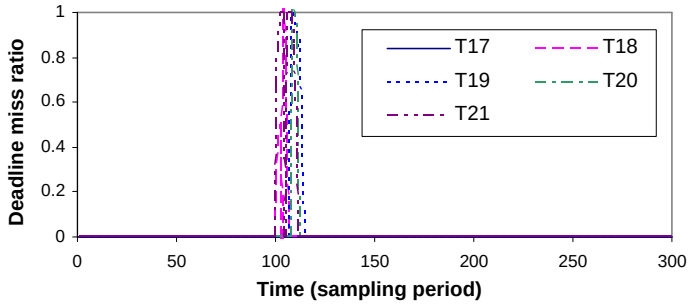

(b) Local fluctuation on $P_{10}$

Figure 5.6: CPU utilization of $P_6$ to $P_{10}$ when execution times fluctuate at run-time

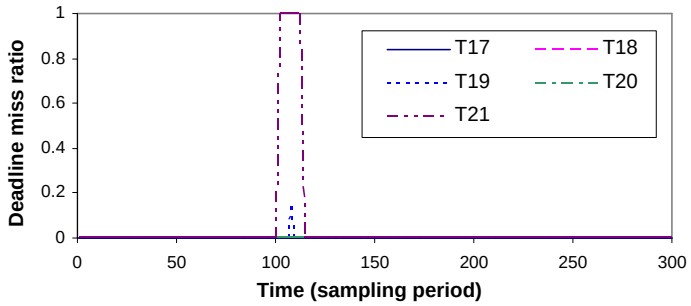
Figure 5.6(a) shows a typical run with global workload fluctuation[4]. The *ietf* is initially 1.0. At time $100T_s$, it is decreased to 0.56, which corresponds to an 79% increase in the execution times of all subtasks such that all processors are suddenly overloaded. Figure 5.7(a) shows that the deadline miss ratios of tasks $T_{17}$ to $T_{21}$ increase suddenly from zero to almost 100%[5]. This kind of significant deadline misses is undesired to most real-time applications. DEUCON responds to the overload by decreasing task rates which causes the utilizations on all processors to re-converge to their set points within $20T_s$. As a result, all end-to-end tasks are able to meet their deadlines again. At time $200T_s$, the *ietf* is increased to 1.67 corresponding to a 66% decrease in execution times. The utilizations on all processors drop sharply, causing DEUCON to dramatically increase task rates until the utilizations re-converge to their set points. The system maintains stability and avoids any significant oscillation throughout the run, despite the variations in execution times.

[4]Only the results of $P_6$ to $P_{10}$ are included for clarity. The results of $P_1$ to $P_5$ are similar.

[5]We choose to show the deadline miss ratios of tasks $T_{17}$ to $T_{21}$ because they are located on $P_6$ to $P_{10}$ and three of them ($T_{17}$, $T_{19}$ and $T_{21}$) are located on $P_{10}$ which is chosen to show local fluctuation.




(a) Global fluctuation




(b) Local fluctuation on $P_{10}$

Figure 5.7: Deadline miss ratio of $T_{17}$ to $T_{21}$ when execution times fluctuate at run-time

In each run with local workload fluctuation, the *ietf* on $P_{10}$ follows the same variation as the global fluctuation, while all the other processors have a fixed *ietf* of 1.0. As shown in Figure 5.6(b), the utilization of $P_{10}$ converges to its set point after the significant variation of execution times at $120T_s$ and $250T_s$, respectively. We also observe that the other processors experience only slight utilization fluctuation after the execution times change on $P_{10}$. This result demonstrates that DEUCON effectively handles the coupling among processors during rate adaptation. Figure 5.7(b) shows that only tasks $T_{19}$ and $T_{21}$ have deadline misses during the execution time increase on processor $P_{10}$. That is because $T_{19}$ and $T_{21}$ are located on $P_{10}$ and their task rates are smaller than the task rates of $T_{16}$ and $T_{17}$ which are the other two tasks on $P_{10}$, as shown in Figure 5.2. As a result of the RMS scheduling [54], $T_{21}$ suffers the most significant deadline misses while $T_{19}$ also has a considerable deadline miss ratio. The performance results of DEUCON in this experiment are very close to EUCON's performance reported in [60].

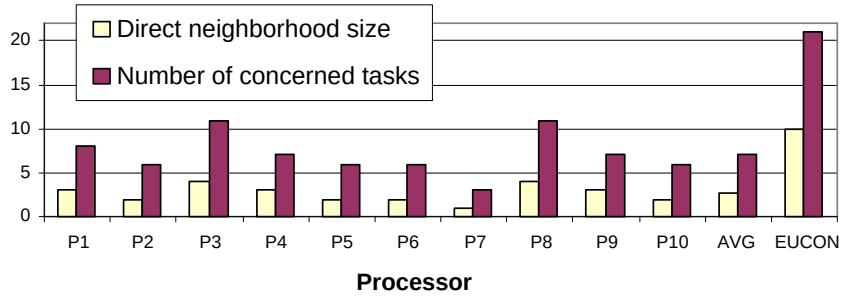



Figure 5.8: Entire system size vs. neighborhood size

### 5.3.3 Overhead

As discussed in Section 5.1, a major limitation of a centralized controller is that the run-time overhead is related to the size of the entire system. In contrast, the overhead of each local controller in DEUCON is just a function of its neighborhood size. Figure 5.8 compares the size of the entire system with the neighborhood size of each processor for the medium size workload. The centralized EUCON controller needs to model all the 10 processors and the 21 tasks in the system. In contrast, the average for DEUCON controllers is only 2.6 processors and 7.1 tasks, corresponding to a reduction by 74% and 66%, respectively.

To estimate the *average* computation overhead of the controllers, we measure the execution time of the least squares solver which dominates the computation cost on a 2GHz Pentium IV PC with 256MB RAM. In order to minimize the effect of the time delay caused by the IPC communication between the simulator and the MATLAB process, we use a single MATLAB command to run this least squares solver for 1000 times as a subroutine. The data shown in Figure 5.9 is the average of those 1000 runs. The average execution time of all controllers in DEUCON is only 62% of EUCON's centralized controller. We note that the speedup in execution times is not strictly polynomial in the numbers of neighbors and concerned tasks as one would expect from the theoretical complexity of MPC algorithms. This is attributed to difference between the *average* execution time of MATLAB's `lsqlin` solver and the *worst-case* computational complexity. In addition, the initialization cost in the optimization calculations is not negligible for relatively small scale problems in our workload.

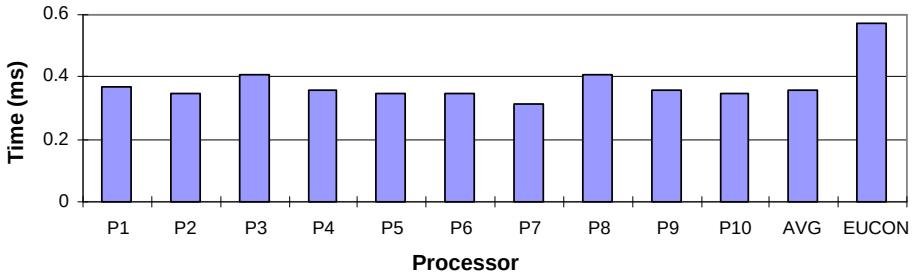



Figure 5.9: Controller execution time in MATLAB

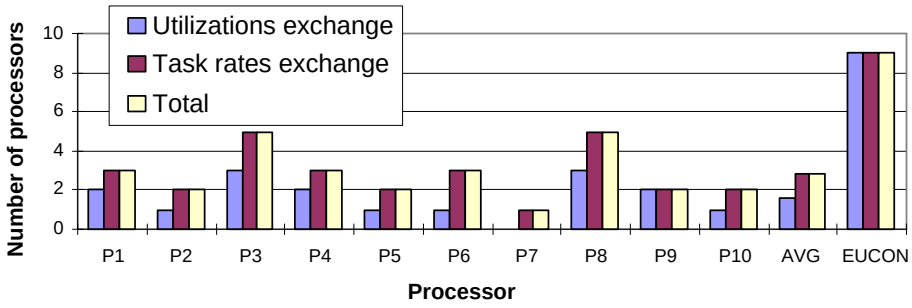



Figure 5.10: Estimated communication overhead

We now investigate DEUCON's communication overhead. As mentioned in Section 5.2, a controller's communication overhead is a function of the number of processors communicating with it[6]. To estimate communication overhead due to utilizations exchange, we count the number of processors from which a controller receives predicted utilizations. This is equal to the number of direct neighbors of the controller. To estimate communication overhead due to task rates exchange, we count the processors from which a controller receives the actual rate changes for one or more of its concerned tasks. The set of processors communicating with a controller is the union of these two processor sets. From Figure 5.10 we can see that DEUCON's average estimated per-controller communication overhead is 33% of the EUCON controller's communication overhead.

[6]Multiple data values (utilizations and/or rates) from a same processor can be easily combined to a single message in a real system implementation.

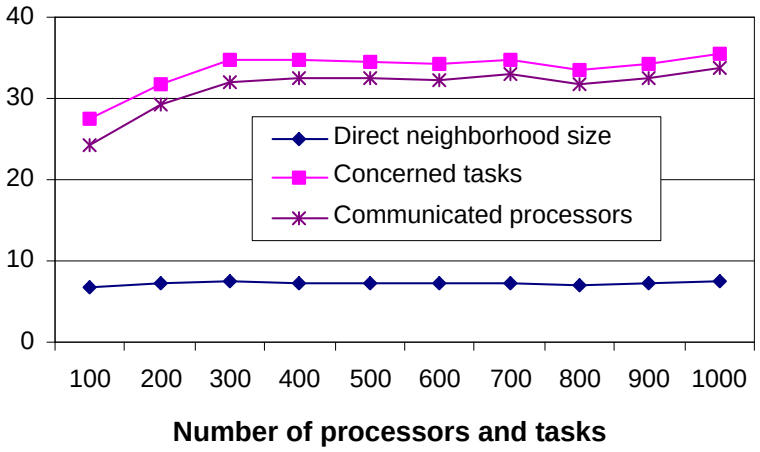



Figure 5.11: Per-controller overhead when tasks increase with processors

### 5.3.4 Scalability

Our final set of simulations evaluates the scalability of DEUCON in large systems. In all the following simulations, we employ randomly generated workloads. All subtasks are randomly allocated to processors such that every processor has the same number of subtasks. The number of subtasks per processor is fixed at 5 in all following simulations. To evaluate the scalability of DEUCON, we increase both the number of processors and the total number of subtasks in the systems proportionally.

Since the total number of subtasks is the product of the number of tasks and the number of subtasks per task, the system size can be varied in two ways.

- Case 1, we keep the number of subtasks per task fixed at 5 and increase both the number of tasks and the number of processors from 100 to 1000.
- Case 2, we keep the number of tasks fixed at 500 and then increase the number of subtasks per task from 1 to 10 and the number of processors from 100 to 1000.

Figure 5.11 shows the direct neighborhood size, the number of concerned tasks, and the number of communicated processors of a controller in case 1. Every result is the average value of all controllers in the system. We can see that the size of direct

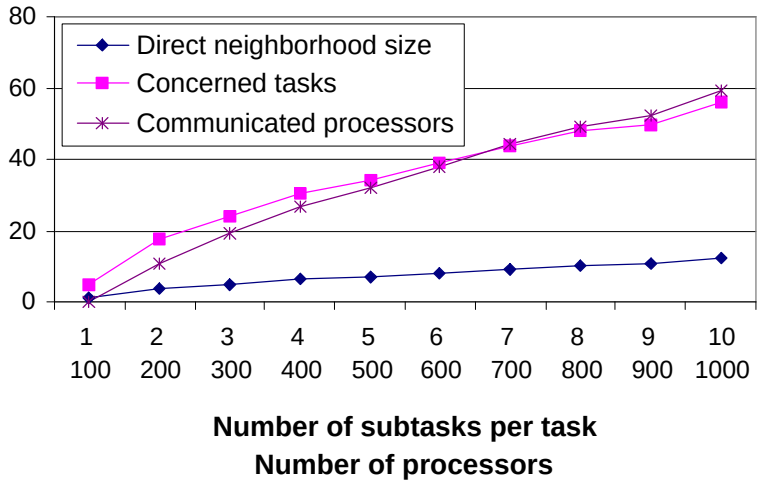



Figure 5.12: Per-controller overhead when subtasks increase with processors

neighborhood remains almost constant despite the ten-fold increase in the number of processors. At the same time, the numbers of concerned task and communicated processors increase very slowly. Even in the system with 1000 processors, a controller only communicates with fewer than 34 processors on average. These results demonstrate that the per-controller overhead of DEUCON is almost independent of the total size of the system when the number of subtasks per task remains fixed.

We then investigate case 2. Figure 5.12 shows that the three overhead metrics increase when the numbers of processors and subtasks increase. This is because, when each task has more subtasks, the number of processors in the control model of a controller also increases, resulting a larger neighborhood. However, our results show that the per-controller overhead remains moderate even when each task has a high number of subtasks. For example, a controller communicates with only 59.3 of 1000 processors on average even when each task has 10 subtasks. We note that in practice it is rare for a task to have an extremely large number of subtasks.

In addition to per-controller overhead, we are also interested in the *master ratio*, defined as the fraction of processors in the system that are master processors. The master ratio is important because only master processors have controllers. Figure 5.13 shows the fraction of master processors in the whole system in both cases. For a fixed number of subtasks per task (case 1), the master ratio remains fixed at around

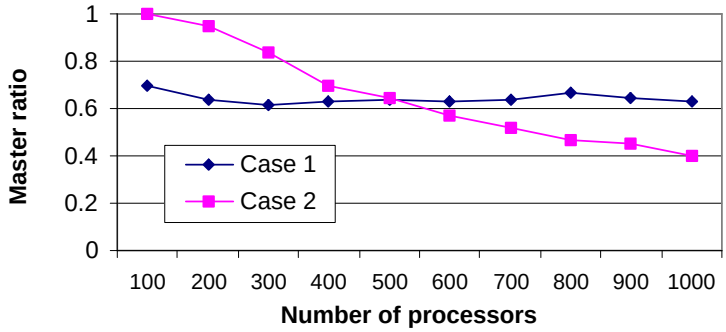



Figure 5.13: Fraction of master processors in both cases

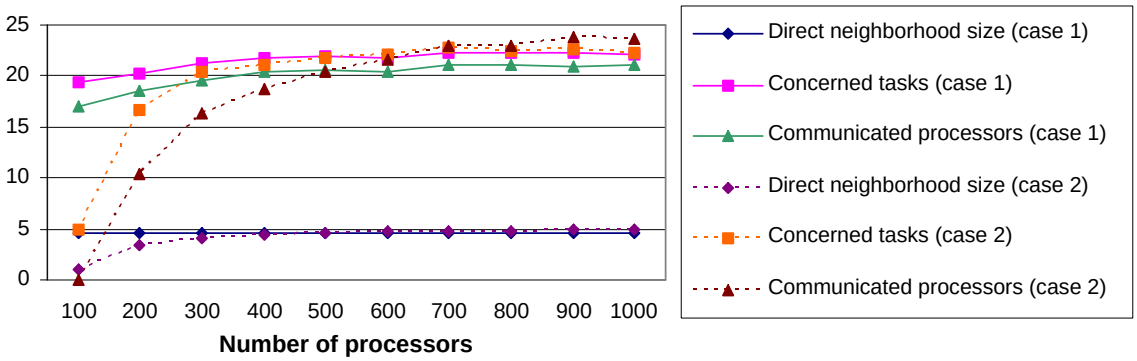



Figure 5.14: Average per-processor control overhead of the system in both cases

0.65, i.e., only 65% of the processors have controllers. On the other hand, the master ratio decreases while the number of subtasks per task increases (case 2).

We then evaluate the average *per-processor* control overhead of the system by multiplying the per-controller overhead present in Figure 5.11 and Figure 5.12 with the fraction of the master processors in Figure 5.13. Figure 5.14 shows that the average per-processor overhead converges to fixed levels in both cases as the size of the system increases. Note that in case 2 the increase in the *per-controller* overhead is compensated by the decrease in the master ratio. As a result, the per-processor overhead is almost independent of the system size for large systems. Therefore, DEUCON is highly scalable in large systems.

Moreover, we observe that real-world systems may allocate subtasks in a clustered fashion, i.e., all subtasks of a subsystem tend to share several processors and only a small number of tasks run across multiple subsystems. We expect such clustered

allocation to result in even smaller neighborhood size than the random allocation in our simulations.

## 5.4 Summary

We have presented the DEUCON algorithm for dynamically controlling the utilization of DRE systems. DEUCON features a novel decentralized control structure to handle the coupling among multiple processors due to end-to-end tasks. Both stability analysis and simulation results demonstrate that DEUCON achieves robust utilization guarantees even when task execution times deviate significantly from the estimation or changes dynamically at run-time. Furthermore, DEUCON can significantly improve the system scalability by distributing the computation and communication cost from a central processor to local controllers distributed in the whole system and tolerating network delays.

# Chapter 6

# FCS/nORB: Uniprocessor QoS Control Middleware

Object Request Broker (ORB) middleware has shown promise in meeting the functional and real-time performance requirements of distributed real-time and embedded (DRE) systems. However, existing real-time ORB middleware standards such as RT-CORBA do not adequately address the challenges of (1) providing robust performance guarantees portably across different platforms, and (2) managing unpredictable workload. To overcome this limitation, we have developed a QoS control middleware called FCS/nORB that integrates a single-processor Feedback Control real-time Scheduling (FCS) service with the nORB small-footprint real-time ORB designed for networked embedded systems. FCS/nORB serves as a foundation for us to develop end-to-end QoS control middleware for DRE systems.

The main feature of FCS/nORB is several feedback control loops that provide real-time performance guarantees by automatically adjusting the rate of remote method invocations transparently to an application. FCS/nORB thus enables real-time applications to be truly portable in terms of real-time performance as well as functionality, without the need for hand tuning. This chapter presents the design, implementation, and evaluation of FCS/nORB. Our extensive experiments on a Linux testbed demonstrate that QoS control middleware can provide deadline miss ratio and utilization guarantees in face of changes in the platform and task execution times, while introducing only a small amount of overhead.

## 6.1 Introduction

Object Request Broker (ORB) middleware [81][90] has shown promise in meeting the functional and real-time performance requirements of distributed real-time and embedded (DRE) systems built using common-off-the-shelf (COTS) hardware and software. DRE systems such as avionics mission computing [20], unmanned flight control systems [99], and autonomous aerial surveillance [56] increasingly rely on real-time ORB middleware to meet challenging requirements such as communication and processing timeliness among distributed application components.

Several kinds of middleware are emerging as fundamental building blocks for these kinds of systems. Low-level frameworks such as ACE [80] provide portability across different operating systems and hardware platforms. Resource management frameworks such as Kokyu [29] use low-level elements to configure scheduling and dispatching mechanisms in higher-level middleware. Real-Time ORBs such as TAO [81] and nORB [90] are geared toward providing predictable timing of end-to-end method invocations. ORB services such as the TAO Real-Time Event Service [33] and TAO Scheduling Service [29] offer higher-level services for managing functional and real-time properties of interactions between application components. Finally, higher-level middleware services [37][71][98][107] provide integration of real-time resource management in complex vertically layered DRE applications.

However, before it can fully deliver its promise, ORB middleware still faces two key challenges.

- *Provide real-time performance portability:* A key advantage of middleware is supporting portability across different OS and hardware platforms. However, although the *functionality* of applications running ORB middleware is readily portable, real-time *performance* can differ significantly across different platforms and ORBs. Consequently, an application that meets all of its timing constraints on a particular platform may violate the same constraints on another platform. Significant time and cost must then be incurred to test and re-tune an application for each platform on which it is deployed. Hence, DRE applications are not strictly portable even when developed using today's ORB middleware. The

lack of robust real-time performance portability thus detracts from the benefits of deploying DRE applications on current-generation ORB middleware.

- *Handle unpredictable workloads:* The task execution times and resource requirements of many DRE applications are unknown a *priori* or may vary significantly at run time - often because their executions are strongly influenced by the operating environment. For example, the execution time of a visual tracking task may vary dramatically as a function of the number and location of potential targets in a set of camera images sent to it.

A key reason that existing ORB middleware cannot deal with the above challenges is that common scheduling approaches are based on *open-loop* algorithms, e.g., Rate Monotonic Scheduling (RMS) or Earliest Deadline First (EDF) [54], which depend on accurate knowledge of task execution times to provide real-time performance guarantees. However, when workloads and available platform resources are variable or simply not known a priori, open-loop scheduling algorithms either result in extremely underutilized systems based on pessimistic worst-case estimation, or in systems that fail when workloads or platform characteristics vary significantly from design-time expectations.

As a foundation for developing adaptive ORB middleware that supports end-to-end QoS control, in this chapter, we first integrate a single-processor Feedback Control real-time Scheduling (FCS) framework [57] with real-time embedded ORB middleware [90], to provide portable real-time performance and robust handling of unpredictable workloads. FCS/nORB provides key scheduling support that makes DRE software performance (1) portable across OS and hardware platforms and (2) more robust against workload variations when tasks have negotiable QoS parameters that can be adjusted. The FCS service we have implemented in this work automatically adjusts the rates of method invocations on remote application objects, based on measured performance feedback. Our choice of this adaptation mechanism is motivated by the fact that in many DRE applications, e.g., digital feedback control loops [23][83], sensor data display [19], and video streaming [10], task rates can be adjusted on-line without causing instability or system failure. Other QoS adaptation mechanisms such as online task admission control can also be incorporated easily into the FCS/nORB service.

Specifically, this chapter makes three main contributions to research on DRE systems:

- Design documentation of a FCS service at the ORB middleware layer, that provides real-time performance portability and robust performance guarantees in face of workload variations,
- Implementation of a feedback control loop in an distributed ORB middleware that dynamically adjust the rates of remote method invocations transparently to the application (subject to application-specified constraints), and
- Results of empirical performance evaluations on a physical testbed that demonstrate the efficiency, robustness and limitations of applying FCS at the ORB middleware layer.

The rest of this chapter is structured as follows. We first briefly review previous work on FCS control algorithms in Section 6.2. Section 6.3 describes the design and implementation of our FCS service for nORB. We present results of our performance evaluation on a Linux testbed in Section 6.4. Finally, Section 6.5 summarizes the this chapter.

## 6.2 Feedback Control Real-time Scheduling

Recent research has shown that FCS algorithms can provide performance guarantees in terms of deadline miss ratios and CPU utilization even when actual task execution times are unknown or vary at run time. In this section, we describe our instantiations of three existing FCS algorithms in a nORB middleware service. Further details of the algorithms and their control analyses can be found in [57].

### 6.2.1 Task Model

We first describe the task model adopted by FCS. With ORB middleware, applications typically execute using method invocations on objects distributed across multiple endsystems. Invocation latency for remote methods includes latency on the client,

the server, and the communication network. Each method invocation may be subject to an end-to-end deadline. An established approach for handling timeliness of remote method invocations is through end-to-end scheduling [55]. In this approach, an end-to-end deadline is divided into intermediate deadlines on the server, client, and communication network, and the problem of meeting the end-to-end deadline is thus transformed into the problem of meeting every intermediate deadline.

In this chapter, we focus on the problem of meeting intermediate deadlines on a single processor, the server. We assume that the client and server are not collocated on the same processor. Such a configuration is common in networked digital control applications that run multiple control algorithms on a server processor that interacts with several other client processors attached to sensors and actuators. Communication delay is not the focus of this dissertation, although it is possible to treat a network similarly as a processor in an end-to-end scheduling model.

In the rest of this chapter, we use the term *task* to refer to the execution of a remote method on the server. We assume that each task $T_i$ has an *estimated* execution time $EE_i$ known at design time. However, the *actual* execution time of a task may be significantly different from $EE_i$ and may vary at run time. We also assume that the rate of $T_i$ can be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. Earlier research has shown that task rates in many real-time applications (e.g., digital feedback control [19], sensor data update, and multimedia [10][11]) can be adjusted without causing application failure. Specifically, each task $T_i$ is described by the following three attributes:

- $EE_i$: the *estimated* execution time,
- $[R_{min,i}, R_{max,i}]$: the range of acceptable rates, and
- $R_i(k)$: the rate in the $k^{th}$ sampling period.

We use $X(k)$ to represent the value of a variable $X$ in a sampling period $[(k-1)W, kW)$ seconds, where $k > 1$ and $W$ is the sampling period length. We assume all tasks are periodic, and each task $T_i$'s (relative) deadline on the server, $D_i(k)$, is proportional to its period.

A key property of our task model is that it does not require accurate knowledge of task execution times. The execution time of a task may be significantly different from its estimation and may vary at run time.

### 6.2.2 FCS Algorithms

The core of an FCS algorithm is a feedback control loop that periodically monitors and controls its *controlled variables* by adjusting QoS parameters (e.g., task rates or service levels). Candidate controlled variables include the total (CPU) utilization and the (deadline) miss ratio. The *utilization*, $U(k)$, is defined as the fraction of time when the CPU is busy in the $k^{th}$ sampling period. The *miss ratio*, $M(k)$, is the number of deadline missed divided by the total number of completed tasks[1] in the $k^{th}$ sampling period. *Performance references* represent the desired values of the controlled variables, i.e., the desired miss ratio Ms or the desired utilization Us. For example, a particular system may require a miss ratio $M_s = 1.5\%$ or a utilization $U_s = 70\%$. The goal of an FCS algorithm is to enforce the performance references specified by the application, via run-time QoS adaptation.

Three FCS algorithms have been developed based on the choice of different sets of these controlled variables. The FC-U and FC-M algorithms each control $U(k)$ or $M(k)$, respectively, and the FC-UM algorithm controls both $U(k)$ and $M(k)$ at the same time. The feedback control loop in each FCS algorithm is composed of one or more Monitors, a Controller, and one or more QoS Actuators. The Utilization and Miss Ratio Monitors measure the controlled variables, $U(k)$ and $M(k)$, respectively. At the end of each sampling period, the Controller compares the controlled variable with its corresponding performance reference ($U_s$ or $M_s$), and computes $B(k+1)$, the *total estimated utilization* for the subsequent sampling period. The QoS Actuators then adjust tasks' QoS parameters to enforce the total estimated utilization on the server. For example, a Rate Actuator assigns a new set of task rates such that i.e., $B(k+1) = \sum_i (EE_i * R_i(k+1))$, and instructs each client to adjust its invocation rate accordingly. Other examples of QoS actuation mechanisms include admission control and adaptation techniques based on the imprecise computation model [40].

[1]When a task has a *firm* deadline, it may be aborted when it misses its deadline. An aborted task is counted as a completed one and a deadline miss for miss ratio calculation.

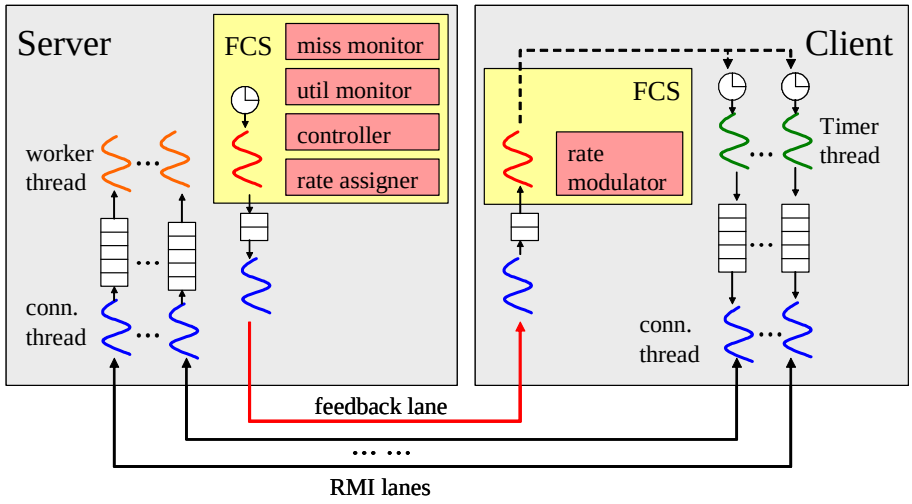



Figure 6.1: The Architecture of FCS/nORB

It is important to note that $B(k)$ may be different from $U(k)$ due to the difference between the estimated and actual task execution times. The details of the three FCS algorithms are available in [57].

## 6.3 FCS/nORB Architecture

In this section, we present the architecture of an FCS service that instantiates the FCS algorithms described in [57] atop a real-time embedded middleware ORB called nORB, as illustrated in Figure 6.1. We first give an overview of our extensions to nORB for use with FCS, and then describe the design and implementation of the FCS service.

### 6.3.1 Extensions to nORB for FCS

nORB [90] is a light-weight real-time ORB designed to support networked embedded systems. Both nORB and the new FCS service are based on ACE [79]. The current implementation of nORB, to which we applied our FCS extensions, only supports fixed priority scheduling. To avoid priority inversion at the communication layer, a

separate TCP connection called a *lane* [74] is established between a server and a client for each priority level that is used for method invocation requests. Further details of nORB are presented in [90].

While FIFO queuing in each static priority lane is sufficient for many applications, to support the on-line adaptation of task rates needed by FCS we had to extend the basic nORB queuing capabilities, as is shown in Figure 6.1. An FCS/nORB client has a number of timer threads and connection threads. Each pair of timer/connection threads (connected through a buffer) is assigned a priority and submits method invocation requests to the server at this priority. Each timer thread is associated with a timer that generates periodic timeouts, to initiate method invocation requests at a specified rate. A basic FCS/nORB server has several pairs of worker and connection threads. Each pair of worker/connection threads is assigned a priority and is responsible for processing method invocation requests at that priority. Connection threads receive method invocation requests from clients, and worker threads invoke the corresponding methods and send the results back to clients. We apply the RMS policy [54] to assign task priorities to the thread pairs on the server. Each thread pair on the client shares a same priority as the thread pair on the server that it is connected to. A key contribution of this work is to show that FCS services can be realized in reduced-feature-set ORBs such as nORB that are tailored to fit within the space and power limitations seen in many networked embedded systems, without sacrificing real-time performance.

### 6.3.2 Configuration Interface

Application developers can specify a set of scheduling parameters in a configuration file that is used to initialize the FCS service when the system is started. Configuration parameters include the specific FCS algorithm to run, the performance references, the sampling period, and two parameters, $G_A$ and $G_M$. Based on the control analysis in [57], FCS/nORB determines the value of the control parameters based on $G_A$ and $G_M$ in order to achieve the desired control performance. In general, higher $G_A$ and $G_M$ increase the range of platform and workload variability that FCS can handle. Detailed analysis of $G_A$ and $G_M$ is available in [57]. Applications can register their tasks in

a task description file. Each task $T_i$ is described by a tuple $(EE_i, R_{min,i}, R_{max,i})$ as described in Section 6.2.

### 6.3.3 Feedback Control Loop

The FCS service on a server includes a *utilization monitor*, a *miss ratio monitor*, a *controller*, a *rate allocator*, and a pair of FCS/connection threads. The FCS service on a client includes a *rate modulator* and a pair of FCS/connection threads. All FCS/connection threads in the FCS service are assigned the highest priority so that the feedback control loop can run in overload conditions, when it is needed most. The FCS/connection threads on the server are connected with each client connection thread through a TCP connection we call a *feedback lane*. We now present the details of each component.

**Utilization Monitor:** The utilization monitor uses the /proc/stat file in Linux to estimate the CPU utilization in each sampling period. The /proc/stat file records the number of jiffies (each 1/100 of a second) since the system start time, when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle task. At the end of each sampling period, the utilization monitor reads the counters, and estimates CPU utilization by dividing the number of *jiffies* used by the idle task in the last sampling period by the total number of jiffies in the same period. We note that the same technique is used by the benchmarking tool, NetPerf [39].

**Deadline Miss Monitor:** The deadline miss monitor measures the percentage of completed tasks that miss their deadlines on the server in each sampling period. FCS/nORB maintains two counters for each pair of connection/worker threads on the server. One counter records the number of completed tasks in the current sampling period, and the other records the number of tasks that missed their deadlines in the same period. Each connection thread timestamps every method invocation request when it arrives from its nORB lane. The worker thread checks whether a completed task has missed its deadline and updates the counters after it sends the invocation result to the client. At the end of each sampling period, the deadline miss monitor

aggregates the counters of all worker/connection threads, and computes the deadline miss ratio in the sampling period. Note that FCS/nORB maintains separate counters for each pair of connection/worker threads instead of shared global counters, to reduce contention among threads updating the counters. This use of thread-specific storage is important because contention among worker threads could either allow priority inversions or introduce unnecessary overhead to prevent them.

**Controller:** The controller implements the control function presented in Section 2.2. Each time its periodically scheduled timer fires, it invokes the utilization and/or deadline miss monitors, computes the total estimated utilization for the next sampling period, and then invokes the rate assigner.

**Rate Assigner:** The rate assigner on the server and the rate modulator on its clients together serve as actuators in the feedback control loop. The rate assigner computes the new task rates to enforce the total estimated utilization computed by the controller. Different policies can be applied to assign task rates. Our rate assigner currently implements a simple policy that is called *Proportional Rate Adjustment* (PRA) in this chapter. Assuming that the initial rate of task $T_i$ is $R_i(0)$, the initial total estimated utilization $B(0) = \sum_i (EE_i R_i(0))$, and the total estimated utilization for the following $k_{th}$ sampling period is $B(k)$, the PRA policy assigns the new rate to task $T_i$ as follows: $R_i(k) = (B(k)/B(0))R_i(0)$. If $R_i(k)$ falls outside its acceptable range [R imin ,R imax], it is rounded to the closer limit. It can be easily proven that PRA enforces the total estimated utilization, i.e., $B(k) = \sum_i (EE_i R_i(k))$, if no task rates reach their lower or upper limits.

The PRA policy treats all the tasks "fairly" in the sense that the relative rates among tasks always remain the same if no tasks reach their rate limits. When an application runs on a faster platform, the rates of all tasks will be *increased* proportionally, while on a slower platform, the rates of all tasks will be *decreased* proportionally. A side effect of the PRA policy is that priorities of tasks will not change at run-time under RMS because the relative order of task rates remains the same. This reduces overhead on the clients because they do not need to change task deadlines on the fly. However, since PRA potentially changes the rate of every task in each sampling

period, it may introduce relatively high overhead for resetting all the timers on the clients. Fortunately, as shown in our measurement in Section 4.5.3, such overhead is small when ACE timers are used.

Note that the PRA policy is based on the assumption that all tasks are "equally important". More precisely, it assumes that all tasks' *values* to the application are uniformly proportional to their execution times. When this assumption is not true, the rate assigner needs to optimize the total system value under the constraint of the total estimated utilization. Although the value optimization problem is not a focus of this study, existing optimization algorithms, e.g., [49], could be used in the rate assigner to address this problem.

**Rate Modulator:** A Rate Modulator is located on each client. It receives the new rates for its remote method invocation requests from the server-side rate assigner through the feedback lane, and resets the interval of the timer threads whose request rates have been changed.

### 6.3.4 Implementation

FCS/nORB 1.0 is implemented in C++ using ACE 5.2.7 on Linux. The entire FCS/nORB middleware (excluding the code in the ACE library and IDL compiler library) is implemented in 7898 lines of C++ code - compared to 4586 lines of code in the original nORB. Both nORB and FCS/nORB are open-source software and can be downloaded from

- nORB: `http://deuce.doc.wustl.edu/nORB/`
- FCS/nORB: `http://deuce.doc.wustl.edu/FCS_nORB/`

## 6.4 Empirical Evaluations

In this section, we present the results of four sets of experiments we ran on a Linux testbed. Experiment I evaluated the performance portability of applications on FCS/nORB on two different server platforms. On both platforms, we ran the same synthetic workload for which the actual task execution times significantly deviate from their estimated execution times (the same estimates were used in all experiments). Experiment II stress-tested FCS/nORB's ability to provide robust performance guarantees with a workload whose task execution times varied dramatically at run-time. Experiment III adopted an image matching workload that is representative of target location applications, to re-examine FCS/nORB's robust performance guarantees in realistic environment. Finally, Experiment IV measured the overhead introduced by FCS from three different perspectives.

### 6.4.1 Experimental Set-up

#### Platform

We performed our experiments on three PCs named **Server A**, **Server B**, and **Client**. Server A and Client were Dell 1.8GHz Celeron PCs, each with 512 MB of RAM. Server A and Client were directly connected with a 100 Mbps crossover Ethernet cable. They both ran Red Hat Linux release 7.3 (Kernel 2.4.19). Server B was a Dell 2GHz Pentium4 PC with 256 MB of RAM. Server B and Client were connected through our departmental 100 Mbps LAN. Server B ran Red Hat Linux release 7.3 (Kernel 2.4.18). Server A and Server B served as servers in separate experiments, while Client served as the only client host in all experiments.

#### Workload

To evaluate the robustness of FCS/nORB, we used both a synthetic workload and a more realistic one that simulated real applications in our experiments. Since we focused on unpredictable workload and platform portability, the estimated execution

Table 6.1: Methods invoked by the workload

| method | est. execution time (ms) | min rate | max rate | number of invoking tasks |
|---|---|---|---|---|
| 1 | 8.4 | [1.1,2.1] | 35 | 6 |
| 2 | 1.2 | [1.3,1.9] | 50 | 2 |
| 3 | 7.0 | [1.2,2.2] | 40 | 4 |

times were different from the actual execution times in all experiments. The same estimated execution times were used in all experiments despite the fact that they used different platforms and as a result had different actual task execution times. With FCS, re-profiling of task execution times was *not* needed to provide performance guarantees.

The **synthetic workload** comprised 12 tasks. Each task periodically invoked one of three methods (shown in Table 6.1) of an application object. All the tasks invoking the same method shared the same maximum rate, but their minimum rates were randomly chosen from a range listed in the "min rate" column in Table 6.1.

The **realistic workload** comprised an avionic task set and an additional target location task. The avionics task set is based on an F-16 simulator presented in [1]. It includes four separate tasks (guide, control, slow navigation and fast navigation) with different rate ranges and execution times as shown in Table 6.2. We chose these tasks in our workload because their rate ranges are available [1][2].

The additional target location task is included because of its relatively computing intensity and its potential execution time variation in the runtime. Those two properties make the simulated avionic system suffer a runtime performance variation, which provides a typical platform for FCS to apply.

A common solution for target location includes a series of steps including image restoration and enhancement, geometric correction, image matching, etc. For the sake of simplicity, we implemented only the most critical step, image matching [30], in our experiment. The goal of image matching is to search *input images* (periodically captured by camera equipments) for a target, which is represented by another smaller

[2]We acknowledge that FCS may not be directly applicable to safety-critical real-time systems such as flight control for manned aircraft, which require hard performance guarantees.

Table 6.2: Task sets in real image matching workload

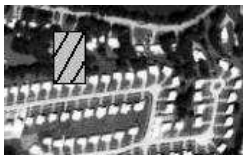
| Task | est. execution time (ms) | min rate | max rate |
|---|---|---|---|
| Guide | 100 | 0.2 | 1.0 |
| Control | 80 | 1.0 | 5.0 |
| Slow navigation | 100 | 0.2 | 1.0 |
| Fast navigation | 60 | 1.0 | 5.0 |
| Target location | 150 | 0.2 | 5.0 |

sized template image. Specifically, every pixel in the input image is potentially part of where the target is located so they all are *candidate points*. All those candidate points will be checked exhaustively for their similarity values with the target template (some advanced image matching algorithms only check a subset of all pixel positions). At each of the candidate points, a *candidate region* with the same size as target template is extracted from the input image to compare with target template pixel by pixel. All the individual similarity values from each of the corresponding pixel pairs are summed up as an overall similarity value for this candidate point. The candidate point with largest similarity will be identified as the *match place*, so long as its similarity value is larger than a pre-defined threshold. A target is considered to have been located when its match place is found. In our experiment, Absolute Difference (AD) [70] is used to compute the similarity.

The application scenario for our experiment is as follows. Before the target object of interest is located, the image matching task searches the full input image for a match with the template. After an object is found at a particular location, a *focus region* is shrunk from the full image to a small region that is centered at the known location of the object in subsequent images to save CPU cycles for other tasks. However, in some cases a fast moving object may escape from the focus region between two consecutive invocations of the task, resulting in the loss of the object template. In this situation, the full image must be searched again to relocate the target. Therefore, in our scenario the execution time for the image matching task starts at a high level in the beginning, then drops to a low level when the target has been detected. After this target is lost, the execution time returns again to its initial high level. The variation in the execution time is *unknown a priori* because it depends whether the target is being found on the input image or the focus region.

(a) Input image

(b) Target template



(c) Focus region

Figure 6.2: Images used in Experiment III

Figure 6.2 shows those images used in our experiment. Since the execution time of the exhaustive image matching with AD algorithm depends only on the sizes of those images and is insensitive to their contents, a same input image (Figure 6.2a) can be used in every invocation of the image matching task without affecting the task workload, which is the main concern of FCS/nORB. Similarly, a same focus region (Figure 6.2c) can also be used. The switch between the full input image and the focus region is forced to simulate the target capture and loss. In the sequence described in above scenario, we first use the full input image to search for the target template. At a certain time the target is found and we then start to search the focus region image for a while. Finally we change back to the input image by assuming the target is lost from the focus region. Although the images used in our experiments are simplified compared to real world scenarios, they are sufficient for the purpose of causing realistic variations in the task execution time.

**Control Configuration**

The configuration parameters for FCS are shown in Table 6.3. To demonstrate the robustness of feedback control, the *same* configuration was used in all experiments even though they were performed on different platforms and tested with different workloads and execution times. The Controller parameters were computed using control theory based on $G_A$ and $G_M$, which determine the robustness of FCS [57]. The utilization reference of FC-U is chosen to be 70%, slightly lower than the RMS schedulable utilization bound for 12 tasks: $12(2^{1/12} - 1) = 71\%$. FC-UM had a higher utilization reference (75%) because it uses miss ratio control as we discussed in Section 4. The sampling period used in Experiment I and Experiment II is 4 seconds. Since in the real image matching workload task 1 and task 3 both have 5 seconds as their maximum period, we set the sampling period in Experiment III to 10 seconds to decrease the sampling jitter caused by rate tuning. As a baseline, we also ran these experiments under open-loop scheduling (RMS) by turning off the feedback loop. For simplicity, the open-loop baseline is called **OPEN** in the rest of the chapter.

Table 6.3: Control configuration in all experiments

| | FC-U | FC-M | FC-UM |
|---|---|---|---|
| reference | $U_s$=70% | $M_s$=1.5% | $M_s$=1.5%<br>$U_s$=75% |
| $G_A, G_M$ | $G_A$=2 | $G_A$=2, $G_M$=0.447 | |
| sampling period | 4 seconds (10 seconds in Experiment III) | | |

### 6.4.2 Experiment I: Performance Portability

In Experiment I, the execution time of each task on Server A remained approximately twice its estimated value throughout each run. The purpose of this set of experiments was to evaluate the performance of the FCS algorithms and OPEN when task execution times vary significantly from estimated values, either due to the difference between a new deployment platform and the original platform on which the tasks were profiled, or to significant inaccuracy in task profiling.

Our first experiment emulates common engineering practice based on open loop scheduling. We first tuned task rates based on the estimated execution times so that the total *estimated* utilization was 70%. However, when we ran the tasks at the rates according to the predicted rates, the server locked up. This is not surprising: since the estimated execution times were inaccurate, the actual total requested utilization by all nORB threads reached approximately 140%. This caused the Linux kernel to freeze because all nORB threads were run at real-time scheduling priorities that are higher than kernel priorities on Linux. When the CPU utilization requested by nORB threads reached 100%, no kernel activities were able to execute. To avoid this problem using common real-time engineering techniques, all the tasks would need to be re-profiled for each platform on which the application is deployed. Hence, the open loop approach can cost developers significant time to tune the workload to achieve the same performance on different platforms. This lack of performance portability is an especially serious problem when there is a large number of potential platforms (e.g., in a product line) or if a potential platform is unknown at system development time.

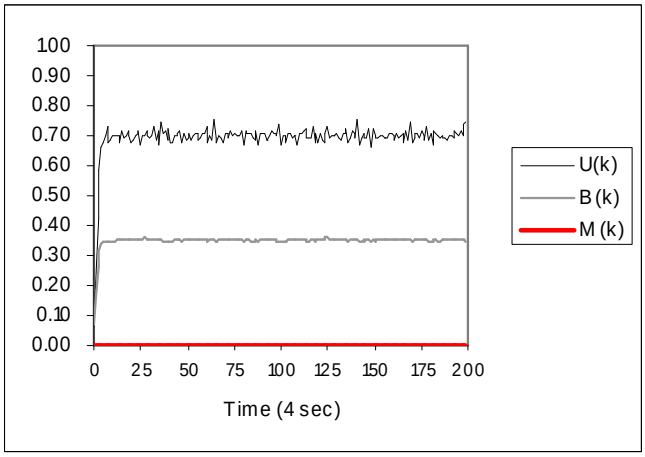



Figure 6.3: A typical run of FC-U on Server A

We now examine the experimental results for the FCS algorithms themselves. As an example, Figure 6.3 illustrates the sampled utilization $U(k)$, the miss ratio $M(k)$, and the total estimated utilization $B(k)$ computed by the controller in a typical run under FC-U. All tasks started from their lowest rates. The feedback control loop rapidly increased $U(k)$ by raising task rates (proportional to $B(k)$). At the 5th sampling point, the $U(k)$ reached 67.7% and settled in a steady state around 70%. This result shows that FC-U can self-tune task rates to achieve the specified CPU utilization even when task execution times were significantly different from estimated values. The results are consistent with the control analysis presented in [57].

The performance results for FC-U, FC-M, and FC-UM on Server A are summarized in Figure 6.4(a-c). The performance metrics we used included the miss ratio and utilization in steady state, and the settling time. The *steady-state miss ratio* is defined as the average miss ratio in a steady state. The *steady-state utilization* is similarly defined as the average utilization in a steady state. Both metrics measure the performance of a system after its adaptation process settles down to a steady state. Settling time represents the time it takes the system to settle down to a steady state. The settling time can also be viewed as the duration of the self-tuning period after an application is ported to a new platform. It is usually difficult to determine the precise settling time on a noisy, real system. As an approximation, we considered that FC-U and FC-M entered a steady state at the first sampling instant when U(k) reached 0.99Us, and FC-M entered a steady state at the first sampling instant when

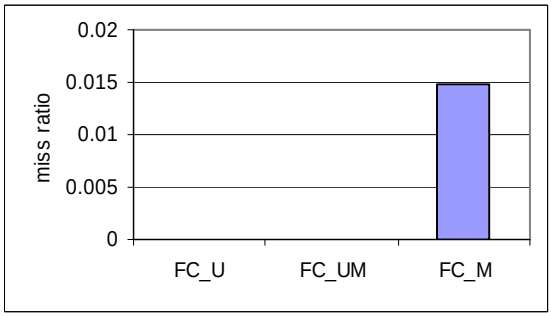
U(k) reached 0.99Us in the last sampling period of the experiment. Each data point in Figure 6.4(a-c) is the mean of three repeated runs, and each run took 800 seconds. The standard deviations in miss ratio, utilization, and settling time are below 0.01%, 0.03%, and 6.11 seconds (i.e., a 1.53 sampling period), respectively.

From Figure 6.4(a), we can see that both FC-U and FC-UM caused no deadline misses in steady-states. FC-M's steady-state miss ratio is 1.49%, compared to the miss ratio reference of 1.5%. At the same time, the steady-state utilizations of FC-U and FC-UM are respectively 70.01% and 74.97%, compared to respective utilization references of 70.00% and 75%. The result for FC-UM occurred because the utilization control dominated in steady state due to the fact that its steady state utilization is lower than the miss ratio control. In contrast, FC-M achieved a higher utilization (98.93%) in the steady state at the cost of a slightly higher miss ratio.

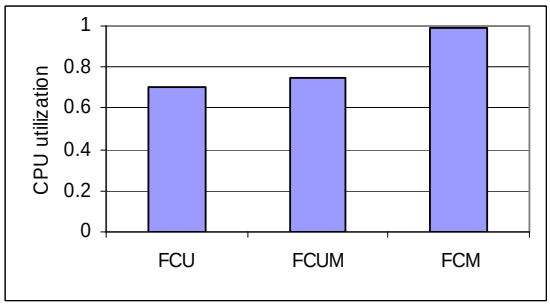
As shown in Figure 6.4(c), FC-M and FC-UM both had significantly longer settling times than FC-U due to the saturation of miss ratio control in underutilization. This means that FC-M and FC-UM need more self-tuning time before they can reach steady states. Note that the settling times of FC-M and FC-UM are related to the initial task rates. In our experiments, all tasks started from their lowest possible rates in the beginning of the self-tuning phase. The settling times can be reduced by setting the initial task rates closer to the desired rates. For example, we may choose the initial rates to be the same as the desired rates on the slowest platform in a product line.

To further evaluate the performance portability of FCS/nORB, we re-ran the same experiments on Server B. Typical runs of FC-U, FC-UM, and FC-M are shown in Figures 6.4, 6.5, and 6.6, respectively. Each run takes 1200 seconds. As was the case on Server A, all the algorithms successfully enforced their utilization and/or miss ratio references in steady state. The difference is that all tasks ran at a higher rate (proportionally to $B(k)$) on Server B than Server A because Server B is faster than Server A. In addition, all algorithms had longer settling times on Server B than Server A. This is consistent with our control analyses in [27].

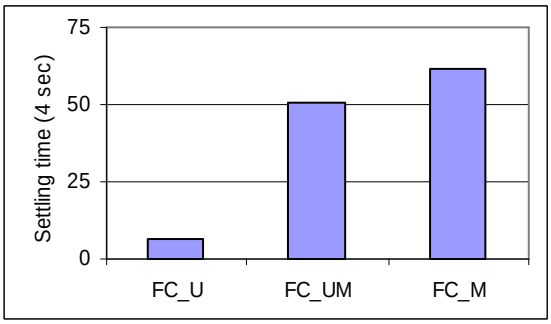
In summary, Experiment I demonstrated that FCS/nORB can provide a desired utilization or miss ratio even when 1) applications were ported to different platforms and




(a) Average steady-state miss ratio




(b) Average steady-state CPU utilization




(c) Average settling time

Figure 6.4: Performance results of FCS algorithms on Server A in Experiment II

2) task execution times were significantly different from their estimations. Therefore, FCS/nORB represents a way to perform automatic performance tuning on a new platform.

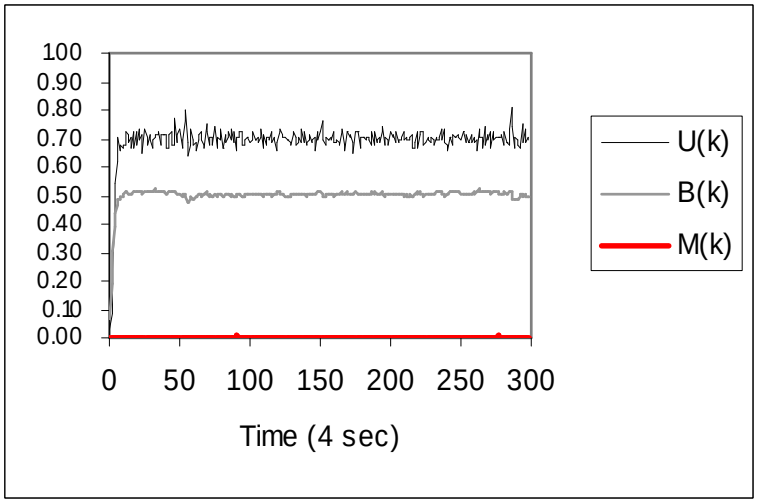



Figure 6.5: A typical run of FC-U on Server B

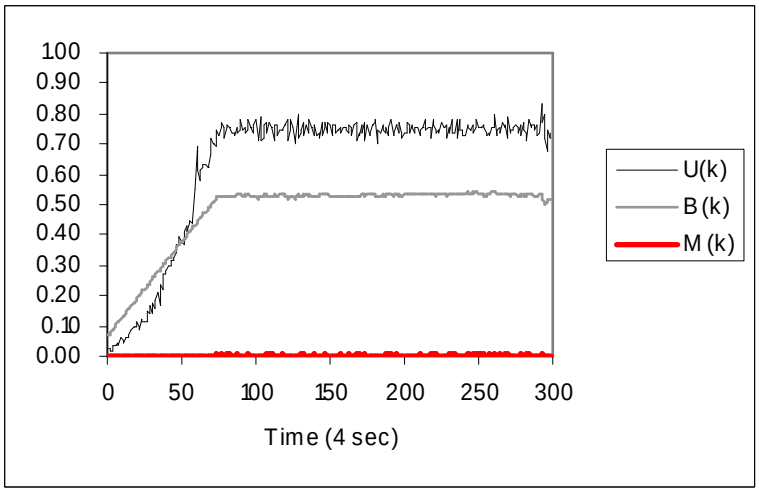



Figure 6.6: A typical run of FC-UM on Server B

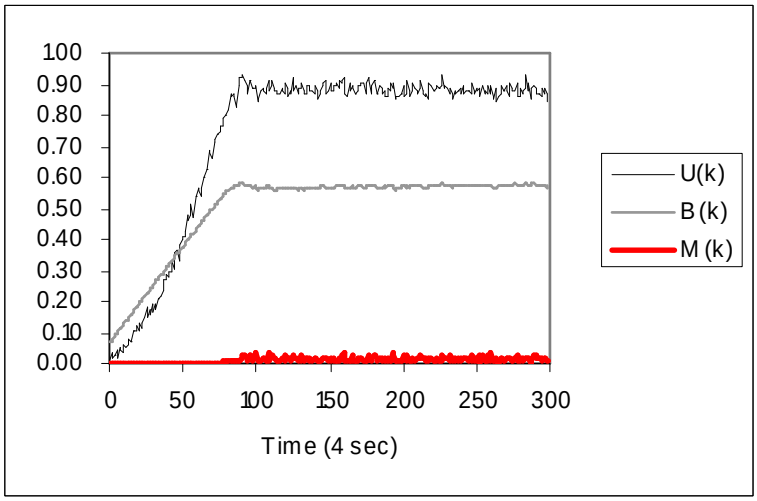
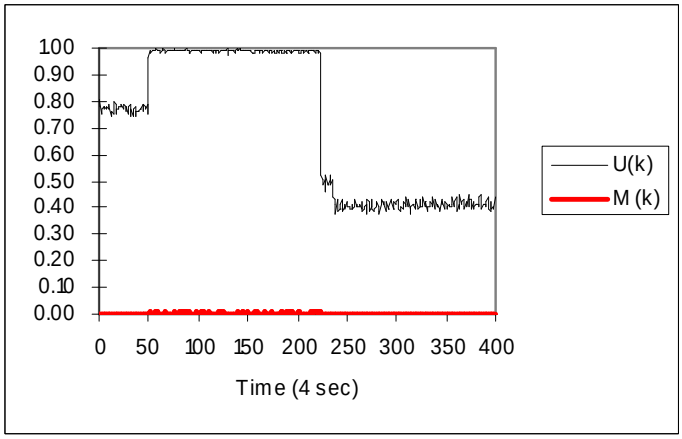



Figure 6.7: A typical run of FC-M on Server B

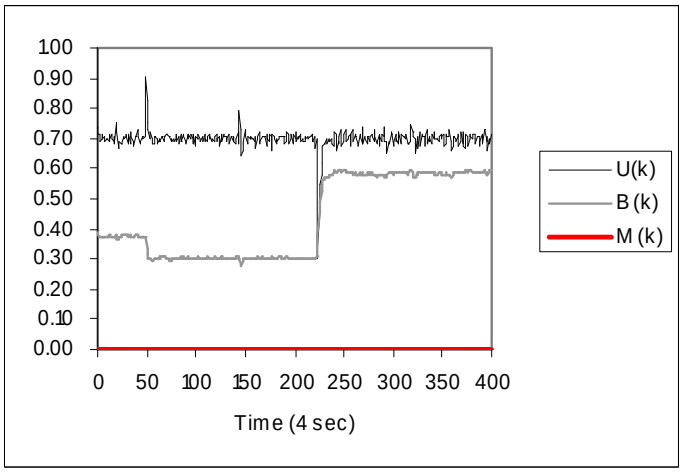
In addition, we note that a combination of FCS and open-loop scheduling can be used to achieve both self-tuning and run-time efficiency for applications with steady workloads. When an application is ported to a new platform, it can be scheduled initially using the FCS algorithm to converge to a steady state with desired performance. Then the feedback control loop can be turned off and the applications can continue to run at the correct rates under open-loop scheduling.

### 6.4.3 Experiment II: Varying Synthetic Workload

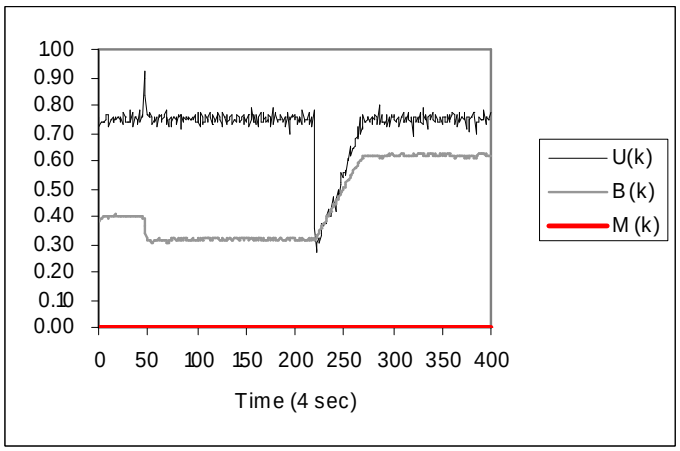
In this set of experiments, we evaluated the performance of FCS/nORB and OPEN on Server A when task execution times vary significantly at run-time. We first study the performance of OPEN. A typical run of OPEN is illustrated in Figure 6.8(a). We initially hand-tuned the task rates to achieve a utilization of approximately 75%. In the beginning of the 50th sampling period (200 sec), however, the execution times of method 1 (invoked by 6 tasks - see Table 6.1) was suddenly increased, which caused the utilization to reach almost 100% and resulted in deadline misses. Note that the system kernel would have frozen (as in Experiment II) had the execution time of method 1 been increased even slightly more. At 900 seconds, the execution time of method 1 was suddenly decreased causing the utilization to drop to approximately 40%. This experiment shows that even fine-tuned applications can not always achieve




(a) OPEN




(a) FC-U




(b) FC-UM

Figure 6.8: Utilization and deadline miss ratio under varying workload

acceptable performance under OPEN. When the actual execution times exceed the initial execution time used for tuning, the system can be overloaded and may even lock up. On the other hand, if the actual execution times become lower than those used at tuning time, the CPU is underutilized when task could have run at a higher rate and thus shown improved QoS.

In contrast, both FC-U and FC-UM maintained specified CPU utilizations (70% and 75%, respectively) in steady states despite the variations in task execution times (as illustrated in Figure 6.8(b) and 6.8(c), respectively). Both algorithms effectively adapted task rates (proportionally to $B(k)$) in response to changes in system load. FC-UM had a long settling time in the underutilized condition. However, its settling time is significantly shorter in the overload condition. The short settling time under overload is important because adaptation is much more important in overload conditions than in underload conditions.

Interestingly, FC-M caused the system to lock up when the execution times increased. This is because FC-M achieved a high utilization (more than 90%) before the execution time increased at time 200 sec. The utilization then increased to 100% due to the increase in execution times, and the system again locked up due to kernel starvation. In contrast, previous simulation results [57] showed that FC-M could handle such varying workload because the impact of CPU over-utilization by the middleware on kernel activities was not modeled in the simulator, which was design to simulator a scheduler in the OS kernel.

In general, FCS/nORB cannot handle varying workload that even transiently increases the utilization to 100% due to the starvation of the kernel under such conditions. This result shows a limitation of middleware implementations on top of common general purpose operating systems (e.g., Linux, Windows, and Solaris) in which real-time scheduling priorities are higher than kernel priorities. On such platforms, the range of variation in utilization that the FCS algorithms can handle is limited by its steady-state utilization before the variation occurs. For example, with a utilization reference of $U_s$, FC-U can only handle a utilization increase of no more than $(1 - U_s)$ in order to provide robust utilization guarantees. Therefore, the utilization reference of FC-U and FC-UM should consider this *safety margin* in the face

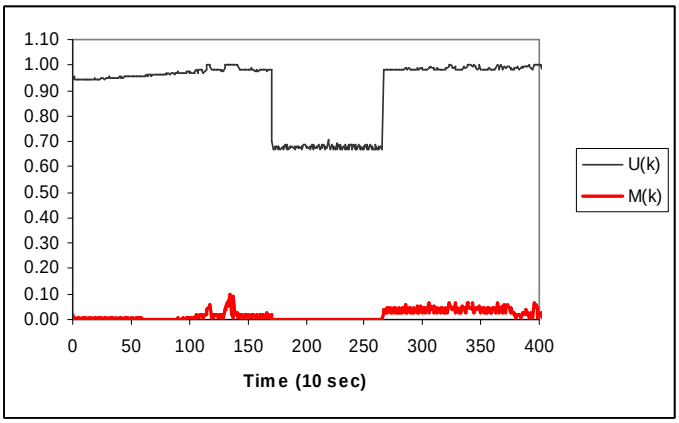
of varying workload. Since FC-M usually achieves a high utilization and, more importantly, does not have control over its safety margin, a middleware implementation of FC-M is less appropriate for time varying workloads.

### 6.4.4 Experiment III: Varying Realistic Workload

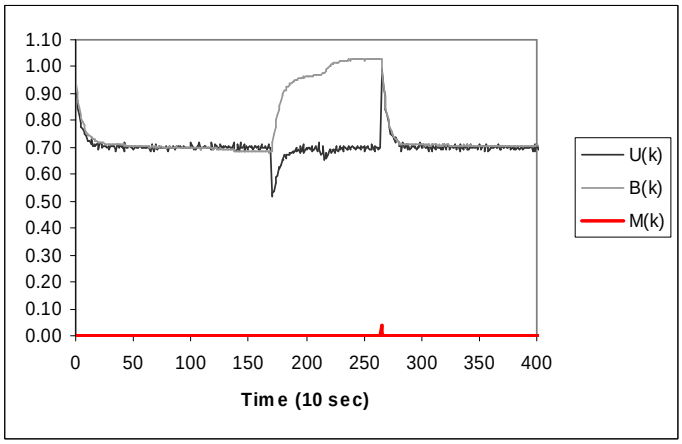
In Experiment III, we reexamined FCS/nORB's performance with the realistic workload in which the execution time of the target location task vary dynamically.

Figure 6.9(a) shows a typical run of OPEN. In the beginning of the run, the target location task had a long execution time while it searched the whole input images for the interested object. Consequently, the CPU utilization was close to 95% and a number of task invocations missed their deadlines. At around 160th sampling period, the target was assumed to have been found, so the focus region was shrunk to locate the target. CPU utilization dropped significantly as we can observe in Figure 6.9. This drop switched the system from an overloaded to an underutilized status. We continued by assuming the target escaped from the focus region at around the 265th sampling period, so the execution time of the target location task then returned to its original level. At that point utilization again returned to an overload condition. Hence in this scenario, the OPEN system just switched back and forth between overload with deadline misses, and underutilization with unnecessarily low task rates, neither of which leads to satisfactory performance.

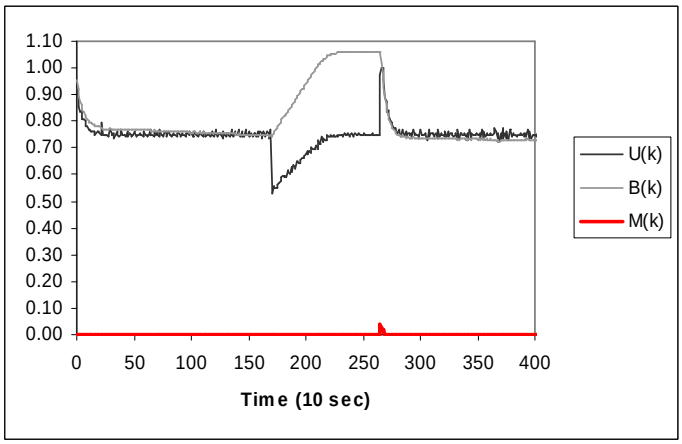
Figure 6.9(b) and Figure 6.9(c) show that both FC-U and FC-UM maintained specified utilization levels in their steady states, which was over most of the entire run. The performance of FC-U is illustrated in Figure 6.9(b). The CPU utilization was decreased to the set point (70%) at the 15th period, so deadline misses were avoided. At around 160th period, when the system found the object, FC-U drove the utilization back up to the set point by increasing the rates of all current tasks to utilize the CPU better. The faster rates also improved the system utility. Particularly for our image matching task, more frequent invocation of the image matching task improves tracking precision while reducing the chance that the tracked object may escape from the window image. At the 165th sampling period, when the target did escape from the focus region, the full image was searched again to relocate the target. FC-U only had




(a) OPEN




(a) FC-U




(b) FC-UM

Figure 6.9: Utilization and deadline miss ratio under realistic workload

a very transient spike of deadline misses, which is highly preferable compared with OPEN. The performance of FC-UM shown in Figure 6.9(c) had similar results to FC-U. The only difference is that the settling time was longer when the system recovered from underutilized status, for the same reasons explained in previous section.

Both Experiments II and III demonstrated that FC-U and FC-UM can provide robust performance guarantees, even when task execution times vary (within the aforementioned safety margin) at run-time.

### 6.4.5 Experiment IV: Overhead Measurement

The feedback control loop for each FCS algorithm introduces overhead. This overhead is caused by several factors including the timer associated with FCS, the cost of utilization and miss ratio monitoring, the control computation in the controller, and the rate calculation and communication overhead in the rate assigner. FCS/nORB is a viable middleware only if the overhead it introduces is sufficiently low.

#### Coarse-grained overhead measurement

To quantify the overhead imposed by the FCS algorithms, we compared the average CPU utilization under different scheduling algorithms when the same workload is applied to the system running on Server A. To limit the overhead caused by the utilization monitoring for OPEN and FC-M, average CPU utilizations were measured by setting the sampling period of the utilization monitor to the duration of the entire run, i.e., the utilization monitor is only invoked twice for each run with FC-M and OPEN - once at the beginning of the run, and once at the end of the run. The average CPU utilization of FC-U and FC-UM was measured by averaging the utilization of each sampling period, since they need to execute the utilization monitor periodically. To keep the application workload constant, we disabled the rate modulator on the clients so that all tasks always ran at constant rates.

The results of the overhead measurements are summarized in Table 6.4. The first row shows the mean of the average utilizations in 8 repeated runs, along with its

Table 6.4: Results of coarsed-grained overhead measurement

| | OPEN | FC-U | FC-M | FC-UM |
|---|---|---|---|---|
| utilization (%) | 74.15 ±0.30 | 74.55 ±0.42 | 74.70 ±0.10 | 75.05 ±0.16 |
| overhead (%) | | 0.40 | 0.54 | 0.90 |

90% confidence interval. Each run lasted for 800 seconds, a total of 200 sampling periods. The second row shows the overhead of each FCS algorithm in terms of CPU utilization, which is computed by subtracting OPEN's utilization from each FCS algorithm's utilization.

The 90% confidence interval of the most efficient algorithm, FC-U, actually overlapped with that of OPEN, which meant that FC-U showed no statistically significant overhead based on our measurement. FC-M and FC-UM, however, showed statistically significant overhead compared to OPEN. Over a 4 second sampling period, all three FCS algorithms introduced overhead of less than 1% of the total CPU utilization. FC-U introduced the least overhead among all FCS algorithms, indicating that the utilization monitor was more efficient than the miss ratio monitor. While the utilization monitor only needs to read the /proc/stat file once every sampling period, the miss ratio monitor requires time-stamping every method invocation twice. FC-UM's overhead is slightly less than the sum of the overheads from FC-M and FC-U. This is because, while FC-UM ran both monitors, it only execute the controller and actuator once per invocation.

**Fine-grained overhead measurement**

Although the above overhead measurement shows satisfactory results based on utilization comparison between OPEN and FCS algorithms, we noticed two limitations of the above measurement approach. The first one is that the Linux system file /proc/stat records the number of *jiffies*. Since each jiffy is 10ms (1/100 of a second), the granularity of above measurement is too coarse for precise measurements on the overhead. The second problem is that CPU utilization may suffer interference from the operating system itself even though we minimized the number of system processes.

To measure overhead more accurately, we adopted a time stamping approach. Firstly, we differentiated all FCS related code from the original nORB code. Then two time stamps were taken at the starting point and finishing point of each segment of FCS code to get the execution time of FCS. Fortunately, since most FCS code is within feedback lane which is running with highest Linux real-time priority, the code segment between two timestamps will not be preempted during its execution. Hence, the time-stamped result accurately reflects the real execution overhead.

To achieve fine grained measurements, we needed an accurate time stamping function. The commonly used *gettimeofday* system call can not be used here since this function is also based on a 10ms scale. Instead we adopted a nanosecond scale time measuring function called *gethrtime*. This function uses an OS-specific high-resolution timer, which can be found on Solaris, AIX, Win32/Pentium, Linux/Pentium and VxWorks, to return the number of clock cycles since the CPU was powered up or reset. The *gethrtime* function has a low overhead and is based on a 64 bit clock cycle counter. With the clock counter number divided by the CPU speed, we can get reasonably precise and accurate time measurements. Since *gethrtime* is supported on Pentium processor, we performed our fine-grained overhead measurements on Server B, a Dell Pentium4 PC.

In Table 6.5, we list all FCS related operations and their overheads for the three FCS algorithms respectively. All results in that table are averaged values of 10 runs and each run's result is an average over 300 continuous sampling periods. Operations 1 to 4 respectively give the overhead of the utilization monitor, the miss ratio monitor, the controller, and the rate assigner, all of which ran in a feedback lane at highest priority. Operation 5 ran in the remote method invocation lane and was used to time stamp each remote method invocation from the Client side *twice* to check whether it meets the deadline as Section 3.3 explains. The overheads of operations 1 to 4 are relatively fixed for each sampling period, while the overhead of operation 5 depends on how many invocations come from Client side in a given sampling period. The measured overhead for one single *gethrtime* call is 0.0623us. With n invocations in one sampling period, the overhead for time stamping is 0.1246n us. In the total value row, we assume a common application model which has 10 tasks running at a rate of 100 invocations per second. If the sampling period is one second, we get 1000 remote method invocations per sampling period.

Table 6.5: Results of differentiated fine-grained overhead measurement

| no | Name | Description | FC-U (us) | FC-M (us) | FC-UM (us) |
|---|---|---|---|---|---|
| 1 | utilization monitor | /proc/stat system file reading | 160.90 | N/A | 263.22 |
| 2 | miss ratio monitor | Deadline miss ratio reading | N/A | 181.29 | |
| 3 | controller | Control analysis | 40.64 | 49.84 | 43.27 |
| 4 | rate assigner | Calculating new rate; transmitting new rate to client side | 659.90 | 633.73 | 637.74 |
| 5 | time stamp | Time stamping each remote method invocation twice to check deadline | N/A | 0.1246 | 0.1246 |
| | Total | (Assuming 1000 remote method invocations in each sampling period) | 861.44 | 989.46 | 1068.82 |

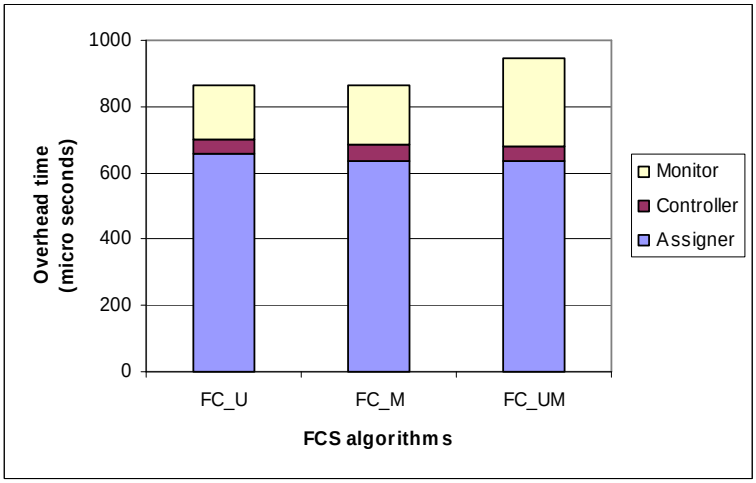



Figure 6.10: Detailed overhead measurement

From Table 6.5, it is easy to see that FC-U has the lowest overhead and FC-UM has the highest overhead. That observation is consistent with our coarse-grained overhead measurements. It is also interesting to find that fine-grained overhead result for FCS algorithms is actually much less than the result of the coarse-grained measurement. The reason is coarse-grained measurement is based on 10ms measurement accuracy so it is unable to gauge this overhead precisely.

Figure 6.10 illustrates the overheads for the monitor, controller and rate assigner in the three FCS algorithms while the overhead of time stamping is not included. Rate assigner has the dominant overhead because it involves relatively more complicated internal data structure access, modification and socket handling while there are just several lines of code for Monitor and Controller. Overall, the server overhead of

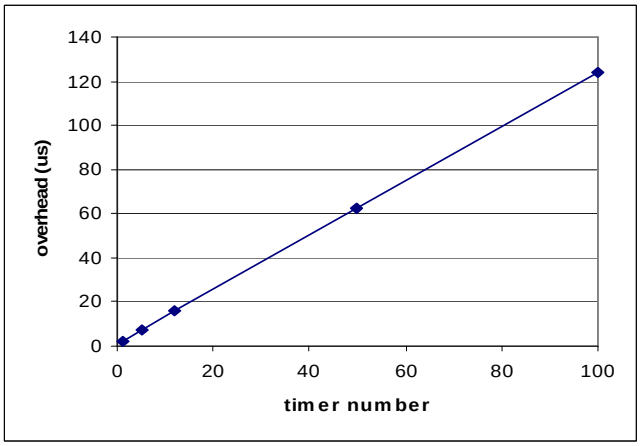



Figure 6.11: Overhead measurement of adjusting timer

all FCS algorithms in our experiments is around 1ms per sampling period, which is clearly acceptable in a wide range of real-time and embedded applications.

### Client side overhead measurement

In the previous two sections we focused on the overhead on the Server side which hosts the monitors, controller and rate assigner. To be more specific, here we also want to measure the overhead caused by FCS on the Client side. As we introduced in Section 3.3, new rates for remote method invocations are calculated on Server side and then transmitted to Client side. The only operation involved with this on the Client side is that Client needs to read these new rates and call a timer interval adjustment function to enforce the new rates. Since the delay for reading from a socket is highly dependent on the state of the network, we only focus on the overhead caused by adjusting the timer interval using the ACE *reset_timer_interval* function call [79].

In our experiment we used the time stamping approach introduced in the previous section to measure the overhead of timer interval adjustments. The test was executed at the highest real-time priority in a single thread testing program so there is no preemption or lock waiting involved. This testing program is separated from FCS/nORB to avoid the overhead of multi-thread interaction and to focus only on the overhead of resetting timer. In our experiment, we tested resetting timers with *ACE_Timer_Heap*

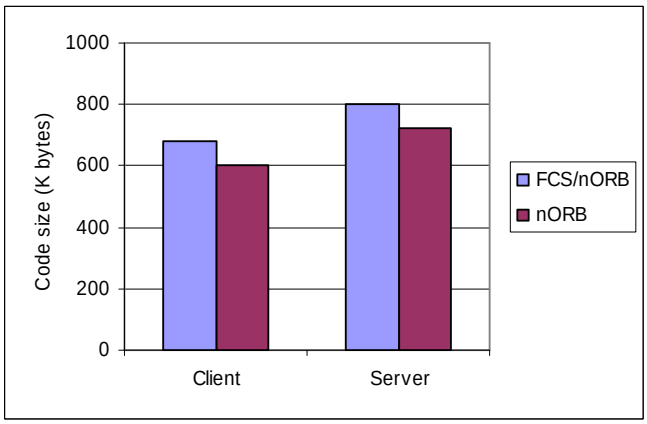



Figure 6.12: Code size difference with/without FCS service

timer queue in a fixed sequence which is consistent to FCS/nORB. In ACE [79], the execution time of *reset_timer_interval* is highly dependent on the internal timer queue structure and access sequence of different timers. Therefore, while this testing result provides a performance metric for FCS/nORB, it does not serve as a benchmark for ACE itself.

All results reported are averages of 10 separate runs. Figure 6.11 shows the overhead in terms of execution time as a function of the number of changed timers. As Figure 6.11 illustrates, the overhead for adjusting one timer interval is on the scale of several microseconds. We also observe that the measured overhead increased linearly with the number of timers. Hence, the overhead of FCS on the Client side is thus simply the product of the number of tasks and the overhead of adjusting one timer. Hence we can easily get the conclusion that even with 1000 tasks running in FCS/nORB, the total overhead on Client is just around 1ms, which is acceptable to many applications.

### Memory footprint measurement

Besides execution time, memory overhead is also a significant factor for overall system performance. For embedded systems, however, code size is a major part of the memory footprint because all code of a system is typically loaded into the memory before the system starts to execute. Hence, it is useful to measure the code size increase after we plugged in FCS related code. The measured results for both Client side

and Server side are illustrated in Figure 6.12. We see that adding FCS only resulted in an increase of 78K bytes on both the Client and Server. The ratio of increase is only 12% and 10% for Client side and Server side respectively. This minor increase is acceptable considering the system performance improvements that were seen in the previous experiments. We note that the combined static footprint on each given endsystem, of both FCS/nORB and the client or server application, is well below the static footprint of a full-featured ORB such as TAO alone (detailed footprint results for nORB and TAO are available in [81]).

## 6.5 Summary

In summary, we have designed and implemented a single-processor QoS control ORB middleware for real-time embedded systems. Performance evaluation on a physical testbed has shown that (1) FCS/nORB can guarantee specified miss ratio and CPU utilization levels even when task execution times deviate significantly from their estimated values or change significantly at run-time; (2) FCS/nORB can provide similar performance guarantees on platforms with different processing capabilities; and (3) the middleware layer instantiation of performance control loops only introduces a small amount of processing overhead on the client and server. These results demonstrate that a combination of QoS control and ORB middleware is a promising approach to achieve robust real-time performance guarantees and performance portability for DRE applications. In the next chapter, we introduce an end-to-end QoS control ORB middleware that is designed based on FCS/nORB, to provide robust end-to-end QoS guarantees.

# Chapter 7

# FC-ORB: Robust End-to-End QoS Control Middleware

A key challenge for distributed real-time and embedded (DRE) middleware is maintaining both system reliability and desired real-time performance in unpredictable environments where system workload and resources may fluctuate significantly. In this chapter, we present FC-ORB, a real-time Object Request Broker (ORB) middleware that employs the EUCON control algorithm to handle fluctuations in application workload and system resources. FC-ORB demonstrates that the integration of utilization control, end-to-end scheduling and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

## 7.1 Introduction

Distributed real-time and embedded (DRE) applications have stringent requirements for end-to-end timeliness and reliability whose assurance is essential to their proper operation. In recent years, many DRE systems have become open to unpredictable operating environments where both system workload and platform may vary significantly at run time. For example, the execution of data-driven applications such as autonomous surveillance is heavily influenced by sensor readings. External events such as detection of an intruder can trigger sudden increase in system workloads.

Furthermore, many mission-critical applications must continue to provide real-time services despite hardware failures, software faults, and cyber attacks.

While DRE middleware has shown promise in improving the real-time properties of many applications, existing middleware systems often do not work well in unpredictable environments due to their dependence on traditional real-time schedulability analysis. When accurate knowledge about workloads and platforms is not available, a DRE application configured based on schedulability analysis may suffer deadline misses or even system crash [58]. A critical challenge faced by application developers is to achieve *robust* real-time performance in unpredictable environments. Since in DRE systems, an end-to-end application that violates its real-time properties is equivalent to (or sometimes even worse than) an application that does not perform its computation, utilization guarantees affect directly the availability of the end-to-end application.

This chapter presents the design and empirical evaluation of an adaptive middleware called *FC-ORB* (Feedback Controlled ORB) that aims to enhance the robustness of DRE applications. The novelty of FC-ORB is the integration of end-to-end scheduling, adaptive QoS control, and fault-tolerance mechanisms that are optimized for unpredictable environments. Specifically, this chapter makes three contributions.

- *End-to-End Real-Time ORB*: Our ORB service supports end-to-end real-time tasks based on the end-to-end scheduling framework [55]. The FC-ORB architecture is designed to facilitate efficient end-to-end adaptation and fault-tolerance in memory-constrained DRE systems.
- *End-to-End Utilization Control*: The utilization control service enforces desired CPU utilizations in a DRE system despite significant uncertainties in system workloads. The core of the utilization control service is a distributed feedback control loop that coordinates adaptations on multiple interdependent processors.
- *Adaptive Fault Tolerance*: FC-ORB handles processor failures with an adaptive strategy that combines reconfigurable utilization control and task migration. A unique feature of our fault tolerance approach is that it can maintain *real-time* properties for DRE applications after a processor failure.

FC-ORB has been implemented and evaluated on a Linux platform. Our experimental results demonstrate that FC-ORB can significantly improve the end-to-end real-time performance of DRE middleware in face of a broad set of dynamic uncertainties and fluctuations in task execution times, resource contention from external workloads, and processor failures. FC-ORB demonstrates that the integration of utilization control, end-to-end scheduling, and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

FC-ORB is particularly useful for DRE applications that are amenable to rate adaptation such as digital feedback control systems [64][83], monitoring systems [105], and multimedia [10]. In these systems, task rates can be adjusted without causing system failure. Furthermore, tasks running at higher rates contribute higher values to the application (e.g. increasing the sampling rate of a digital controller improves the control performance). Our framework can benefit Supervisory Control and Data Acquisition (SCADA) systems which provide monitoring and control functions that are inherently periodic at geographically distributed sites.

The rest of the chapter is organized as follows. Section 7.2 describes the design of the FC-ORB architecture. Section 7.3 presents the experimental results. Section 7.4 summaries the chapter.

## 7.2 Design of the FC-ORB Architecture

In this section, we first introduce the end-to-end task model and scheduling framework supported by FC-ORB. We then describe the main components of FC-ORB: the end-to-end ORB service, the utilization control service, and the adaptive fault-tolerance mechanisms.

### 7.2.1 Applications

FC-ORB supports an end-to-end task model [55] employed by many DRE applications. An application is comprised of $m$ periodic tasks $\{T_i|1 \leq i \leq m\}$ executing on $n$ processors $\{P_i|1 \leq i \leq n\}$. Task $T_i$ is composed of a chain of subtasks $\{T_{ij}|1 \leq j \leq n_i\}$ which are implemented as a sequence of object operations on different processors.[1] A subtask may be executed by one or more operation requests on a same processor. The invocation of a subtask $T_{ij}(1 < j \leq n_i)$ is triggered by its predecessor $T_{ij-1}$ through a remote operation request. A non-greedy synchronization protocol called release guard [91] is used to ensure that the interval between two consecutive releases of the same subtask is not less than its period. Hence, all the subtasks of a periodic task share the same rate as the first subtask. In FC-ORB, the rate of a task (and all its subtasks) can be adjusted by changing the rate of its first subtask. An example DRE application with two end-to-end tasks running on three processors is shown in Figure 7.1.

Our application model has two important properties. First, while each subtask $T_{ij}$ has an *estimated* execution time $c_{ij}$ available at design time, its *actual* execution time may be different from its estimation and may vary at run-time. Such uncertainty is common for DRE systems operating in unpredictable environments. Second, the rate of a task $T_i$ may be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. This assumption is based on the fact that the task rates in many DRE applications (e.g., digital control [64][83], sensor update, and multimedia [10]) can be dynamically adjusted without causing system failure. A task running at a higher rate contributes a higher value to the application at the cost of higher utilization. For instance, although a digital control system usually has better control performance when it executes at a higher rate, it can usually remain stable when executing at a lower rate.

Each task $T_i$ is subject to an end-to-end soft deadline related to its period. FC-ORB implements the end-to-end scheduling approach [91] to meet task deadlines. The deadline of a task is divided into subdeadlines of its subtasks [42][66]. The release guard protocol is used to synchronize the execution of subtasks such that each subtask can be modeled as a periodic task. Hence, the problem of meeting the deadline is

[1] FC-ORB can be extended to support a more general task model in which a task is composed of a graph of subtasks[55].

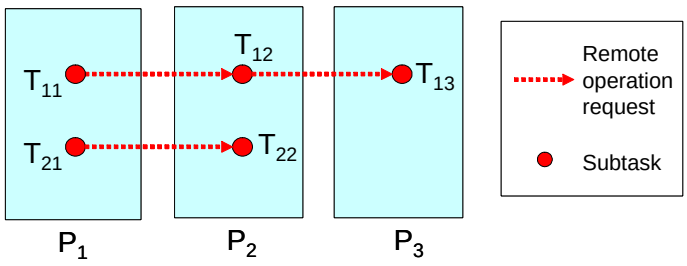



Figure 7.1: An example DRE application

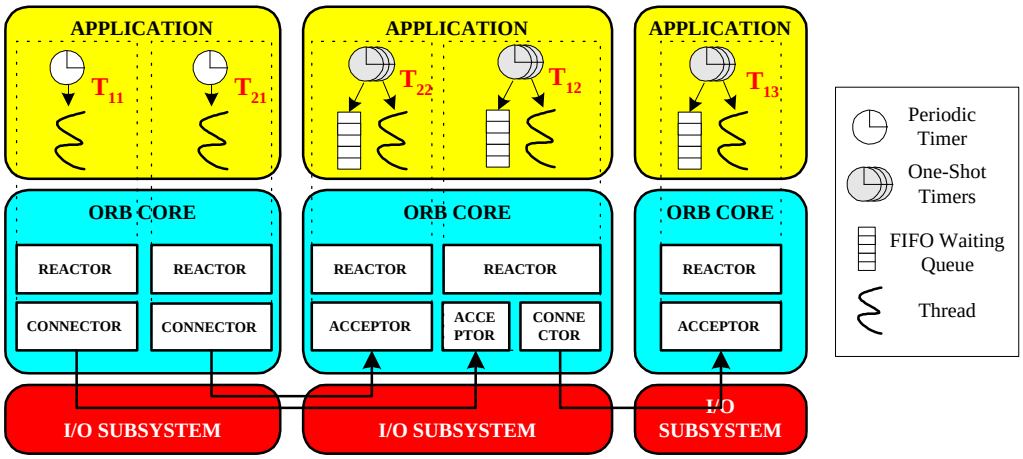



Figure 7.2: FC-ORB's end-to-end architecture

transformed to the problem of meeting the subdeadline of each subtask. A well known approach for meeting the subdeadlines on a processor is to ensure that its utilization remains below its schedulable utilization bound [50][54]. Therefore the end-to-end scheduling approach enables FC-ORB to meet end-to-end deadlines by controlling the utilizations of all processors in the system.

### 7.2.2 Middleware Support for End-to-End Tasks

#### Implementation of End-to-End Tasks

Figure 7.2 illustrates the FC-ORB implementation of the example DRE application shown in Figure 7.1. Each subtask is executed by a separate thread whose priority is decided by a priority manager. In Figure 7.2, each dashed box spanning from the

application layer to the ORB core layer represents a subtask in Figure 7.1. Every subtask is associated with a separate Reactor [77] to create timeout events and to manage communication connections.

As shown in Figure 7.2, the first subtask of a task is implemented with a periodic ACE timer, a Reactor, and a Connector [78] . The ACE Connector framework is used to decouple communication initialization from application-specific tasks that communication services perform once initialization is complete. Connector can be configured with different IPC mechanisms to support communication in different distributed applications. The ACE Reactor framework is an extensible, object-oriented demultiplexer that dispatches events to application-specific handlers. It can support I/O-based, timer-based, signal-based, and synchronization-based events. It simplifies the development of event-driven programs for many distributed applications. The timer periodically triggers a local operation (e.g., a method of an object) which implements the functionality of this subtask. Following the execution of this operation, a one-way remote operation request is pushed through the Connector to the succeeding subtask that is located on another processor. The succeeding subtask employs an Acceptor [78] to accept the request from its preceding subtask. Each pair of Connector and Acceptor maintains a separate TCP connection to avoid priority inversion in the communication subsystem. The release guard protocol enforces to be the interval between two successive invocations of a same subtask is bounded below by its period. Earlier research has shown that the release guard protocol can effectively reduce the end-to-end response time and jitter of tasks in DRE systems [91]. FC-ORB implements the release guard protocol with a FIFO waiting queue and one-shot ACE timers. Upon receiving a remote operation request, a subtask compares the current time with the last invocation time of this operation. Based on the release guard rules [91], the subtask either immediately invokes the requested operation or enqueues this request to the waiting queue if the request arrives too early. When the request is enqueued, a one-shot ACE timer is registered with the Reactor to trigger the requested operation at the time that equals the last invocation time plus the task's period. After the one-shot timer fires and the enqueued request is served, a remote operation request is sent to the next subtask in the end-to-end task chain. An end-to-end real-time task is completed when the execution of its last subtask is completed.

**Priority Management**

The integration of end-to-end scheduling and utilization control introduces new challenges to the design of scheduling mechanisms in ORB middleware. For instance, the rate adaptation mechanism adopted by FC-ORB and several other projects [58][60] may dynamically change the rates of end-to-end tasks. This may cause the middleware to change the priorities of all its subtasks, e.g., when the Rate Monotonic Scheduling (RMS) policy is used. To satisfy the special requirements posed by rate adaptation and end-to-end scheduling, our ORB service is configured with the *server-declared priority* model [82] and the *thread-per-subtask* concurrency architecture.

To support the server-declared priority model, FC-ORB implements a priority manager on each processor to assign priorities to local subtasks. The incoming requests from another processor are served by a thread with a real-time priority dictated by the priority manager located on the host processor. Currently the priority manager only supports the RMS policy, although the following discussions are also applicable to other rate- or deadline-dependent scheduling policies (note that task deadlines are usually related to their periods). There are several advantages of using server-declared priority model in the FC-ORB system. First, each processor is able to change thread priorities locally, based on the current rates of the subtasks located on it, so a processor only needs to know the local subtasks. This makes the system more scalable to large applications. Moreover, the server-declared model has less overhead because it does not have to adjust a thread's priority every time the priority of its predecessor subtask is changed, as it would do with the client-propagated model.

The thread-per-priority concurrency architecture has been adopted in existing DRE middleware (e.g., [81]). In this model,the same thread is responsible for executing all subtasks with a same priority. This is because the workload is assumed to use only a limited number of fixed task rates. However, this concurrency architecture is not suitable for rate adaptation. Due to rate adaptation, the rates and thus the priorities of subtasks vary dynamically at run-time. In such situations, the thread-per-priority architecture would require the ORB to dynamically move a subtask from one thread to another thread which can introduce significant overhead.

To avoid this problem FC-ORB implements the thread-per-subtask architecture that executes each subtask with a separate thread. FC-ORB adjusts the priorities of the threads only when the *order* of the task rates is changed. While the task rates may vary at every control period, the order of task rates often changes at a much lower frequency. Therefore, the thread-per-subtask architecture enables FC-ORB to adapt task rates in a more flexible way, with less overhead.

A potential advantage of the thread-per-priority architecture is that it may need fewer threads to execute applications. However, as FC-ORB is targeted at memory-constrained networked embedded systems that commonly have limited number of subtasks on a processor, each subtask can be easily mapped to a thread with a unique native thread priority even in a thread-per-subtask architecture.

### 7.2.3 End-to-End Utilization Control Service

FC-ORB allows users to specify a set of application parameters in a configuration file that is used to initialize the middleware when the system is started. Configuration parameters include the desired CPU utilization on each processor and the allowed range of rate for each real-time task. The utilization control service dynamically enforces the desired CPU utilizations on all processors by adapting the rates of real-time tasks within the specified ranges, despite significant uncertainties and fluctuation in system workload and platform. Therefore, to meet end-to-end deadlines, the application users only need to specify the utilization reference of each processor to a value below its schedulable utilization bound.

In the rest of this subsection, we first give an overview of the feedback control loop of the utilization control service, and then describe each component of the loop in detail.

#### Feedback Control Loop

The utilization control service implements the EUCON algorithm [60] as a distributed feedback control loop in the middleware. As shown in Figure 7.3, the feedback control

loop is composed of a utilization monitor, a rate modulator and a priority manager on each processor, and a centralized controller.

The feedback control loop is invoked at the end of every sampling period. It works as follows: (1) the utilization monitor on each processor sends its utilization in the last sampling period to the controller; (2) the controller collects the utilizations from all processors, computes the new task rates, and sends the new task rates to the rate modulators on all processors where the tasks are running; (3) the rate modulators on processors that host the first subtasks of tasks change the rates of the first subtasks according to the input from the controller; and (4) the priority manager on each processor check and adjust the thread priorities based on the new task rates if necessary.

The controller computes the new task rates using a Model Predictive Control (MPC) algorithm. The control algorithm solves at every time step an optimization problem that minimizes the difference between the desired and the actual utilizations subject to the task rate constraints. The optimization problem is a constraint least-square problem that can be solved efficiently using quadratic programming. Assuming that the optimization problem is feasible, i.e. there exist task rates that satisfy the utilization bounds, the stability of the controller can be formally analyzed and provide statistical guarantees for the schedulability of the system. Details can be found in [60].[2]

As shown in Figure 7.3, the three components of the feedback control loop on an application processor (i.e., a processor executing applications and the ORB) are executed by a separate thread called the *control thread*. This control thread has the highest priority in the middleware system so that the feedback control loop can be executed in overload conditions, when it is needed most. The controller is implemented as an independent process that can be deployed on a separate processor or on an application processor. The controller also serves as a coordinator of the FC-ORB system. Every application processor in the system tries to connect with the controller through a TCP connection (called *feedback lane*) when the node is started. Once all

[2]We note that, as the feedback control loop is designed to control the *average* utilization within each sampling period, transient deadline misses may occur during a sampling period. Therefore FC-ORB can only provide a statistical guarantee on utilization and deadlines.

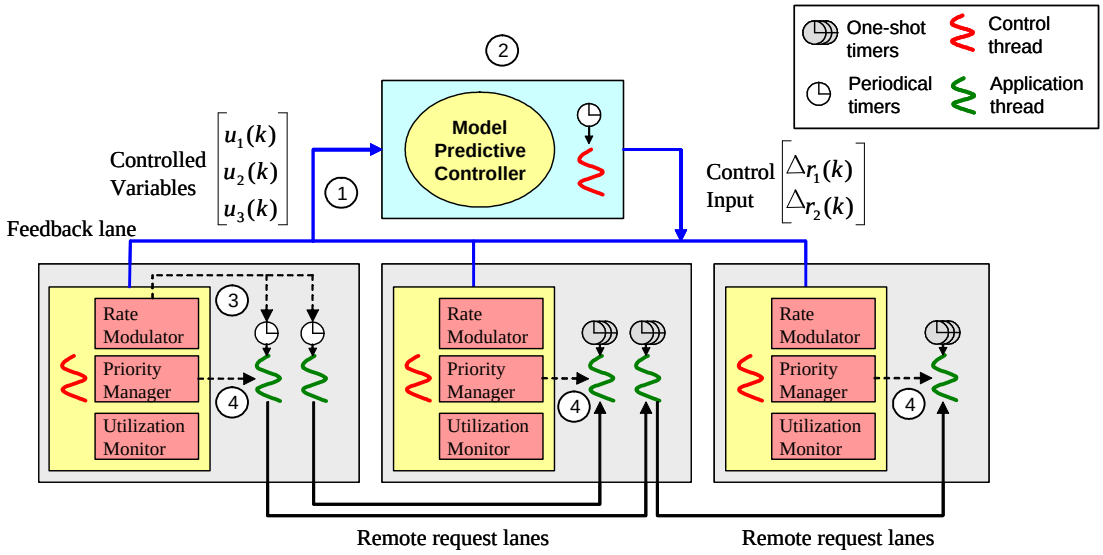



Figure 7.3: The distributed feedback control loop of the utilization control service

application processors are connected to the controller, the whole system starts to run the configured application.

### Control Components

We now present the details of each utilization control component.

**Controller:** The controller is implemented as a single-thread process. It employs a Reactor to interact with all processors in the system. Each time its periodic timer fires, it sends utilization requests to all application processors through the feedback lanes. The incoming replies are registered with the Reactor as events to be handled asynchronously. This enables the controller to avoid being blocked by an overloaded application processor. After it collects the replies from all processors, it executes a MPC algorithm proposed in [60] to calculate the new task rates. Then, for each task whose rate needs to be changed, the controller sends the task's new rate to all processors that host one or more subtasks of the tasks whose rates have been changed. If a processor does not reply in an entire control period, its utilization is treated as 100%, as the controller assumes this processor is saturated by its workload.

**Utilization Monitor:** The utilization monitor uses the /proc/stat file in Linux to estimate the CPU utilization in each sampling period. The /proc/stat file records the number of jiffies (usually 10ms in Linux) when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle task, since the system starts. At the end of each sampling period, the utilization monitor reads the counters, and estimates the CPU utilization as 1 minus the number of jiffies used by the idle task in the last sampling period divided by the total number of jiffies in the same period.

**Rate Modulator:** A Rate Modulator is located on each processor. It receives the new rates for its remote invocation requests from the controller through the feedback lane, and resets the timer interval of the first subtask of each task whose invocation rate has been changed.

**Priority Manager:** All processors in FC-ORB assign priorities to their subtasks based on a real-time scheduling algorithm (e.g., RMS). It is important to strictly enforce the scheduling algorithm to achieve desired real-time performance. However, as a result of rate adaptation, a task with a rate higher than another task could be assigned a lower rate in the next sampling period. Consequently, the priority of this task has to be adjusted at run-time. The priority manager on each processor checks the rate order of all subtasks on this processor. If the rate order of two or more subtasks is reversed, the priority manager reassigns the correct priorities for the threads of those subtasks.

### 7.2.4 Fault Tolerance

A robust DRE middleware must maintain both reliability and real-time properties required by the applications despite partial system failure. Traditional fault-tolerance mechanisms usually focus on reliability aspects of the system based on *entity redundancy*. No single point of failure, transparent failover and transparent redirection, and reinvocation are among the requirements of a fault-tolerant ORB [32]. However, less attention has been paid to maintaining desired real-time properties in the presence of faults.

Before describing the fault tolerance techniques in FC-ORB, we first introduce the fault model. FC-ORB is designed to handle one or more persistent processor failures. The fault model, known as *fail-stop processors*, is very important and has been considered extensively in the design of fault-tolerant computing systems [76][26]. We assume that the communication between the remaining processors does not fail and the network is not overloaded. This is a reasonable assumption for a common class of DRE systems with processors connected by a switched/fast Ethernet LAN with sufficient bandwidth. It should be noted that our utilization control service can be integrated with more sophisticated fault detection and recovery techniques to handle more complex fault models.

FC-ORB improves system robustness in terms of both reliability and real-time properties by integrating three complementary mechanisms. First, FC-ORB provides replication for subtasks and supports transparent failover to backup subtasks located at different processors in face of processor failure. Second, after a processor fails, the remaining processors may experience dramatic workload increase due to the activation of the backup subtasks, which may cause them to miss deadlines or fail. A unique feature of FC-ORB is that it can effectively handle the workload increase via utilization control so that applications can maintain desired real-time properties despite processor failure. Finally, the FC-ORB controller can automatically reconfigure itself at runtime to rebuild its control model, in order to effectively control the DRE system whose deployment is changed due to processor failure.

In our replication mechanism, a subtask may have a backup subtask located on a different processor. For example, the subtask $T_{13}$ shown in Figure 7.1 can have a backup subtask $T'_{13}$ located on processor $P_1$. As a result, when processor $P_3$ fails because of hardware failure, the execution of subtask $T_{13}$ is migrated to processor $P_1$ to continue automatically. Similar to the *COLD_PASSIVE* replication style used in Fault-Tolerant CORBA (FT-CORBA) [32], all subtasks are assumed to be stateless (except the connections between subsequent subtasks which are maintained by the middleware) so that the overhead of active state synchronization is avoided.

The failover mechanism works as follows. In the normal mode, each subtask pushes remote operation requests only to the primary instance of its successor. As a result, the backup instance does not receive any requests and its thread remains idle.

After a processor fails, the predecessor of a subtask located on the failed processor detects the communication failure based on the underlying socket read/write errors. The predecessor immediately switches the connection to the backup instance of its successor and sends the remote operation requests to it. In the case when the failed processor hosts the first subtask of a task, the controller activates the backup instance of the subtask. Consequently, the execution of the end-to-end tasks is resumed after a transient interruption.

As a part of the fault-tolerant support, the controller in the utilization control service has been designed to be self-configurable. This is important because the control algorithm relies on knowledge about the subtask allocation in order to compute correct task rates [60]. When the controller detects communication failure with a processor in the system, it first cancels the periodic timer to pause the feedback control loop. In its internal control model, it then removes the failed processor and moves the subtasks located on the failed processor to the corresponding backup processors. After rebuilding the control model, the controller re-initializes itself and restarts the timer to resume the feedback control loop.

A disadvantage of the centralized control scheme is that the controller becomes a single point of failure. To mitigate this problem, FC-ORB can be easily extended to replicate the controller as well. In this extension, FC-ORB can actively maintain state consistency between the primary controller and the backup controller, in a way similar to the *ACTIVE* replication style used in FT-CORBA [32]. When the controller executes in replicated mode, all processors send their CPU utilizations to both the primary and the backup controllers at every sampling instant. The backup controller performs control computation just like the primary controller. The difference is that the backup controller does *not* send the resultant new task rates to any processor. Instead, it uses this method to keep the state variables in the backup controller consistent with the primary controller. The primary and backup controllers can exchange heartbeat messages in every sampling period. Once the backup controller stops receiving heartbeats from the primary controller, the backup controller takes over the utilization control service. This feature will allow FC-ORB to maintain control of the entire system even after controller failures.

### 7.2.5 Implementation

FC-ORB 1.0 has been implemented in C++ using ACE 5.4 on Linux. FC-ORB is based on the FCS/nORB middleware [58] which integrates a *single-processor* feedback control scheduling service and a light-weight real-time ORB middleware called nORB [90]. FC-ORB is specialized for memory-constrained DRE systems by supporting a smaller set of features than general-purpose DRE middleware such as TAO. The entire FC-ORB middleware (excluding the code in ACE library and IDL library) is implemented in 7017 lines of C++ code. The controller is implemented in 2089 lines of C++ code and a dynamically linked library that implements the constrained least square solver. We use MATLAB Compiler to create the dynamically linked library from *lsqlin.m* in the MATLAB . At the end of each sampling period, the controller collects the utilizations from application processors and calls the *lsqlin* function in the dynamically linked library with the utilizations as parameters. The *lsqlin* function computes the control input and returns it to the controller. All the code is open-source and can be downloaded from `http://deuce.doc.wustl.edu/FCS_nORB/FC-ORB/`.

## 7.3 Empirical Evaluation

In this section, we present the results of five sets of experiments run on a distributed testbed with five machines. Experiments I and II evaluate FC-ORB's performance when task execution times deviate from their estimations and change dynamically at run-time, respectively. Experiment III examines FC-ORB's capability to handle disturbances from external workloads. Experiment IV tests FC-ORB's robustness in face of processor failure. Finally, Experiment V measures the overhead introduced by utilization control.

### 7.3.1 Experimental Setup

All experiments are conducted on a testbed of five machines. All applications and the ORB service run on a Linux cluster composed of four Pentium-IV machines: Ron, Harry, Norbert, and Hermione. Ron and Hermione are 2.80GHz, and Harry and

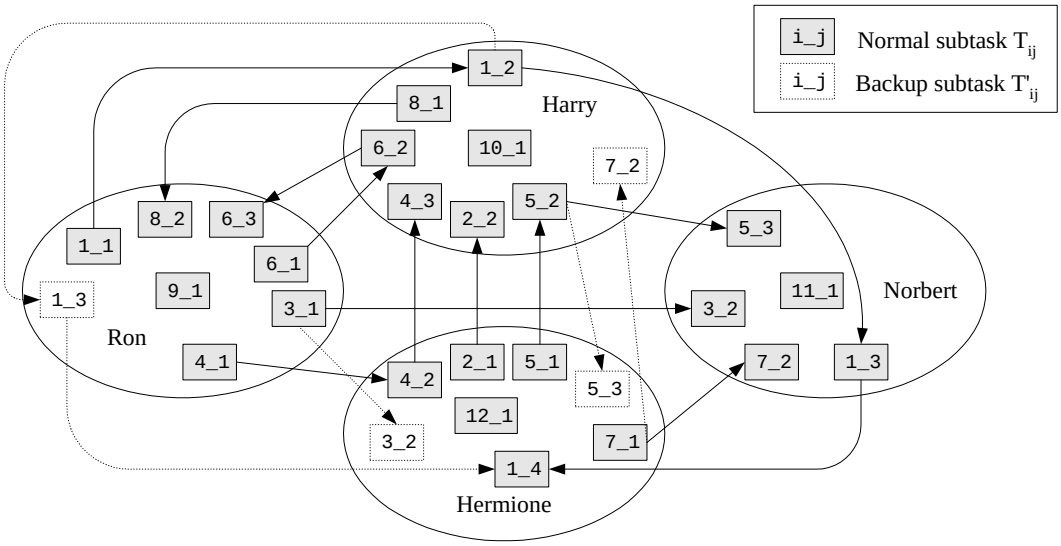



Figure 7.4: A medium size workload

Norbert are 2.53GHz. All four machines are equipped with 512KB cache and 512MB RAM, and run KURT Linux 2.4.22. The controller is located on another Pentium-IV 2.53GHz machine with 512KB cache and 512 MB RAM. The controller machine runs Windows XP Professional. The four machines in the cluster are connected via an internal switch and communicate with the controller machine through the departmental 100Mbps LAN.

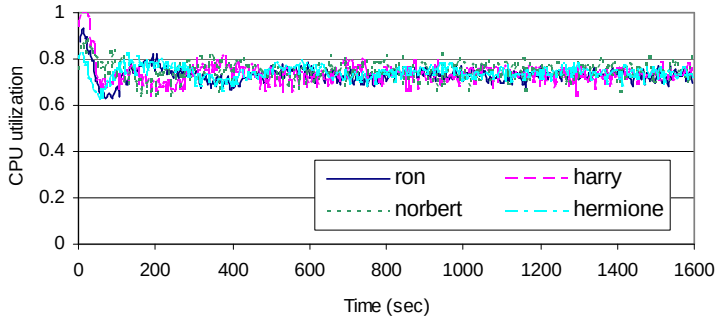
All the experiments run a medium-sized workload that comprises 12 tasks (with a total of 25 subtasks). The tasks include 8 end-to-end tasks (tasks $T_1$ to $T_8$) and 4 local tasks. Figure 7.4 shows how the 12 tasks are distributed on the 4 application processors. A processor failure incident on Norbert is emulated in Experiment IV to test FC-ORB's fault-tolerance capability. Hence in Figure 7.4, we also show the configured backup subtasks for all subtasks on Norbert that belong to an end-to-end task. There is no backup subtask for local task $T_{11,1}$ as we assume that the local task is specific to Norbert.

The subtasks on each processor are scheduled by the RMS algorithm [54]. Each task's end-to-end deadline is $d_i = n_i/r_i(k)$, where $n_i$ is the number of subtasks in task $T_i$ and

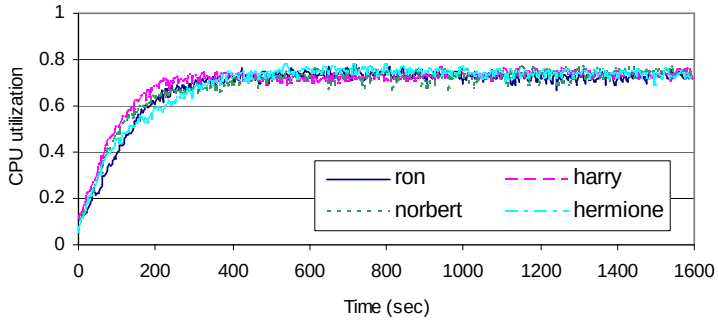
$r_i(k)$ is the current rate of $T_i$. Each end-to-end deadline is evenly divided into subdeadlines for its subtasks. The resultant subdeadline of each subtask $T_{ij}$ equals its period, $1/r_i(k)$. Hence the schedulable utilization bound of RMS [54], $B = m(2^{1/m} - 1)$ is used as the utilization set point on a processor, where $m$ is the number of subtasks (including backup subtasks) on this processor. Specifically, the utilization set points for the four experiment processors are: Ron (72.4%), Harry (72.4%), Norbert (74.3%), and Hermione (72.4%). All (sub)tasks meet their (sub)deadlines if the desired utilization on every processor is enforced. The sampling period of the utilization control service is $T_s = 4$ seconds.

To evaluate the robustness of FC-ORB when execution times deviate from the estimations, the execution time of each subtask $T_{ij}$ can be changed by tuning a parameter called the *execution-time factor*, $etf_{ij}(k) = a_{ij}(k)/c_{ij}$, where $a_{ij}$ is the actual execution time of $T_{ij}$. The execution time factor ($etf$) represents how much the actual execution time of a subtask deviates from the estimation. The $etf$ (and hence the actual execution times) may be kept constant or changed dynamically in a run. In the following we use *inversed etf* (*ietf*,specifically, $ietf_{ij}(k) = 1/etf_{ij}(k)$) because DRE systems commonly have undesired oscillation when execution times are underestimated (i.e. $etf > 1$).

We compare FC-ORB against a baseline called OPEN. In OPEN, the utilization control service of FC-ORB is turned off and the middleware becomes a representative real-time ORB without control. OPEN uses a typical open-loop approach to assign task rates based on *estimated* execution time to achieve the desired utilizations. OPEN results in desired utilization when estimated execution times are accurate (i.e.,$ietf = 1$). However, it causes underutilization when execution times are overestimated (i.e., $ietf > 1$), and over-utilization when execution times are underestimated (i.e., $ietf < 1$). This is a common problem faced by application developers because it is often difficult to estimate a tight bound on execution times, especially in unpredictable environments where execution times are heavily influenced by the value of sensor data or user input.




(a) ietf = 0.5




(b) ietf = 4

Figure 7.5: CPU utilizations under FC-ORB when task execution times deviate from estimations

### 7.3.2 Experiment I: Uncertain Execution Times

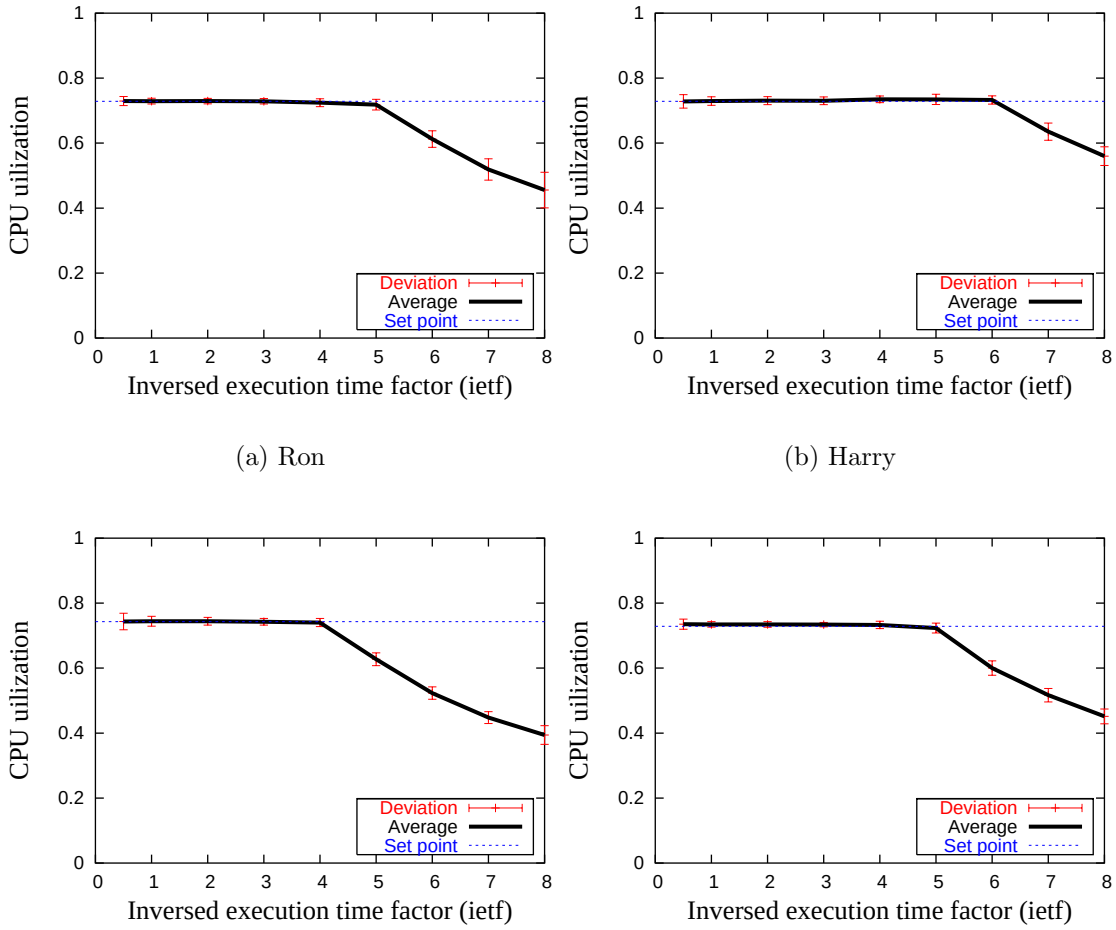
In this subsection, we evaluate FC-ORB's performance when task execution times deviate from the estimations. In each run of this experiment, all subtasks share a fixed execution-time factor ($ietf$).

First, we run experiments for OPEN which chooses task rates based on estimated execution times so that the estimated utilizations of all processors equal their set points. While the system achieves the desired utilizations in the ideal case when $ietf = 1$, all processors freeze when we set the $ietf$ to 0.5. This is not surprising, because the actual execution time of every subtask in the system is *twice* its estimated

execution time when $ietf = 0.5$. Consequently, the requested utilization on each processor is about 145% (twice of the desired utilization). Since all FC-ORB threads run at real-time priorities that are higher than the kernel priority on Linux, no kernel activities are able to execute causing the system to crash. This result shows that uncertainties in workloads can significantly degrade the robustness of applications on DRE middleware. On the other hand, the utilizations of all processors drop to only around 18% under OPEN when the actual execution times are only a *quarter* of their estimations ($ietf = 4$). This results in a extremely underutilized system and unnecessarily low task rates.

In contrast, FC-ORB achieves the desired utilizations on all processors even when execution times deviate significantly from the estimations. Figure 7.5(a) shows the utilizations when the average execution time of every subtask is twice its estimation. In the beginning, all processors are overutilized because of the initial task rates. The utilization control service quickly decreases the task rates until the utilizations of all processors converge to the desired levels in around 400 seconds. Figure 7.5(b) shows the utilizations of all processors when the execution time of every subtask is severely overestimated ($ietf = 4$). In this case, all processors are initialized underutilized due to the low execution times. FC-ORB then increases the task rates until the utilizations of all processors converge to the set points roughly at 500 seconds. In this experiment, the utilization control service successfully prevents the system from crashing and underutilization via rate adaptation.

To examine FC-ORB 's performance under different execution time factors, we plot the mean and standard deviation of utilizations of all processors during each run in Figure 7.6. Every data point is based on the measured utilization $u(k)$ from time 1200 seconds to 1600 seconds to exclude the transient response at the beginning of each run. FC-ORB consistently achieves the desired utilizations for all tested execution-time factors within the $ietf$ range $[0.5, 4]$ which corresponds to eight times variation in execution times. The results show that FC-ORB can enhance system reliability and achieve robust real-time performance under a wide range of operating conditions. Interestingly, when the $ietf$ is lower or equal to 0.33, the system freezes due to the extremely high utilization in the beginning of the run. Even though the control thread runs at highest real-time priority, the communication subsystem of Linux runs only at kernel priority. Therefore, the control thread of FC-ORB is blocked on communication




(a) Ron

(b) Harry

(c) Norbert

(d) Hermione

Figure 7.6: CPU utilizations of all processors under different execution-time factors

because the Linux kernel is preempted by the middleware threads. As a result, the system fails to recover promptly from overload when the *ietf* is equal to or lower than 0.33, even with the help of FC-ORB. In addition, as observed in [60], the EUCON algorithm can cause performance oscillation when execution times are underestimated ($ietf < 1$). Therefore, application developers should use pessimistic estimations of task execution times in FC-ORB . A fundamental advantage of FC-ORB is that it does not cause system underutilization even when task execution times are severely overestimated.

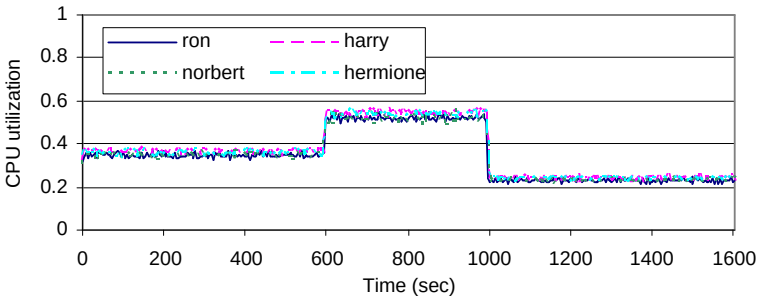
However, we note that some processors fail to reach the utilization set points when *ietf* is equal to or larger than 5. This is because the achievable utilizations are limited by the task rate constraints. For example, when *ietf* is 6, even though the rates of all subtasks on Norbert are adjusted to the maximum values, the utilization of the processor remains below the utilization set point. Note that this is the desired behavior, i.e., task rates are maximized when the system is underloaded.

### 7.3.3 Experiment II: Varying Execution Times

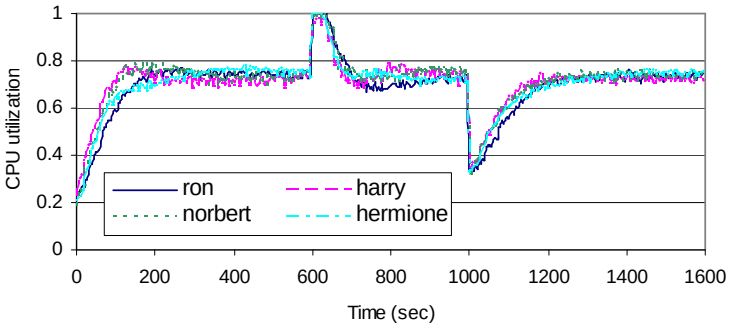
The second set of experiments tests FC-ORB's ability to maintain robust real-time performance when task execution times vary *dynamically* at run-time. To investigate the robustness of FC-ORB we create two scenarios of workload fluctuation. In the first set of runs, the average execution times on all processors change simultaneously. In the second set of runs, only the execution times on Ron change dynamically, while those on the other processors remain unchanged. The first scenario represents *global* load fluctuation, while the second scenario represents *local* fluctuation on a part of the system.

Figure 7.7(a) shows a typical run of OPEN under global workload fluctuation. The *ietf* is initially 2. At 600 seconds, it is decreased to 1.33, which corresponds to a 50% increase in the execution times of all subtasks. At time 1000sec, the *ietf* is increased to 3 to emulate a 56% decrease in execution times. OPEN fails to achieve the desired utilizations due to the lack of dynamic adaptation. In sharp contrast to OPEN, FC-ORB effectively maintains the desired utilizations on all processors under the same workload. As shown in Figure 7.7(b), the *ietf* changes to 1.33 at 600 seconds such that all processors are suddenly overloaded. FC-ORB responds to the overload condition by decreasing task rates which causes the utilizations on all processors to re-converge to their set points within 100 seconds (25 control periods). At 1000 seconds, the utilizations on all processors drop sharply due to the 56% decrease in execution times, causing FC-ORB to dramatically increase task rates until the utilizations re-converge to their set points.

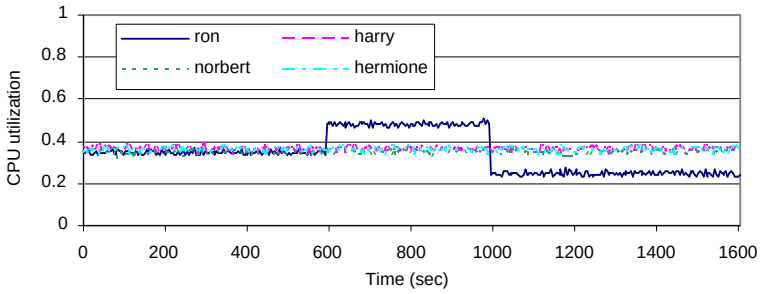
In each run with local workload fluctuation, as shown in Figure 7.7(c), the *ietf* on Ron follows the same variation as the global fluctuation, while all the other processors



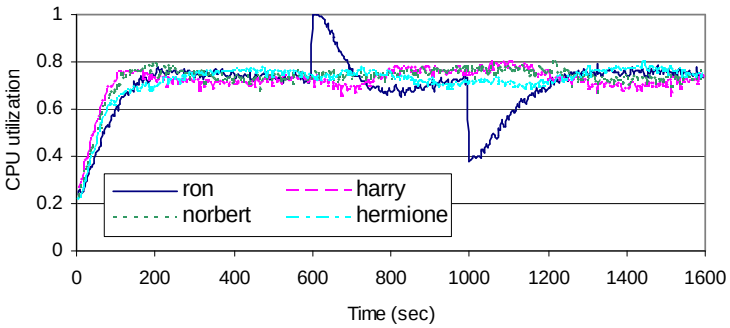

(a) OPEN with global fluctuation




(b) FC-ORB with global fluctuation




(c) OPEN with local fluctuation




(d) FC-ORB with local fluctuation

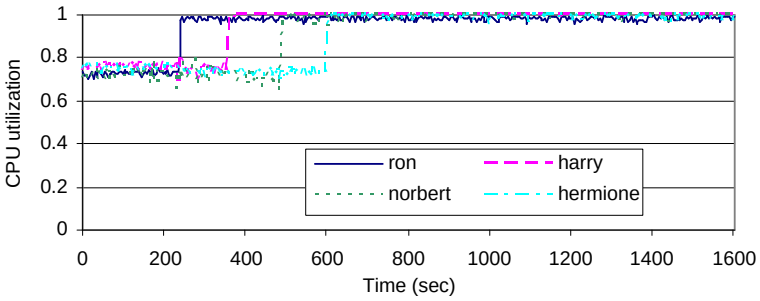
Figure 7.7: CPU utilizations of all processors when execution times fluctuate at run-time ($ietf = 2$)

have a fixed *ietf* of 2. As shown in Figure 7.7(d), under FC-ORB the utilization of Ron converges to its set point after the significant variation of execution times at 600 seconds and 1000 seconds, respectively. We also observe that the other processors experience only slight utilization fluctuation after the execution times change on Ron. This result demonstrates that FC-ORB effectively handles the interdependencies among processors during rate adaptation.

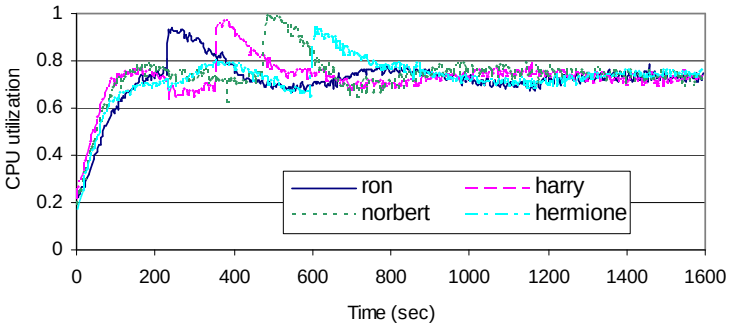
### 7.3.4 Experiment III: External Disturbances

We now evaluate FC-ORB under resource contention from external workloads that are not controlled by FC-ORB. Such external disturbances may be caused by a variety of sources including (i) processing of critical events that must be executed at the cost of other tasks, (ii) varying workload from a different subsystem (e.g., legacy software from a different vendor), and (iii) software faults or adversarial cyber attacks. To stress-test FC-ORB, we emulate the external disturbances using a high priority real-time process to compete with FC-ORB for CPU resource. To investigate the robustness of FC-ORB we create both periodic and aperiodic disturbances. In the first set of runs, the external process *periodically* invokes a function with a *fixed* execution time of 100ms every 500ms. In the second set of runs, the external process *aperiodically* invokes another function with a *random* execution time. Both the request interarrival time and the execution time follow exponential distributions with mean values of 50ms and 10ms, respectively.

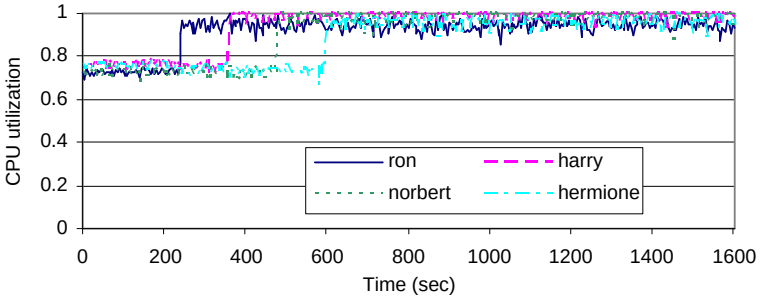
The workload controlled by FC-ORB has an $ietf = 2$. Here we manually configure the task rates in OPEN such that the workloads achieve the desired utilizations without the external disturbances. As shown in Figure 7.8(a), the system does achieve the required performance initially. However, at time 240sec, 360sec, 480sec and 600sec, the external task is activated sequentially on Ron, Harry, Norbert and Hermione. Consequently, the utilizations of all processors are raised to 100%. In contrast to OPEN, Figure 7.8(b) shows that FC-ORB successfully maintains the desired utilizations and thus tolerates the external resource contention. Similar situations occur for aperiodic disturbance, except that in this case, both OPEN and FC-ORB have higher fluctuation. Despite noise introduced by the aperiodic requests, FC-ORB still



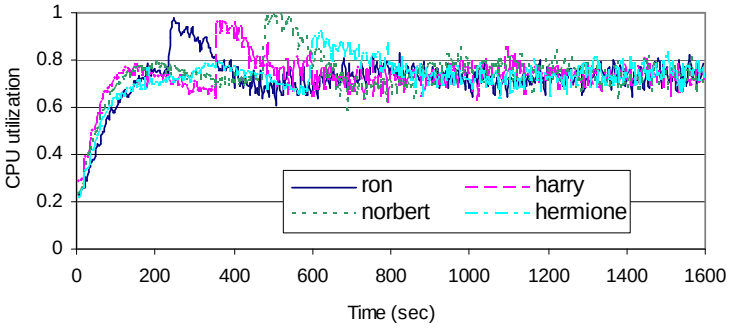

(a) OPEN with periodic disturbance




(b) FC-ORB with periodic disturbance




(c) OPEN with aperiodic disturbance




(d) FC-ORB with aperiodic disturbance

Figure 7.8: CPU utilizations of all processors under external disturbances ($ietf = 2$)

successfully maintains the CPU utilization under 80% most of the time and achieves the desired CPU utilizations on average.

### 7.3.5 Experiment IV: Processor Failure

In this experiment, we evaluate FC-ORB's ability to recover from processor failure. At 800 seconds, we emulate the failure of Norbert by using the Linux *kill* command to eliminate the process which carries FC-ORB and the application. The CPU utilization of Norbert immediately drops to almost zero because no other application is running on Norbert. All subtasks on Norbert have backup subtasks located on other processors as shown in Figure 7.4, except the local task $T_{11,1}$. Their preceding subtasks on other processors detect the communication failure with Norbert and then redirect the remote operation requests to the backup subtasks. Hence, the load of Norbert is distributed to the other 3 processors in the system.

As demonstrated in Figure 7.9, the CPU utilizations of the other 3 processors increase simultaneously after the failure of Norbert. At the same time, the controller on the control processor re-configures itself to rebuild its control model after it detects the communication failure with Norbert. Thanks to the utilization control service, the high utilizations on the other 3 processors quickly converge to the desired utilization bounds within 100 seconds so the desired end-to-end real-time performance is ensured. Our results demonstrate that the system successfully recovers from a processor failure and the utilization of the remaining processors converges to a desirable state that ensures the real-time properties of the end-to-end application.

Fault injection using the *kill* command allows us to focus on the robustness of the utilization control service rather than the error detection method. Error detection is a complementary problem to the FC-ORB adaptation for error recovery. Our experimental evaluation of the FC-ORB robustness can be extended to more realistic processor crash failures assuming an appropriate error detection method. The time required for error recovery will include both the time needed for error detection and the convergence of the utilization control service. Formally evaluating the availability of the distributed application requires the definition of an appropriate benchmark [65], and is a subject of future work.

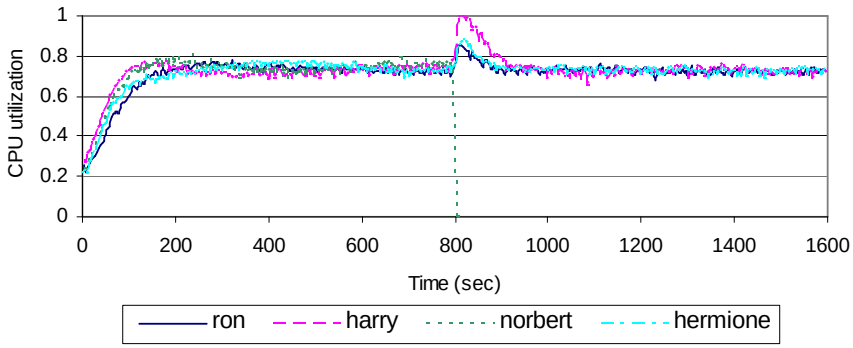



Figure 7.9: CPU utilizations of all processors while Norbert has a system failure ($ietf = 2$)

### 7.3.6 Experiment V: Overhead

The utilization control service necessarily introduces overhead. This overhead is caused by several factors including the timers associated with FC-ORB, the utilization monitoring, the control computation, the rate enforcement and the thread priority adjustment. Utilization control is a viable middleware service only if the overhead it introduces is sufficiently low. To measure the overhead accurately, we adopt a time stamping approach. Firstly, we differentiate all control service related code from other FC-ORB code. Then, time stamps are taken at the starting point and at the finishing point of each segment of the control service code to get the execution time of the control service. Since the utilization control service runs at the highest Linux real-time priority, the code segment between two timestamps will not be preempted during its execution. Hence, the time-stamped result accurately reflects the real execution overhead.

To achieve fine grained measurements, we adopt a nanosecond scale time measuring function called *gethrtime*. This function uses an OS-specific high-resolution timer that returns the number of clock cycles since the CPU was powered up or reset. The *gethrtime* function has a low overhead and is based on a 64 bit clock cycle counter on Pentium processors. With the clock counter number divided by the CPU speed, we can get reasonably precise and accurate time measurements.

Table 7.1 lists the average and standard deviation of the overhead of the utilization monitor, the actuator (including the rate modulator and the priority adjuster) and

Table 7.1: Overhead of utilization control

| | *Monitor (ms)* | | *Actuator (ms)* | | *Controller (ms)* | |
|---|---|---|---|---|---|---|
| *Processor* | *Avg* | *Dev* | *Avg* | *Dev* | *Avg* | *Dev* |
| Ron | 0.090 | 0.013 | 19.078 | 18.160 | | |
| Harry | 0.096 | 0.013 | 34.389 | 33.305 | | |
| Norbert | 0.094 | 0.012 | 39.460 | 37.223 | | |
| Hermione | 0.088 | 0.013 | 27.924 | 25.951 | | |
| Controller | | | | | 5.765 | 0.219 |

the controller of the utilization control service. All results in the table are obtained from over 600 continuous sampling periods. The overhead of the utilization monitor is very low because it just executes around 20 lines of code to read the utilization data from the Linux system file */proc/stat*.

The actuator has the dominant overhead because it involves relatively more complicated operations. The rate modulator and the priority manager are the two main contributors to the actuating overhead. Our implementation uses the ACE function *reset_timer_interval* to reset the timers and the ACE function *thr_setprio* to adjust the thread priorities in FC-ORB. In most cases, only the rate modulator is invoked to adapt the task rates by adjusting the interval of the timers. In some periods when the order of the task rates has been reversed, the priority manager is invoked to adjust the priorities of the real-time threads. The overhead of adjusting thread priorities is much larger than resetting timer intervals and so the standard deviation of the actuating overhead is large.

To estimate the average computation overhead of the controller, we measure the execution time of the *lsqlin* function in the shared library which dominates the computation cost on the control processor. We call the *lsqlin* function for 1000 times as a subroutine. The result is then divided by 1000 to get the execution time of a single execution of the least square computation. As shown in Table 7.1, the overhead of the controller is stable with small deviation and its amount is between that of the monitor and the actuator. Overall, the execution time overhead of all control components in our experiments is around 46ms per sampling period, corresponding to 1.15% utilization given a sampling period of 4 seconds.

## 7.4 Summary

In summary, we have designed and implemented FC-ORB, a real-time ORB middleware with a novel end-to-end utilization control service. Our experiments on a physical testbed has shown that (1) FC-ORB can enforce desired utilizations on all processors in a DRE system, even when task execution times deviate significantly from their estimated values or vary significantly at run-time; (2) FC-ORB can survive considerable resource contention imposed by external disturbances; (3) FC-ORB enhances the robustness of real-time properties to processor failures; (4) the middleware layer instantiation of the end-to-end utilization control service only introduces a small amount of processing and memory overhead. These results demonstrate that the integration of end-to-end utilization control, fault-tolerance mechanisms, and end-to-end scheduling in ORB middleware is a promising approach to achieve robust real-time performance guarantees for DRE applications. In the future, we plan to enhance FC-ORB to incorporate other adaptation mechanisms such as admission control and task reallocation so that FC-ORB can be applied to a broader class of applications. An important research direction is to integrate FC-ORB with advanced fault detection and recovery in order to handle more complex fault models.

# Chapter 8

# Controllability and Feasibility

In previous chapters, we have developed several control algorithms and robust middleware systems for utilization control in DRE systems. Both control analysis and empirical results demonstrate that the algorithms and middleware can achieve robust utilization guarantees even when task execution times deviate significantly from the estimation or change dynamically at run-time. While previous work has shown promise, several important issues have to be addressed, in order to provide a practical solution to utilization control in real-world DRE systems.

An fundamental problem is guaranteeing system *controllability*. Controllability is an important property of DRE systems. No control algorithm (including EUCON, DEUCON or any other algorithms) can control a system if the system itself is uncontrollable. In utilization control, an uncontrollable DRE system is a system for which it is impossible to find a sequence of task rates that take the utilizations of all processors in the system to certain utilization set points specified by the applications. As a result, some processors may become overloaded while some other processors may be poorly utilized at the same time. This kind of workload unbalance and consequent deadline misses may cause very serious problems in real-time systems. Along with controllability, it is also important to investigate the *feasibility* problem, which is caused by actuation constraints (e.g., rate constraints of a DRE system). A controllable system may still fail to achieve the desired utilization set points due to its rate constraints. Therefore, both controllability and feasibility are very important system properties and have to be guaranteed for DRE systems.

The contributions of this chapter are four-fold:

- We formulate and transform the controllability and feasibility problem to an end-to-end task allocation problem.
- We design task allocation algorithms to ensure a system is controllable and robustly feasible.
- We analyze the impact of workload variations on controllability and feasibility and design efficient online algorithms to dynamically adjust task allocation.
- We present both empirical and numerical results to demonstrate the effectiveness of our algorithms.

The rest of this chapter is structured as follows. We first formulate the controllability and feasibility problems in Section 8.1. Section 8.2 analyzes the controllability problem and the impact of workload variations. Section 8.3 presents our offline task allocation algorithms. Section 8.4 presents our online allocation adjustment algorithms. Section 8.5 introduces the middleware implementation of the algorithms in the FC-ORB middleware system. Section 8.6 presents our numerical and empirical results. Finally, Section 8.7 summarizes this chapter.

## 8.1 Problem Formulations

In this section, we formulate the controllability and feasibility problems.

### 8.1.1 Controllability Problem

In a MIMO control system, if a sequence of control input variables can be found that take all control output variables from any initial conditions to any desired final conditions in a finite time interval, the MIMO system is said to be *controllable*, otherwise the system is uncontrollable [25]. According to control theory [24], a MIMO system $\mathbf{x(k+1)} = \mathbf{\Phi x(k)} + \mathbf{\Gamma v(k)}$ with $n$ control outputs $[x_1(k) \dots x_n(k)]$ and $m$ control inputs $[v_1(k) \dots v_m(k)]$ is controllable iff the rank of its *controllability matrix* $\mathbf{C} = [\ \mathbf{\Gamma}\ \ \mathbf{\Phi\Gamma}\ \ \dots\ \ \mathbf{\Phi^{n-1}\Gamma}\ ]$ is $n$, the order of the system.

**Definition** Based on the above definition of controllability, a DRE system is *controllable* if there exists a sequence of task rates that take the utilizations of all processors in the system to any utilization set points.

As described in Chapter 3, for a DRE system with $n$ processors and $m$ end-to-end periodic tasks, the system model of the end-to-end utilization control is

$$\mathbf{u(k+1)} = \mathbf{u(k)} + \mathbf{GF\Delta r(k)} \tag{8.1}$$

where $\mathbf{F}$ is the $n \times m$ subtask allocation matrix and $\mathbf{G} = diag[g_1 \dots g_n]$ and $g_i$ represents the ratio between the actual utilization and its estimation. As explained in Chapter 4, matrix $\mathbf{G}$ is assumed to be $diag[1 \dots 1]$ because system gains are unknown at design time. We will show later that system gains do not affect system controllability. Hence, the controllability matrix of the system model is an $n \times nm$ matrix $\mathbf{C} = [\ \mathbf{F} \quad \mathbf{F} \quad \dots \quad \mathbf{F}\ ]$.

Based on the above analysis, in order to have a controllable DRE system, we have to guarantee the rank of its controllability matrix is $n$, the number of processors in the system.

### 8.1.2 Feasibility Problem

While controllability is an important property of DRE systems, it alone is not enough. As introduced in our task model, the rate of each task $T_i$ can only be adjusted within a range $[R_{min,i}, R_{max,i}]$, namely $R_{min,i} \leq r_i \leq R_{max,i}, (1 \leq i \leq m)$. However, in control theory, the condition of controllability is derived with the assumption that there is no actuation constraints (i.e. rate constraints). Therefore, a system proved to be controllable may still not be able to achieve the desired utilization set points, as the task rates may saturate at the boundaries of the rate ranges.

**Definition** If a controllable DRE system cannot get to the set points because the rates of one or more of its tasks saturate at the rate boundaries, we say it is *infeasible* to achieve the set points for the system. Otherwise we say utilization control is *feasible* for the system.

An effective solution to the feasibility problem is subtask allocation adjustment. For instance, if a processor in the system remains overloaded because all its subtasks already reach their lower rate boundaries, we may move one subtask away from the processor so it can have less workload and then recover from the overload. While this solution is sufficient for systems where execution times never change, it has to be extended because execution times may vary unpredictably in real DRE systems. As a result of the variations, a previously feasible system may become infeasible at runtime. Since execution time variations are unpredictable, it would introduce large runtime overhead to continuously monitor feasibility and migrate subtasks. Hence, instead of guaranteeing a system to be feasible for certain execution times, we try to increase the probability of the system being feasible even under variations. The higher the probability, the less the necessity of moving subtasks later at runtime.

We first introduce several definitions.

**Definition** The *maximum estimated utilization* of processor $P_i$ is defined as the summation of the products of the estimated execution times and the maximum allowed rates of all subtasks on the processor. Specifically, $u_{max,i} = \sum_{T_{jl} \in S_i} c_{jl} R_{max,j}$, where $S_i$ represents the set of subtasks located at processor $P_i$. Similarly, the *minimum estimated utilization* of processor $P_i$ is defined as $u_{min,i} = \sum_{T_{jl} \in S_i} c_{jl} R_{min,j}$.

**Definition** The difference between the set point of processor $P_i$ and its maximum estimated utilization is defined as its *upper margin*. Specifically, $margin_{upper} = u_{max,i} - B_i$. Similarly, the difference between the set point and the minimum estimated utilization is defined as $P_i$'s *lower margin*, namely $margin_{lower} = B_i - u_{min,i}$.

To increase the feasibility probability, when the variations of execution times cause the utilization of $P_i$ to deviate from its set point $B_i$, we hope there is enough margin that allows task rates to adapt so that the utilization can reconverge to the set point. Hence, we want to adjust subtask allocations so that the task rates can stay as far away from their boundaries as possible when processors settle at their set points. In other words, we want to maximize both upper margin and lower margin for all processors in order to maximize the probability of having a feasible system under variations. In real DRE systems, however, the lower margin is usually more important because overload

is highly undesirable in DRE systems. In contrast, underutilization is typically not a problem as it does not cause deadline misses or system crash. Therefore, in the following, we focus on *practical feasibility* instead of the general feasibility defined before.

**Definition** A DRE system is *practically feasible* if its task rate constraints allow the utilizations of all processors to either get to the desired set points or stay below the set points.

**Definition** As the lower margin has significant influence on the practical feasibility of a DRE system under execution time variations, we define it as the *feasibility margin* of the system.

If we assume that all the processors in the system have the same probability for execution time variations, the feasibility problem becomes a problem of maximizing the smallest feasibility margin among all processors in the system. Hence, the feasibility problem can be formulated as finding a subtask allocation to optimize the following objective.

$$\max(\min_{1\leq i\leq n}(|B_i - u_{min,i}|)) \tag{8.2}$$

This optimization problem is subject to two constraints. The first one is utilization constraint. The minimum estimated utilization $u_{min,i}$ of each processor $P_i$ is not allowed to be larger than $B_i$, because the system is infeasible in that case. The second one is resource constraint. As a common practical issue in DRE systems, each subtask can only be allocated to a specific set of processors due to resource availability. In addition, when the system is scheduled by some algorithms like RMS [54], the set point $B_i$ of each processor $P_i$ is commonly a function of its number of subtasks and so may vary when subtask allocation changes.

## 8.2 Controllability Analysis

In this section, we investigate several important aspects of controllability including the condition for a system to be controllable, structural controllability and the impact of common workload variations on system controllability.

### 8.2.1 Controllability Condition

We first analyze the controllability matrix to see how we can guarantee its rank to be equal to the number of processors in the system.

**Theorem 8.2.1** *A DRE system is controllable if and only if the rank of its allocation matrix* $\mathbf{F}$ *is* $n$.

**Proof:** We prove that the rank of the subtask allocation matrix $\mathbf{F}$ is equal to the rank of the controllability matrix $\mathbf{C} = [\ \mathbf{F} \quad \mathbf{F} \quad \ldots \quad \mathbf{F}\ ]$. We first transform $\mathbf{C}$ to a matrix $\mathbf{C}' = [\ \mathbf{F} \quad 0 \quad \ldots \quad 0\ ]$ by subtracting every column of the first $\mathbf{F}$ from the rest $\mathbf{F}$'s. Since elementary transformations do not change the rank of a matrix, $\mathbf{C}$ has the same rank as $\mathbf{C}'$. Clearly, $\mathbf{C}'$ has the same rank as $\mathbf{F}$. Hence, the system is controllable if and only if the rank of $\mathbf{F}$ is $n$. ■

**Example** The DRE system shown in Figure 7.1 is not controllable because the rank of its subtask allocation matrix $\mathbf{F}$ is 2, while the order of the system is 3 (3 processors). To be specific,

$$Rank(\mathbf{F}) = Rank(\begin{bmatrix} c_{11} & c_{21} \\ c_{12} & c_{22} \\ c_{13} & 0 \end{bmatrix}) = 2 \tag{8.3}$$

An observation from the above example is that a DRE system with $n$ processors and $m$ end-to-end periodical tasks is *uncontrollable* if $m < n$. In other words, any DRE system must have more tasks (control inputs) than processors (control outputs) in order to be controllable. Note that $m > n$ is a necessary but not sufficient condition

of controllability. When this condition is met, a system is not necessarily controllable. However, as we will show later, we can always adjust the subtask allocation matrix of the system to make it controllable. Hence, similar to the feasibility problem, the controllability problem has also been transformed to a subtask allocation problem.

### 8.2.2 Structural Controllability

As the algorithms we are proposing are used in DRE systems, here we narrow down our attention from *complete controllability* (i.e. controllability defined before) to *structural controllability* [86]. A system is structurally controllable if there exists another system which is structurally equivalent to the system and is completely controllable [86]. Two systems are structurally equivalent if there is a one-to-one correspondence between the locations of the fixed zeros and nonzero items in their controllability matrices [86].

A structurally controllable system may not be completely controllable because specific numbers could make two rows/columns of its controllability matrix become proportional so its rank is smaller than the system order. In our system model, two rows are proportional means that the subtasks on two processors belong to exactly a same set of tasks and the execution times of corresponding subtasks are strictly proportional to each other. Two columns are proportional means that two tasks are deployed on exactly a same set of processors and the execution times of their subtasks on a same processor are strictly proportional to each other. Although there is a very small probability that those situations may happen in real DRE systems, it is not worth the unbounded computation overhead introduced by existing complex algorithms [86]. Instead, we adopt a conservative way to handle those situations by treating proportional rows/columns as a single row/column. We assume that we know two tasks/processors have proportional workload beforehand and so we require the controllability matrix to have full rank even after removing any proportional rows/columns. The conservative solution is sufficient for a structurally controllable system to be controllable. In the following, we focus on structural controllability and we use controllability to mean structural controllability.

Table 8.1: Impact of different workload variations

| *Variations* | *Feasibility* | *Controllability* |
|---|---|---|
| Task arrival | harmful | harmless |
| Task termination | harmless | harmful |
| Processor failure | harmless | harmful |
| Exec time variation | harmful | harmless |

### 8.2.3 Impact of Workload Variations

In real DRE systems, workload variations often happen and may change subtask allocations which in many ways affect system feasibility or controllability. Hence, it is necessary to investigate their possible impact on system feasibility and controllability. In this dissertation, we focus on four common workload variations: task arrival and task termination, processor failure, and execution time variation. In the following, we analyze the possible impact of each type of variation on the two important system properties. If a type of variation does not affect feasibility or controllability, we define it as a *harmless* variation to feasibility or controllability. Otherwise we say it is *harmful*. The categorization of harmless and harmful variations allows us to execute our runtime adjustment algorithms only when harmful variations happen, so we can minimize the runtime overhead.

We investigate feasibility first by finding which types of variation may reduce the feasibility margin of a system. Clearly, any variations that increase system workload may cause the feasibility margin to decrease. Therefore, execution time variation, task arrival are harmful to system feasibility because they may increase the workload of some processors in the system. Task termination reduces the workload of some processors so it is harmless. Processor failure causes task termination so it is also a harmless variation to system feasibility. The impact of different workload variations on feasibility is summarized in Table 8.1.

The focus of our impact investigation is on controllability. First, controllability is a more fundamental property of DRE systems. According to the definition of feasibility, a system needs to be controllable first in order to be feasible. Second, unlike the feasibility problem for which we have a margin to tolerate workload variations to some extent, any variation may affect controllability because the variation may change the

rank of the allocation matrix. Third, the analysis of impact on controllability is less intuitive than that on feasibility. As discussed in previous section, we focus on structural controllability as it is more realistic in DRE systems.

**Theorem 8.2.2** *Dynamic task arrival in a DRE system is harmless to controllability.*

**Proof:** Dynamically adding an end-to-end task in a DRE system is equivalent to adding a new column to the subtask allocation matrix $\mathbf{F}$, which does not reduce the rank of $\mathbf{F}$. ■

Therefore, if the system is controllable, it is still controllable after the arrival. If the system is uncontrollable, it may become controllable after the arrival. However, the rank of $\mathbf{F}$ has to be recalculated in that case.

**Theorem 8.2.3** *Dynamic task termination in a DRE system is harmful to controllability.*

The proof is straightforward. Removing a column from the allocation matrix may reduce the rank of the matrix.

**Theorem 8.2.4** *Processor failure is harmful to controllability if the failed processor has more than* $m - n + 2$ *subtasks.*

**Proof:** Removing a processor from a DRE system is equivalent to removing a row from the subtask allocation matrix $\mathbf{F}$. Without any task migration mechanism, all tasks having subtasks on the failed processor may terminate. The termination is equivalent to removing several columns from the allocation matrix. If the rank of matrix $\mathbf{F}$ is originally $n$, any of its submatrices with size as $n' \times m'$ has the rank as $\min(n', m')$. So after the processor failure, the allocation matrix has its rank as $\min(n-1, m')$. In order for the matrix to have a rank less than $n-1$, we need to have $m' \leq n-2$. Hence, we need to terminate at least $m - m' = m - n + 2$ tasks. ■

Although processor failure is harmless to controllability when the condition is not met, we still list it as harmful in Table 8.1 to be more general.

**Theorem 8.2.5** *Proportional execution time variations of all subtasks on one or more processors are harmless to controllability.*

**Proof:** Proportional execution time variations can be modeled as the system gain variations which are represented by the matrix $\mathbf{G}$ in the system model (8.1). Since $\mathbf{G}$ is a diagonal matrix, multiplying $\mathbf{G}$ with matrix $\mathbf{F}$ is equivalent to using $g_1, \ldots, g_n$ to multiply the corresponding line of matrix $\mathbf{F}$. These elementary transformations do not change the rank of $\mathbf{F}$. ■

Theorem 8.2.5 guarantees system controllability when the execution time variations of all subtask on one or more processors can be approximated as proportional. In addition to that, as we focus on structural controllability in this dissertation, execution time variation is harmless to controllability. The impact of different workload variations on controllability is also summarized in Table 8.1.

## 8.3 Offline Task Allocation Algorithms

Both controllability and feasibility problems require us to develop novel subtask allocation algorithms for DRE systems. We need to ensure controllability at the last step of the algorithms, because any allocation adjustment resulted from increasing feasibility margin could make a controllable system uncontrollable. On the other hand, however, a careful adjustment for controllability may affect feasibility margin only slightly. Therefore, in this section, we first introduce an algorithm to effectively increase the system feasible margin. Based on the subtask allocation generated by the feasibility algorithm, we then present another algorithm to ensure controllability while minimizing its effect on feasibility margin. Note that the two algorithms are integrated as a complete solution.

### 8.3.1 Feasibility

As suggested by Equation 8.2, the feasibility problem is related to both load balancing [6] and variable-size bin packing [55]. It is related to the variable-size bin packing problem because it needs to pack all subtasks to processors and the capacity of a processor shrinks when its number of subtasks increases. It differs from bin packing because the goal here is to balance the workload on each processor, instead of using fewest possible processors. The problem is closer to the load balancing problem but the difference is that we are trying to maximize the smallest feasibility margin instead of minimizing the largest load. Clearly our problem can be reduced to the load balancing problem which is an NP-hard problem [6]. Here we present a feasibility algorithm which is extended from the standard Max-Min algorithm used for load balancing [6]. The Max-Min algorithm has a good trade-off between solution quality and computation overhead [6].

In our feasibility algorithm, we first sort all subtasks based on their minimum estimated utilization, $u_{min,jl} = c_{ij}R_{min,j}$. Then we pick the subtask with largest $u_{min,jl}$ and allocate it to the processor that has the largest feasibility margin after this allocation. We continue the process until all the subtasks are allocated. Note that the allocation at each step is subject to both the utilization and resource constraints. The utilization constraint is checked at each step when a subtask is allocated to a processor. If the largest feasibility margin after allocating a subtask to the system becomes negative, the algorithm fails. In that case, more advanced algorithm such as Mixed Integer Programming may be adopted to provide a solution at a cost which could have the same complexity as exhaustive search [6]. The resource constraints are represented by an $s \times p$ matrix $cons$, where $s$ is the total number of subtasks in the system and $p$ is the number of processors on which a subtask can execute. Each element $cons[T_{jl}, q]$ is the $q^{th}$ processor that the subtask $T_{jl}$ can be allocated to. The detailed algorithm is shown in the below pseudo code. We assume all processors are homogeneous here, but the algorithm can be easily extended to the systems with heterogeneous processors.

```
(1) sort all subtasks T_jl based on u_min,jl and enqueue them in decreasing order
(2) while there is at least one subtask in the queue
      pop up the first subtask T_jl (which has the largest u_min,jl)
      for each processor P_q = cons[T_jl, q++]
        if u_current,q + u_min,jl ≤ B_q
          u_new,q = u_current,q + u_min,jl
           feasibility margin of P_q = B_q − u_new,q
        end if
      end for
      allocate subtask T_jl to the processor P_i with the largest feasibility margin
      if T_jl cannot be allocated to any processor
        algorithm fails
    end while
```

Now we analyze the complexity of this algorithm. The complexity of step 1 is $O(s \log s)$, where $s$ is the total number of subtasks in the system. The complexity of step 2 is $sp$, where $p$ is the number of processors that a subtask can be allocated to. Hence, the time complexity of the feasibility algorithm is $O(\max(s \log(s), sp))$.

### 8.3.2 Controllability

After our feasibility algorithm successfully allocates all subtasks, we can check the allocation matrix **F** to determine whether the current workload configuration is controllable. If it is controllable, the workload is accepted for deployment on the target DRE system. Otherwise we process the workload with a controllability adjustment algorithm to make the uncontrollable system controllable. In the controllability algorithm, for every processor, we search all tasks which have subtasks on the processor to find one task to dedicate to the processor. The task is called the *dedicated task* of the processor and its subtasks on the processor are called the *dedicated subtasks*. A task can only be dedicated to one processor. For those processors which fail to find dedicated tasks, we migrate subtasks of some non-dedicated tasks from other processors to them so they can have those tasks dedicated to them.

**Theorem 8.3.1** *If every processor in a system has dedicated task, the system is controllable.*

**Proof:** If every processor has a dedicated task, the allocation matrix can be proved to have full rank (i.e. its rank equals the order of the system). The proof is straightforward. We can move the columns of the matrix so that all tasks can place their dedicated subtasks on the diagonal of the allocation matrix. If we assume there is no two rows/columns which are proportional to each other in the matrix, as defined for structural controllability, a matrix has full rank if there is no zero on its diagonal. In other words, a system is guaranteed to have structural controllability if every processor has dedicated task. ■

Note that Theorem 8.3.1 is both a sufficient and a necessary condition for structural controllability. The rationale behind dedicating tasks to processors can also be explained from system perspective, each processor can rely on the rate adaption of its dedicated task to achieve its utilization set point, if we assume there is no rate constraints so task rates can be any value (even negative ones).

In our controllability algorithm, we first sort all processors based on their numbers of subtasks. We try to start dedicating tasks to the processors with fewer subtasks first, because that may reduce the necessity of moving subtasks later on. The second step is used to preprocess the allocation matrix for the later dedicating step. For every processor/task pair in the allocation matrix, we search for a candidate subtask by assuming the processor fails to find its dedicated task and needs a subtask of the task to be moved to the processor. Since subtask migration may affect the feasibility margin of a system, we want to minimize the impact by moving the *best candidate subtask* which has the smallest minimum estimated utilization and is allowed by the resource constraints to run on the target processor. Hence, for every element (i.e. processor/task pair) in the allocation matrix $F$, we add some piggyback information such as the location of the best candidate. The information will speed up the search process if a processor loses its dedicated task and needs to find a new one at runtime. In the third step, we sort all existing subtasks on each processor based on their minimum estimated utilization. For those previous zero elements (i.e. no subtask exists there), we sort them based on the minimum estimated utilizations of their best candidate subtasks. The reason for sorting them is also to speed up the search process, which is especially important for an online solution described in Section 8.4. Finally in the last step, we start our dedicating process. If no task can be dedicated

to a processor, we move the best candidate subtask of the first non-dedicated task to the processor. This subtask is guaranteed to have the smallest minimum estimated utilization and so should only cause small impact on the system feasibility margin.

The detailed controllability algorithm is shown in the below pseudo code.

```
(1) In the allocation matrix F, replace all zero elements with maximum integer
    sort all processors with increasing number of subtasks

(2) for each subtask T_jl in resource constraints matrix cons
        for each of its allowed processor P_q
            F(q, j) = min{u_min,jl, F(q, j}
            best candidate subtask of F(q, j) = T_jl
        end for
    end for

(3) for each processor in the allocation matrix F
        for all existing subtasks
            sort their subtasks in the decreasing order of u_min,jl
        for all previous zero elements
            sort their best candidate subtasks in the increasing order of u_min,jl
    end for

(4) for each processor P_i in the allocation matrix F
        for each task T_j already having subtasks on P_i (in decreasing order)
            if T_j is non−dedicated, dedicate T_j to P_i, end if
        end for
        if all tasks are already dedicated to other processors
            for each previous zero element (in the increasing order of u_min,jl)
                if the task is non−dedicated
                    move the best candidate subtask to P_i
                    dedicate the task to P_i
                end if
            end for
            if cannot find a non−dedicated task, algorithm fails , end if
        end if
    end for
```

Now we analyze the time complexity of this algorithm. The complexity of the four steps are $O(n \log n)$, $O(sp)$, $O(nm \log m)$ and $O(nm)$, respectively. Hence, the time complexity of the whole controllability algorithm is $O(\max(sp, nm \log m))$.

## 8.4 Online Allocation Adjustments

Even though the algorithms presented in the previous section can effectively preprocess workloads before deployment to increase feasibility margin and guarantee controllability, there are two issues we have to address. First, as the subtask allocation matrix may change at runtime due to workload variations such as task termination, a workload processed with the above algorithms may still become uncontrollable or infeasible. Hence, controllability and feasibility have to be maintained at runtime as well. Second, as analyzed in the previous section, the feasibility algorithms introduce some computation overhead. While it is acceptable to run the two algorithms to preprocess a workload before it is deployed to DRE systems, we need to develop more efficient ones to incrementally adjust workload at runtime.

In this section, we present online versions of our algorithms to adjust subtask allocations incrementally when certain variations happen to the system, at just a small portion of the cost of the previous algorithms.

### 8.4.1 Feasibility Adjustment

According to Table 8.1, two variations may reduce the feasibility margin of a system. In DRE systems, execution time variations are commonly unpredictable and costly to monitor online. Therefore, as we introduced before, the feasibility margin is used to tolerate possible execution time variations, so we do *not* need to address the variations at runtime. Hence, we focus on how to adjust workload incrementally online to minimize the impact of task arrivals on feasibility, at an acceptable cost.

Even though we may run our offline feasibility algorithm to reallocate all subtasks every time when we have new tasks coming to the system, the large computation

and migration overhead makes it impossible to do so at runtime. Here we run our feasibility algorithm only to those new tasks to have a good trade-off between feasibility margin and runtime overhead. The algorithm presented in previous section will be adopted to sort and allocate only those new arriving tasks. Hence, the computation overhead is now only $O(\max(qn\log(qn), qnp))$, where $q$ is the number of arriving tasks.

### 8.4.2 Controllability Maintenance

According to the previous theorems, there are two situations that may jeopardize the controllability of the system: task termination and processor failure. The reason that processor failure is harmful is that they may cause one or more tasks to terminate. Hence, we only need to check and maintain controllability when tasks terminate, which can be handled incrementally by the following runtime task reallocation algorithm:

```
(1) remove the terminated task from the allocation matrix
(2) if this task is not dedicated to a processor
        algorithm successfully ends
(3) else
        for the processor that the terminated task was dedicated to
            run step (4) of the offline controllability algorithm to find a dedicated task for it
    end if
```

The time complexity of the controllability maintenance algorithm is $O(m)$, where $m$ is the number of tasks in the system.

## 8.5 Middleware Implementation

Both the controllability and feasibility algorithms have been implemented in the FC-ORB middleware system [96]. FC-ORB implements an end-to-end utilization control algorithm and is therefore an ideal platform to demonstrate the importance of controllability and feasibility in DRE systems. The two algorithms are integrated with

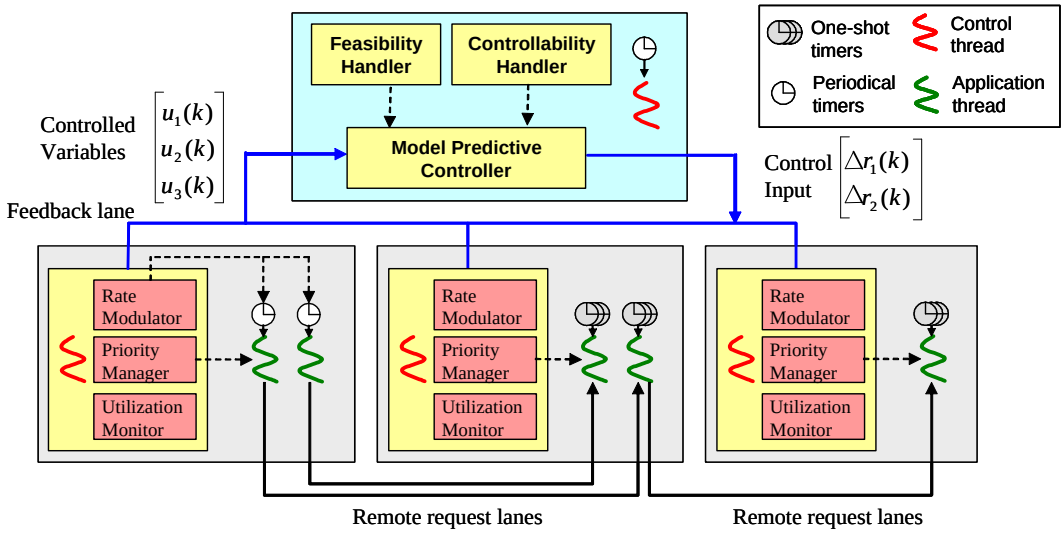



Figure 8.1: Middleware architecture of the extended FC-ORB system

the FC-ORB controller which is running on a different processor from the controlled system. The middleware architecture of the extended FC-ORB system is shown in Figure 8.1.

The controllability maintenance algorithm is implemented as a controllability handler. Based on our analysis in the previous section, only task termination affects the controllability of a system. Consequently, the controllability handler is invoked whenever one or more task terminate at runtime. When that happens, the handler removes the terminated tasks from the control model, and then moves proper subtasks to maintain system controllability. After that the handler re-initializes the controller and resumes the feedback control loop. Similarly, the feasibility adjustment algorithm has been implemented as a feasibility handler to do incremental subtask allocation whenever new tasks are admitted to the system.

The middleware part of FC-ORB is also extended to handle subtask migrations demanded by the controller and dynamic task arrivals. The migration mechanism works as follows. Each subtask can have a primary instance and a few backup instances on the processors where it has the required resource. In the normal mode, each subtask pushes remote operation requests only to the primary instance of its successor. As a result, the backup instances do not receive any requests and their threads remain

idle. After a task migration decision is made by the controller, the predecessor of the migrated subtask switches the connection to the desired backup instance and sends the remote operation requests to it. In the case when the first subtask of a task has to be moved, the controller activates the proper backup instance of the subtask. Consequently, the execution of the end-to-end tasks is resumed after a transient interruption. Task arrivals are handled as dynamic invocation of specified object functions in the existing FC-ORB middleware system.

## 8.6 Experiments

In this section, we present the results of two sets of experiments. First, numerical experiments are used to evaluate the performance of the offline subtask allocation algorithms introduced in Section 8.3. The numerical experiments allow us to use a large number of randomly generated workloads to stress-test our algorithms. Second, empirical results based on the extended FC-ORB middleware system are presented to demonstrate the effectiveness of the dynamic adjustment and maintenance algorithms proposed in Section 8.4. As system feasibility and controllability may change due to workload variations, it is important to investigate how a real DRE system behaves when it becomes uncontrollable or infeasible at runtime. We then show how our algorithms work to make the system controllable and feasible again on the fly.

### 8.6.1 Numerical Results

In all experiments presented in this subsection, the number of tasks has been fixed at 50 (i.e. $m = 50$), while the number of subtasks of each task is varied uniformly from 1 to 7. For each task, its lower rate bound, $R_{min,j}$, is randomly generated between 0.01 and 0.1Hz. For each subtask, its minimum estimated utilization varies randomly between 5% and 15% and its execution time is calculated based on its rate and its minimum estimated utilization. Each subtask can only be executed on 5 processors (i.e. $p = 5$), which are randomly chosen from all processors in the system. Because any system with more processors than tasks is uncontrollable, we vary the number of processors (i.e. $n$) from 35 to 50 to examine the performance of our algorithms

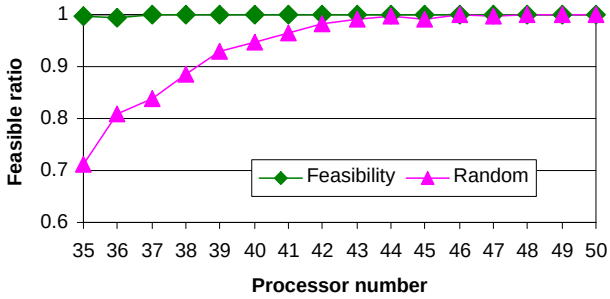



Figure 8.2: Feasible ratio under different processor numbers

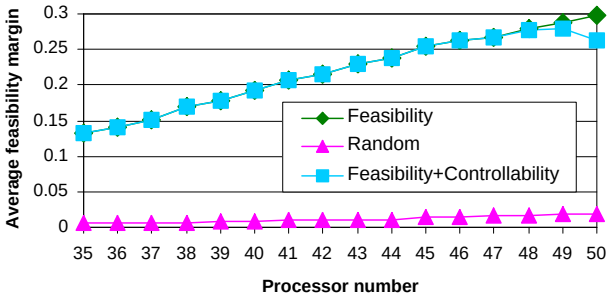



Figure 8.3: Feasibility margin under different processor numbers

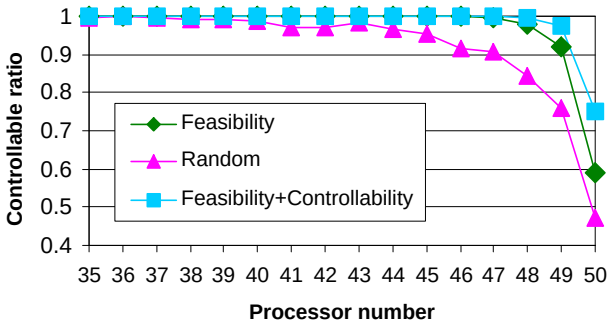



Figure 8.4: Controllable ratio under different processor numbers

while the average number of subtasks on each processor changes. For each value of $n$, 500 different workload configurations are randomly generated and tested. The schedulable utilization bound of RMS[54], namely $B_i = m_i(2^{1/m_i} - 1)$, is used as the utilization set point of each processor $P_i$, where $m_i$ is the number of subtasks on this processor.

We compare our algorithms against a baseline algorithm called Random. Random first ensures there is no *idle* processor by randomly allocating one subtask to each

processor, because otherwise the system is clearly uncontrollable. Then the rest subtasks are also randomly allocated to processors under the utilization constraints and resource constraints. When a subtask cannot be allocated to a processor due to the utilization constraint of the processor, the subtask is randomly allocated to another processor. If a subtask cannot be allocated to any processor, the algorithm fails.

We first examine the *feasible ratio* (i.e. the fraction of task allocations that are feasible) under our feasibility algorithm and Random by applying them to the randomly generated workloads. A workload resulted from an allocation is *feasible* if the minimum estimated utilizations of all processors are equal to or lower than their schedulable bounds. Figure 8.2 shows that the feasibility algorithm achieves higher feasibility ratio than Random when the number of processors is smaller than 44. For example, when processor number is 35, more than 30% of workloads are not feasible under Random, while the ratio is only 1% under the feasibility algorithm. The reason is that when the number of processors is small, each processor has more subtasks, which decreases the probability for Random to find feasible solutions.

As discussed in section 8.1.2, the main goal of our feasibility algorithm is to increase the feasibility margin. Figure 8.3 plots the average feasibility margin of 250 workloads which are feasible under both the feasibility algorithm and Random. The average feasibility margin under Random is much smaller than that under the feasibility algorithm. That means the feasibility algorithm results in workloads which can tolerate much more execution time variations. For example, with 48 processors, the workload generated by the feasibility algorithm can remain feasible even when the task execution times increase by 28%. When the number of processors increases, the difference becomes larger. That is because when each processor has fewer subtasks, the space for the feasibility algorithm to improve becomes larger.

We then compare the *controllable ratio* (i.e. the fraction of task allocations that are controllable) under Random, the feasibility algorithm and the integrated feasibility and controllability algorithm. Same as before, Random and the feasibility algorithm are applied to all randomly generated workloads without any concern of controllability. In contrast, the integrated algorithm adopts the controllability algorithm introduced in Section 8.3.2 to reallocate the subtasks if the workload processed by

the feasibility algorithm is diagnosed to be uncontrollable. Figure 8.4 shows that the controllability algorithm reduces the uncontrollable cases significantly. For example, with 50 processors, the controllable ratio has been increased more than 10%. In addition, the feasibility algorithm can also help improve controllability as its controllable ratio is much higher than Random.

As discussed in Section 8.3, the controllability algorithm will have some impact to the feasibility margin though the algorithm is designed to minimize the impact. Figure 8.3 shows the impact is only roughly 3%. This result demonstrates that the controllability algorithm can improve system controllability significantly only at negligible cost of feasibility margin.

### 8.6.2 Empirical Results

In this subsection, we present the experiments conducted on a real DRE system implemented based on the extended FC-ORB middleware. We first introduce the experimental configurations. Then we present the experimental results on controllability and feasibility, respectively by contrasting systems with and without the dynamic algorithms.

#### Experimental set-up

We perform our experiments on a testbed of six PCs. All applications and the ORB service run on four Pentium-IV machines ($P_1$ to $P_4$) and one Celeron machine ($P_5$). $P_1$ and $P_4$ are 2.80GHz while $P_2$ and $P_3$ are 2.53GHz. $P_1$ to $P_4$ all are equipped with 512KB cache and 512MB RAM. $P_5$ is 1.80GHz and has 128KB cache and 512MB RAM. All machines run RedHat Linux 2.4.22. The controller is located on another Pentium-IV 2GHz machine with 512KB cache and 256MB RAM. The controller machine runs Windows XP Professional with MATLAB 6.0. $P_1$ to $P_4$ are connected via an internal switch and communicate with $P_5$ and the controller machine through the departmental 100Mbps LAN.

Table 8.2: Workload parameters

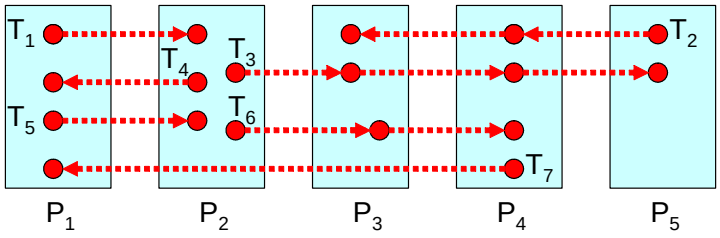
| *Subtask* | *Estimated Execution Time (ms)* | | | | *Initial Rate* | *Min Rate* | *Max Rate* |
|---|---|---|---|---|---|---|---|
| $T_{1,\{1,2\}}$ | 38 | 11 | | | 23.96 | 20 | 60 |
| $T_{2,\{5,4,3\}}$ | 22 | 28 | 43 | | 29.99 | 20 | 60 |
| $T_{3,\{2,3,4,5\}}$ | 14 | 20 | 12 | | 19.40 | 5 | 30 |
| $T_{4,\{2,1\}}$ | 25 | 24 | | | 12.43 | 10 | 60 |
| $T_{5,\{1,2\}}$ | 33 | 26 | | | 26.15 | 5 | 75 |
| $T_{6,\{2,3,4\}}$ | 26 | 16 | 21 | | 16.97 | 5 | 20 |
| $T_{7,\{4,1\}}$ | 16 | 12 | | | 44.05 | 20 | 60 |
| $T_{8,new}$ | 23 | 32 | | | 10.00 | 10 | 60 |
| $T_{9,new}$ | 27 | 19 | | | 10.00 | 10 | 60 |
| $T_{10,new}$ | 36 | 29 | | | 10.00 | 10 | 60 |

Our experiments run a medium-sized workload that comprises 7 end-to-end tasks (with a total of 18 subtasks). Figure 8.5(a) shows how the 7 tasks are distributed on the 5 application processors. The workload parameters are detailed in Table 8.2. The subtasks on each processor are scheduled by the RMS algorithm [54]. Each task's end-to-end deadline is $d_i = n_i/r_i(k)$, where $n_i$ is the number of subtasks in task $T_i$ and $r_i(k)$ is the current rate of $T_i$. Each end-to-end deadline is evenly divided into subdeadlines for its subtasks. The resultant subdeadline of each subtask $T_{ij}$ equals its period, $1/r_i(k)$. The utilization set point of every processor is set as 0.7.[1] All (sub)tasks meet their (sub)deadlines if the desired utilization on every processor is enforced. The sampling period of the utilization control service is $T_s = 5$ seconds.

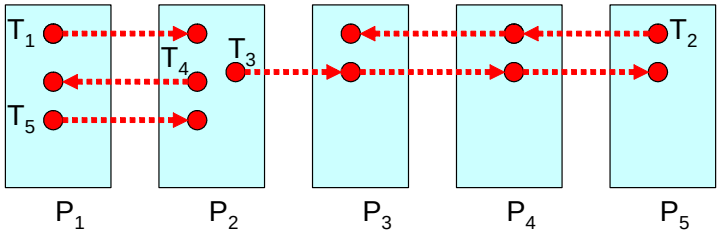
### Controllability

In our first experiment, we run the original FC-ORB with an initial workload shown in Figure 8.5(a). The rates of all tasks in the workload are selected based on their execution times so that the utilizations of all processors can be initially close to their set points. At time $300 \times 5$ seconds, task $T_6$ and $T_7$ terminate so the workload becomes uncontrollable as shown in Figure 8.5(b). From the experimental results shown in Figure 8.6, we can see that only the utilizations of processor $P_2$ and $P_5$ converge to the desired set points. The utilization of $P_1$ stays slightly below the set point. $P_4$ is

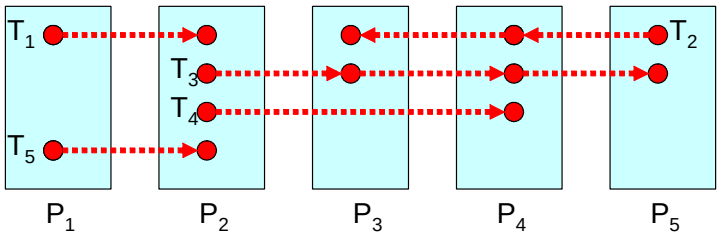
[1]The schedulable utilization bound of RMS [54], $B = m(2^{1/m} - 1)$ may be used as the utilization set point for better utilization, where $m$ is the number of subtasks on this processor.




(a) Initial configuration




(b) After task terminations




(c) After controllability maintenance

Figure 8.5: Workload configuration and variations in controllability experiments

severely underutilized as its utilization is just 50% while $P_3$ is overloaded. As processor overload may cause undeired deadline misses in a real-time system, controllability has to be maintained at runtime.

In the second experiment, we run our extended middleware system with the controllability handler activated. All configurations remain the same as in the first experiment. In the controllability analysis, task $T_7$ is not dedicated to any processor so its termination is ignored. However, task $T_6$ is dedicated to processor $P_4$ so we have to migrate a subtask to $P_4$ after $T_6$'s termination, because the two existing subtasks on $P_4$, $T_2$ and $T_3$ are already dedicated to $P_3$ and $P_5$, respectively. As an outcome of the

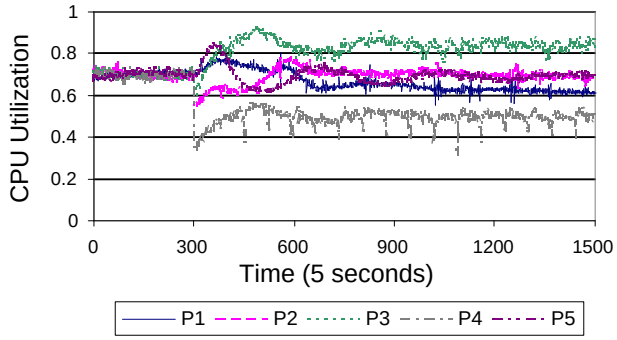



Figure 8.6: System becomes uncontrollable after task termination

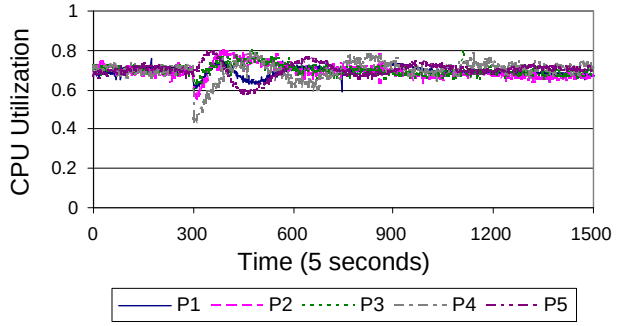



Figure 8.7: System becomes controllable after controllability maintenance

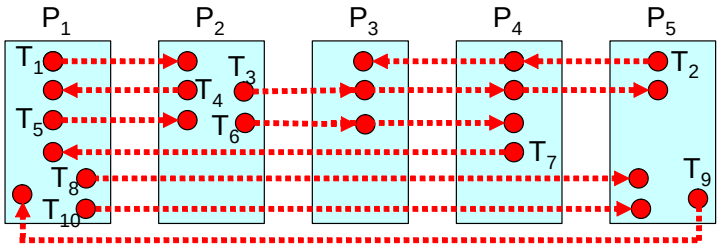
controllability maintenance algorithm, subtask $T_{4,2}$ is migrated from processor $P_1$ to $P_4$, immediately after the task terminations. From the experimental results shown in Figure 8.7, we can see that the previously uncontrollable system indeed becomes controllable again. The utilizations of all processors converge to the desired set points. Undesired processor overload or underutilization have been avoided.

### Feasibility

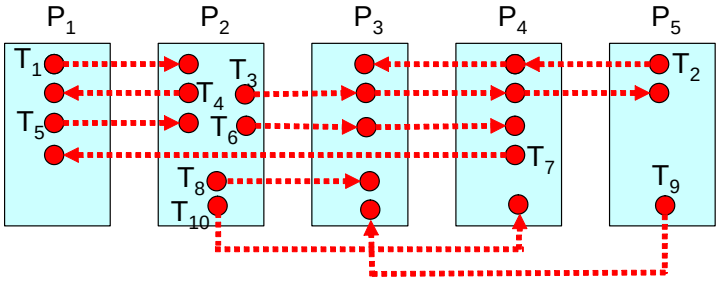
As we analyzed before, controllability maintenance alone is not enough because it may still be infeasible for a controllable system to achieve the desired utilization set points when tasks arrive at runtime. In this set of experiments, we first show

Table 8.3: Task rates of all tasks

| | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ | $T_9$ | $T_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Naive | 20 | 30.6569 | 5 | 29.9830 | 5.6836 | 20 | 50.4092 | 10 | 10 | 10 |
| Feasibility | 43.1741 | 20.6556 | 19.3107 | 11.6857 | 5.0194 | 5.0004 | 50.5294 | 10.0018 | 11.1398 | 10.0008 |




(a) Task arrival and allocation in a naive way




(b) Task allocation resulted from feasibility adjustment

Figure 8.8: Workload variations in feasibility experiments

that some naive allocations of dynamically arriving tasks make it infeasible for the original FC-ORB to achieve the set points. Same as the previous experiments, the utilizations of all processors in the system initially start from their set points. At time $300 \times 5$ seconds, three end-to-end tasks ($T_8$, $T_9$ and $T_{10}$) are admitted to the system and their details are given in Table 8.2. As an example of possible naive allocations, three subtasks are allocated to $P_1$ while the other three are allocated to $P_5$. Figure 8.9 shows that the system becomes infeasible after this allocation. The utilizations of $P_1$ and $P_5$ become higher than their set points right after the new subtasks are allocated. To reduce their utilizations, the rates of all tasks on them are decreased by the controller. However, this decrease affects the utilizations of $P_2$ to $P_4$ and causes them to be underutilized. Hence, the controller has to decide based on control theory which tasks to decrease rate and which ones to increase rate. From Figure 8.8(a), we can see that an effective way to reduce the utilizations of $P_1$ and $P_5$ without affecting other processors is to decrease the rates of the new tasks: $T_8$ to $T_{10}$. Table 8.3 shows the average task rates of all tasks in the last 50 control periods. Due to the rate

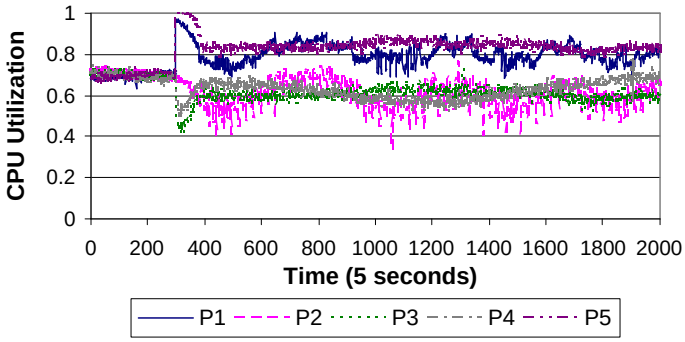



Figure 8.9: System becomes infeasible after task arrivals

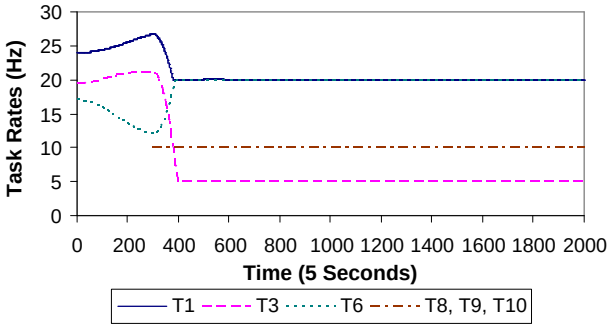



Figure 8.10: Task rates saturate at boundaries when system is infeasible

constraints shown in Table 8.2, the task rates of $T_8$ to $T_{10}$ saturate at their lower boundaries so cannot be decreased anymore. Figure 8.10 shows the task rates that saturate after the task arrivals. In addition to the new tasks, the rates of tasks $T_1$ and $T_3$ also reach their lower boundaries and so cannot be decreased anymore. On the other hand, from Figure 8.9, we can see processors $P_2$ to $P_4$ are underutilized. A possible way to increase their utilizations without affecting other processors is to increase the task rate of $T_6$. Similarly, the rate of task $T_6$ already reaches the upper boundary so cannot be increased any further. As a result of the saturations, all processors cannot achieve their desired utilization set points because it is infeasible to do so.

We then run the same experiment on our extended middleware system with the feasibility handler enabled. Whenever there are new tasks admitted to the system, the feasibility handler conducts incremental Max-Min algorithm presented in Section 8.4 to allocate the subtasks. Based on the task details shown in Table 8.2, we can get the subtask allocation shown in Figure 8.8(b). From Figure 8.11, we can see that the new tasks first have a smaller impact on the utilizations of the processors in

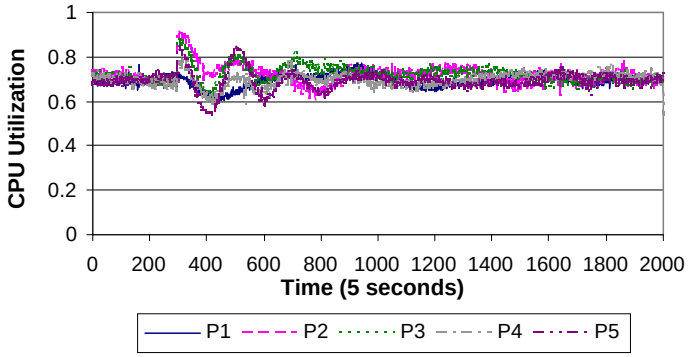



Figure 8.11: System remains feasible after feasibility adjustment

the system, compared to the naive solution. That is because the feasibility handler distributes the impact to different processors. As demonstrated by Figure 8.11, even though the same task rate constrains exist, the system still can achieve the desired utilization set points thanks to feasibility adjustment. Table 8.3 shows that none of the tasks saturate at their rate boundaries. Hence, with feasibility adjustment, it becomes feasible for a previously infeasible system to achieve the desired set points.

## 8.7 Summary

Robustness and performance of the utilization control depends crucially on the end-to-end task configuration. Specifically, task allocation affects the system controllability and the feasibility of the constrained optimization problem. In this chapter, we have presented task allocation algorithms for deploying end-to-end tasks that ensure that the system is controllable and robustly feasible. Further, we have developed runtime algorithms that maintain controllability by reallocating tasks dynamically in response to task termination and arrival. We have evaluated the performance of our approach for end-to-end task deployment using numerical experiments in large systems. In addition, we have reported empirical results for the runtime task reallocation algorithm on an experimental test-bed. Our results demonstrate that the proposed task allocation algorithms improve the robustness of utilization guarantees in distributed real-time embedded systems.

# Chapter 9

# CAMRIT: Control-based Real-Time Image Transmission

Previous chapters have shown the successful application of our adaptive QoS control framework to the end-to-end utilization control problem. In this chapter, we introduce the application of our control framework on another category of real-time application: real-time image transmission which is crucial to an emerging class of distributed embedded systems operating in open network environments. Examples include avionics mission re-planning over Link-16, security systems based on wireless camera networks, and online collaboration using camera phones. Meeting image transmission deadlines is a key challenge in such systems due to unpredictable network conditions.

In this chapter, we present CAMRIT, a Control-based Adaptive Middleware framework for Real-time Image Transmission in distributed real-time embedded systems. CAMRIT features a distributed feedback control loop that meets image transmission deadlines by dynamically adjusting the quality of image tiles. We derive an analytic model that captures the dynamics of a distributed middleware architecture. A control theoretic methodology is applied to systematically design a control algorithm with analytic assurance of system stability and performance, despite uncertainties in network bandwidth. Experimental results demonstrate that CAMRIT can provide robust real-time guarantees for a representative application scenario.

## 9.1 Introduction

Recent years have seen rapid growth of a new generation of Distributed Real-time Embedded (DRE) systems that integrate digital imaging and wireless networking technology. For example, security systems can perform automatic intruder detection through real-time fusion of images from multiple cameras connected through a wireless network [69]. Similarly, to facilitate avionics mission re-planning, personnel on multiple aircraft need to collaborate by exchanging target imagery and display annotations over Link-16 wireless networks [20]. Real-time image transmission is also important in new services on camera-equipped mobile phones (e.g., online collaboration and security monitoring) that rely on "live" image transmission over cellular networks.

These embedded applications are different from traditional imaging applications (e.g., online photo albums) in two ways. First, image transmission in these embedded systems is subject to stringent timing constraints. Second, although higher image quality usually improves system utility, these next-generation embedded applications can tolerate some degree of degradation in image quality. For example, late image delivery can be disastrous in a security system because it may result in a delayed security alarm. On the other hand, distributed event detection algorithms usually can maintain a desired probability of event detection even if input images are not perfect. Similarly, meeting deadlines is much more important in avionics mission re-planning than perfect image quality, as long as key target features are still distinguishable.

These emerging embedded applications are also different from traditional embedded systems, such as process control in factories. While traditional embedded systems usually operate over closed and predictable networks, these new types of embedded systems need to perform image transmission across *open* and *unpredictable* networks. For example, Link-16 is widely used for tactical communication between military aircraft, but has very limited effective bandwidth (e.g., roughly 30 to 340 Kbps divided among all aircraft communicating with a common JTIDS terminal [72]). Furthermore, network bandwidth may vary significantly during a mission due to changes in weather, terrain, and communication distance [20]. These bandwidth-constrained and unpredictable networks make real-time image transmission a challenging task.

We have developed *CAMRIT, a Control-based Adaptive Middleware for Real-time Image Transmission*. The CAMRIT project has made three main contributions to the state of the art in performance control for DRE systems.

1. *Adaptive Architecture*: We present a novel middleware architecture for *feedback-based* adaptive management of image transmission. Our architecture features a distributed feedback control loop that supports fine-grained control over the progress of image transmission by dynamically adjusting the quality factor of image tiles.

2. *Control Modeling*: We derive an analytic model that captures the dynamics of a distributed middleware architecture. Control analysis shows that CAMRIT can assure system stability and transmission latencies under a wide range of available network bandwidth.

3. *Middleware Implementation*: CAMRIT has been implemented as a middleware service based on the TAO [18] real-time CORBA object request broker so it is portable across heterogeneous platforms. Experimental results on a characteristic testbed demonstrate that CAMRIT can provide robust real-time assurance under representative application scenarios.

## 9.2 Middleware Architecture

The primary goal of CAMRIT is to complete transmitting an image from a server node to a client node within a user specified deadline. At the same time, CAMRIT aims to maximize image quality because a higher quality image usually has higher utility to the application. This requirement excludes trivial solutions such as always sending an image at the lowest quality.

To achieve both goals despite an unpredictable network, CAMRIT employs a feedback control loop that dynamically adjusts image quality based on performance feedback. CAMRIT exploits existing image compression standards that support flexible image quality. For example, the widely adopted JPEG [92] standard provides a user-specified parameter called the *quality factor* which can be any integer from 1 to 100. Since

a lower quality factor leads to a smaller image size after compression, the quality factor parameter provides a knob for controlling the time it takes to transmit an image. However, JPEG only supports a *single* quality factor for a whole image. This is insufficient for our feedback control loop, which needs to adjust the quality factor of an image dynamically during its transmission. To support such fine-grained adaptation, CAMRIT splits each image into tiles, each of which may be compressed with a separate quality factor.

CAMRIT is designed as a *middleware service* for real-time CORBA. All the tasks in CAMRIT are managed and scheduled according to the Rate Monotonic Scheduling (RMS) algorithm [54] using the Kokyu [28] dispatcher within the TAO Real Time Event Channel [33]. We note in passing that the CAMRIT architecture may also be instantiated as individual software or be integrated with other middleware.

### 9.2.1 Service Interface

An application interacts with CAMRIT's ImageTransmissionService interface, specified in CORBA IDL. The following parameters are passed to the service:

- *image_id*: An identifier (e.g., an image file name) for the requested image.
- *deadline*: The relative deadline for delivery of the image.
- *num_tile*: The number of tiles into which the image is divided. This parameter allows the application to specify the granularity of control of the image quality, with a trade-off of increased overhead for finer granularity.
- *quality_range*: The defined range of acceptable image quality. This parameter allows configuration of application-specific image quality constraints.

The CAMRIT service implementation serves to hide properties of the underlying network from the the application, particularly the variations in available bandwidth over a network, and delivers the image within the specified deadline. Figure 9.1 shows the major components of the CAMRIT architecture. We first describe the

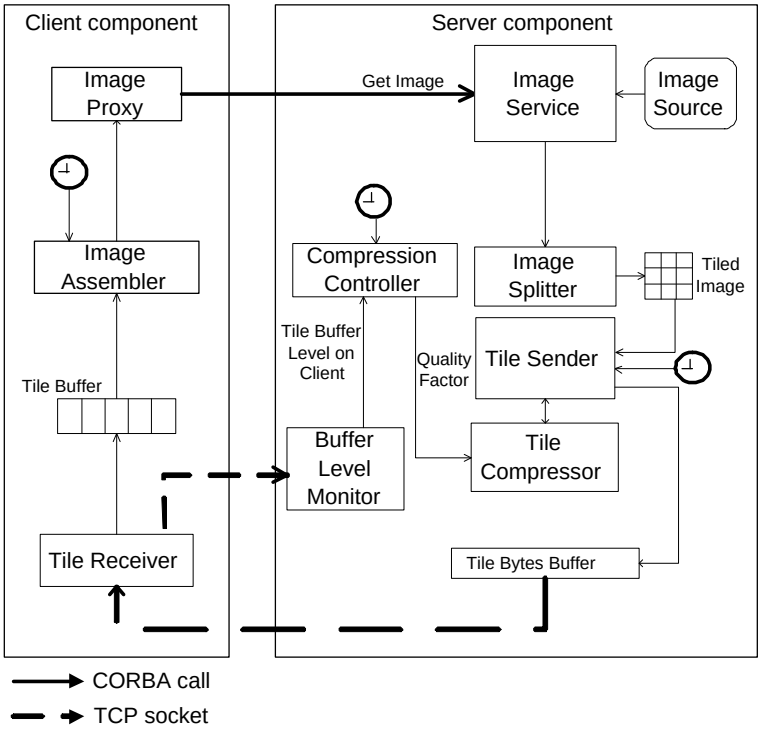



Figure 9.1: Overview of the CAMRIT architecture

mechanisms responsible for requesting and transmitting an image, and then discuss the feedback loop for controlling transmission latency.

### 9.2.2 Image Transmission

The CAMRIT middleware architecture is made up of client and server components, each on a separate endsystem. The *Image Proxy* object in the CAMRIT client component provides the service interface to the application. When it receives a request for an image, this object makes a CORBA call to the *Image Service* object on the server. This CORBA call has the same parameters as the service interface. A *one-way* CORBA call is used to avoid blocking the client thread that executes the call, because transmitting a large image over a bandwidth-constrained network may take a long time.

The Image Service object is implemented as a CORBA servant in the server component, and is advertised to the outside world. When it receives the CORBA call from the client, the Image Service object retrieves the requested image (e.g., from an image repository or a camera), and calls the *Image Splitter* object to split the retrieved image into a specified number of tiles. Each tile is compressed by the *Tile Compressor* object according to the current quality factor, which is periodically updated by the *Controller* object described in Section 9.2.4. The *Tile Sender* object then sends each compressed tile, as a byte stream through a TCP socket, to the client component.

The Tile Sender and Tile Compressor are executed by a periodic task. In each invocation, the Tile Sender fills the TCP buffer by sending image tiles to a TCP socket. The sending socket is set to NON_BLOCKING mode so that the kernel will inform the application layer through an EWOULDBLOCK error from the *send* system call if the TCP buffer is full. Note the sender may push a fraction of a tile to fill the TCP buffer. The pseudo-code for this periodic task is shown below. Tile_Bytes_Buffer is a buffer on the server that is used to hold the bytes of a tile (or fraction of a tile) to be sent.

```
Tile_Sender::handle_timeout() {
  while (1) {
    ret_code = send bytes in Tile_Bytes_Buffer to socket;
    if (ret_code == EWOULDBLOCK)
        exit the current invocation;
    Compress next tile with current quality factor;
    Create a header for the tile ;
    Append the new compressed tile to Tile_Bytes_Buffer;
  }
}
```

The *Tile Receiver* object on the client reads the byte stream from the socket. The boundaries between tiles are indicated in the tile header that precedes each tile. After it receives a whole tile, the Tile Receiver object enqueues the tile into a buffer that holds received but still compressed tiles.

The *Image Assembler* is executed as a periodic task. The first instance of this task is released when the first tile of the image is inserted into the tile buffer. In every invocation, it dequeues and decompresses a tile from the tile buffer if it is not empty.

When all the tiles of an image have been decompressed, it assembles them back into a whole image and notifies the Image Proxy, which then returns a handle (e.g., the memory address) for the decompressed image to the application.

### 9.2.3 Selection of Task Periods

The period of the Tile Sender task is chosen such that the TCP buffer never goes empty while an image is being transmitted to the client. Specifically, if $B$ is the TCP buffer size and $b_{max}$ is the maximum bandwidth of the network, the period of the Tile Sender is set to no higher than $\frac{B}{b_{max}}$. This guarantees that the TCP layer in the kernel has enough bytes of data in the TCP buffer to send before the next invocation of the sending task, and hence the network bandwidth is fully utilized during the transmission of an image.

CAMRIT guarantees image deadlines by achieving the following properties. First, the tile buffer on the client always contains at least one tile during the transmission of an image. This is achieved by a feedback control loop described in the next subsection. Second, every invocation of the Image Assembler task is completed before the end of its period. This property is guaranteed by ensuring that the CPU utilization of the client end-system remains below the schedulable utilization bound of the scheduling algorithm used by RT-CORBA. Finally, the period $p$ of the Image Assembler is selected to meet the end-to-end image deadline, as follows. When the first two properties are satisfied, each invocation of the Image Assembler task decompresses one tile by the the end of its period. Suppose the first tile of an image is inserted into the tile buffer $t_1$ sec after the image request is sent to the server. The first tile is decompressed by $t_1 + p$, and the $i^{th}$ tile is decompressed by $t_1 + ip$. Therefore, the period must satisfy the following condition in order to guarantee the whole image is received and decompressed by the deadline:

$$t_1 + p * num\_tile \leq deadline$$

Hence, the upper bound for the Tile Assembler period is:

$$p \leq \frac{deadline - t_1}{num\_tile} \tag{9.1}$$

### 9.2.4 Feedback Control Loop

As described in the last subsection, CAMRIT must maintain a tile buffer level of at least one tile during the transmission of an image. However, while the Image Assembler dequeues tiles from the tile buffer at a constant rate, the rate at which tiles are inserted into the tile buffer (called the *tile enqueue rate*) depends on the network bandwidth and the size of compressed tiles. To deal with the unpredictable network, we designed a feedback control loop to maintain a specified buffer level (the set point) by periodically adjusting the quality factor of the remaining tiles that are yet to be transmitted. The feedback control loop is composed of a *Buffer Level Monitor*, a *Controller*, and the Tile Compressor described earlier, which serves as an actuator in the control loop.

Each time the Tile Receiver on the client reads a chunk of data from the socket (i.e., completes a `read()` call), it sends the current tile buffer level to the Buffer Level Monitor on the server. Note that the reported buffer level includes the fraction of the tile that is currently being received by the client. For example, if the tile buffer currently contains 3 tiles, and the Tile Receiver has received the first 2KB of another tile of size 5KB, the current buffer level is $3 + 2/5 = 3.4$. The Buffer Level Monitor makes this information available to the Controller. The use of fractional buffer levels as feedback improves control performance because it gives a more precise representation of the buffer level than would integer values.

The Controller periodically re-computes the quality factor of the remaining tiles based on the current tile buffer level. The new quality factor is then used by the Tile Compressor to compress the remaining tiles that are sent in the following sampling period. Clearly, the Controller is critical to the performance of CAMRIT.

## 9.3 Dynamic Model

Modeling the dynamics of the controlled system is crucial for control design. It is also a key challenge in complex distributed middleware systems, whose dynamics are not understood as well as those of many physical control systems. In this section we

establish a dynamic model for a characteristic real-time image transmission system controlled by our feedback control loop.

### 9.3.1 Controlled System Model

As described in the Section 9.2, the controlled variable in our feedback control system is the tile buffer level on the client, and the manipulated variable is the quality factor used by the server to compress tiles. We first introduce some essential notation:

- $T$: the sampling period of the feedback control loop.
- $l(k)$: the tile buffer level at the $k^{th}$ sampling point ($kT$ sec after the system starts). As described in Section 9.2, $l(k)$ may include a fraction of a tile.
- $l_s$: the set point, i.e., the desired tile buffer level.
- $r$: the constant rate (i.e., the frequency) at which tiles are dequeued from the tile buffer by the Image Assembler. It is equal to the inverse of the period of the Image Assembler task, $r = 1/p$.
- $b(k)$: the network bandwidth in the $k^{th}$ sampling period, $[kT, (k+1)T)$. The value of $b(k)$ is unknown *a priori* in an unpredictable network environment, but its range $[b_{min}, b_{max}]$ is usually known.
- $s$: the size of an uncompressed tile. This is known and fixed for a given image and number of tiles.
- $s(q)$: the average size of a tile compressed with a quality factor $q$.
- $q(k)$: the quality factor computed by the controller at the $k^{th}$ sampling point.

In each sampling period, $rT$ tiles are dequeued from the tile buffer. Supposing $n(k)$ tiles are transmitted and inserted to the tile buffer in the $k^{th}$ sampling period, we then have this equation:

$$l(k+1) = l(k) + n(k) - rT \tag{9.2}$$

$n(k)$ depends on the size of compressed tiles and the network bandwidth. The size of a compressed tile is a non-linear function of the quality factor used to compress it. For the purpose of control design, we linearize this function such that

$$s(q) = \frac{sq}{g} \tag{9.3}$$

where $g$ is a gain that can be estimated through linearization in the steady-state operation region of the system. The details of the linearization are presented in Section 9.3.2.

In our control design, we assume $b(k) = b$ where $b$ is the nominal bandwidth. Although we design the controller based on $b$, the controller is tuned such that it remains stable as long as the bandwidth stays within the range $[b_{min}, b_{max}]$.

If we ignore control delay, we get a simple first-order model for the controlled system:

$$l(k+1) = l(k) + \frac{bTg}{sq(k)} - rT \tag{9.4}$$

Unfortunately, this model is inaccurate because control delay plays a major role in the dynamics of our distributed middleware. This control delay can be modeled as the end-to-end latency from the moment when the Tile Receiver sends out the sampled buffer level from the client, to the moment when this new quality factor starts to have an effect on the client tile buffer. We can divide this control delay into the sampling delay from the client to the server and the actuation delay from the server back to the client. Considering the fact that the communication load from the client to the server is signifcantly lower than the opposite direction during the image transmission, we approximate the control delay $t_d(k)$ in our system with the actuation delay, the time interval starting from the moment when the controller on the server outputs the new quality factor $q(k)$.

The control delay is due to residual data in the TCP buffer and the Tile Byte Buffer on the server. When the controller outputs a new quality factor, these buffers still contain tiles compressed with the *old* quality factor, $q(k\text{-}1)$. Hence the system will continue to transmit and enqueue those *old* tiles to the tile buffer on the client until

all the data in the TCP buffer and the Tile Byte Buffer have been transmitted to the server.

Let $s_t(k)$ and $s_b(k)$ denote the amount of data in the TCP buffer and the Tile Byte Buffer, respectively. The control delay is then

$$t_d(k) = \frac{s_t(k) + s_b(k)}{b} \tag{9.5}$$

To calculate the control delay, we need to estimate $s_t(k)$ and $s_a(k)$. First, we consider $s_t(k)$. Suppose the TCP buffer size is $B$, and the period of the Tile Sender task is $p_s$. The TCP buffer is full (i.e., contains $B$ bits of data) at the end of each invocation of the Tile Sender task. During each period of the Tile Sender, $bp_s$ bits of data are transmitted from the TCP buffer. Therefore, the lower bound for the amount of data that the TCP buffer may hold is $B - bp_s$ bits. Since $s_t(k)$ depends on the specific time when the controller outputs $q(k)$, we approximate $s_t(k)$ with the average of its upper bound and lower bound for our control design:

$$s_t = B - \frac{bp_s}{2} \tag{9.6}$$

As Section 9.3.2 describes, the Tile Byte Buffer holds the fraction of a compressed tile that cannot fit into the TCP buffer. On average, this buffer contains half of a tile compressed with quality factor $q(k\text{-}1)$ at the beginning of the $k^{th}$ sampling period. We approximate $s_b(k)$ with its average value, based on (9.3):

$$s_b(k) = \frac{sq(k-1)}{2g} \tag{9.7}$$

As Figure 9.2 illustrates, if we choose a sampling period $T > t_d(k)$, the tiles placed into the tile buffer in the first $t_d(k)$ secconds of the $k^{th}$ sampling period are compressed with quality factor $q(k\text{-}1)$, and the tiles placed there in the remaining part of the sampling period are compressed with quality factor $q(k)$. Therefore, a more accurate model that considers the control delay is

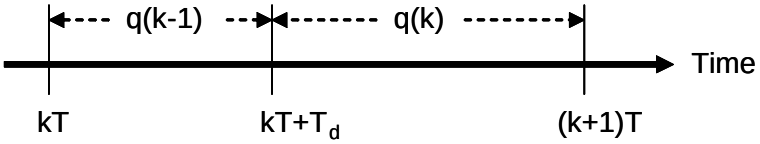



Figure 9.2: Quality factors of tiles received in the $k^{th}$ sampling period

$$l(k+1) = l(k) + \frac{bt_d(k)g}{sq(k-1)} + \frac{b(T-t_d(k))g}{sq(k)} - rT \tag{9.8}$$

Note that the second to last term in (9.8) is non-linear because it includes both $q(k)$ and $t_d(k)$, which is a function of $q(k\text{-}1)$ (see (9.5) and (9.7)). Since the quality factor does not change significantly in a steady state, we can linearize this model by replacing the $q(k\text{-}1)$ in this term with $q(k)$. Finally, let $u(k) = 1/q(k)$ be the control input. We then have an approximate linear model of the controlled system:

$$l(k+1) = l(k) + Au(k) + Cu(k-1) + D \tag{9.9}$$

where $A = \frac{(bT-s_t)g}{s}$, $C = \frac{s_t g}{s}$ and $D = -rT$.

When control delay is zero, this model is the same as the first-order model in (4). However, when control delay is comparable to the sampling period, the coefficient of the second order term $q(k\text{-}1)$ becomes significant, and the second-order model is needed to capture the system dynamics.

### 9.3.2 Tile Size and Quality Factor

We now describe how to estimate the gain $g$. We first compare the size of the compressed sample image $s(q)$ with each quality factor $q$, and plot the *inverse of the compression ratio* $a(q) = s/s(q)$ as a function of the inverse of the quality factor $u = 1/q$, which is the control input. For an example aerial image shown in Figure 9.3 (called Image 0[1] in this chapter) its resulting profile of the relationship between those parameters is a non-linear curve. We linearize $a(u)$ in the operational region of the system in steady state, in the following three steps.

[1]All images used in this chapter are available at
$http://deuce.doc.wustl.edu/FCS\_nORB/CAMRIT$.

Figure 9.3: An example aerial image

1. Given the deadline $d$ for transmission of an image, the rate $r$ of the Image Assembler is calculated using (9.1). In steady state, tiles are transmitted from the server to the client at the same rate as $r$, to maintain a constant tile buffer level.

2. We then use the following equation to calculate the range of $a(u)$, $[a_{min}, a_{max}]$, that can satisfy the tile transmission rate $r$ in steady state based on the range of possible network bandwidth $[b_{min}, b_{max}]$.

$$\frac{ba(u)}{s} = r \tag{9.10}$$

3. Finally, we perform linear regression on the segment of function $a(u)$ where $a_{min} \leq a(u) \leq a_{max}$. The slope of the linear regression is the estimated $g$.

When an image request is submitted, CAMRIT uses the estimation process above to derive $g$, based on the specified deadline and the function $a(u)$ from the profiling results for a representative image. While function $a(u)$ may differ for different images, the difference is small for images in a similar application domain (e.g., landscape images taken from airplanes). Furthermore, the feedback control loop can be designed to tolerate a range of variations in $g$.

As an example, we now show how to estimate $g$ based on hypothetical but plausible system settings, and using the measured profile for Image 0. The key parameters for this example are as follows:

- Image: 640×640 pixels; divided into 64 tiles; each uncompressed tile size $s$ = 18.75 KB.
- Deadline: $d$ = 200 sec.
- Bandwidth: [4 Kbps, 8 Kbps]. The top of this bandwidth range approximates the maximum data rate of a single link at the lowest Link-16 network capacity of 28.8 Kbps [100], with time slots divided among links to 3 aircraft collaborating with a common JTIDS terminal on the Command-and-Control aircraft (C2); we assume a minimum network bandwidth of half the maximum; we use the midpoint of the resulting range, $b$ = 6 Kbps, for our control design.

The rate of the Image Assembler (also the steady-state tile transmission rate) is computed using (9.1). CAMRIT uses 95% of the actual deadline to give some leeway to the transmission, and $t_1$ is estimated based on the nominal bandwidth and the tile size with the initial quality factor (68 in this example). The resultant $r$ = 0.34 tile/sec. According to (9.10), in order to allow the bandwidth variation from 4 Kbps to 8 Kbps, the range for the inverse of compression ratio needs to be [6.38, 12.75]. Linearization is then performed in this range for $a(q)$ as shown in Figure 9.4. The slope of the linear regression is $g$ = 341.34. The linear regression fits well (with an $R^2$ = 94.87%) with the original function in this operation region.

## 9.4 Control Design and Analysis

We now apply linear control theory to design the controller based on the controlled system model described in Section 9.3. The z-transform of the controlled system model (9.9) is:

$$L(z) = z^{-1}L(z) + Az^{-1}U(z) + Cz^{-2}U(z) + \frac{Dz}{z-1} \tag{9.11}$$

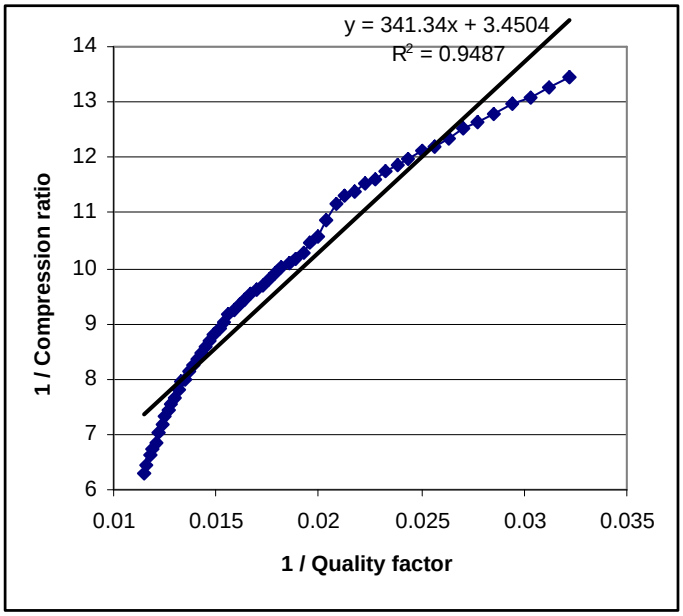



Figure 9.4: Linearization of $a(u)$

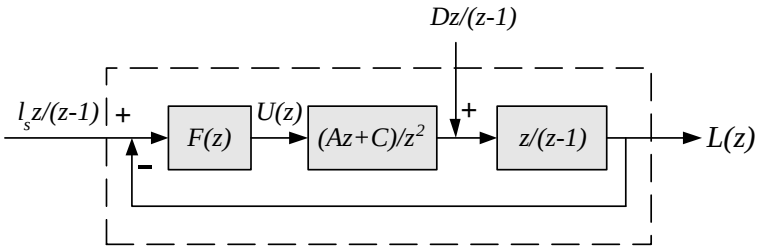



Figure 9.5: Block diagram of closed-loop system

A block diagram of the closed-loop system is shown in Figure 9.5. The system has two inputs: the set point of the tile buffer level and a disturbance input $\frac{Dz}{z-1}$ that represents the dequeuing of tiles from the tile buffer by the Image Assembler.

Letting $F(z)$ be the transfer function of the controller, we can derive the closed-loop transfer function in response to the reference input and disturbance, respectively:

$$\begin{aligned} H_s(z) &= \frac{(Az+C)F(z)}{(z-1)z+(Az+C)F(z)} \\ H_d(z) &= \frac{z^2}{(z-1)z+(Az+C)F(z)} \end{aligned} \tag{9.12}$$

Therefore, the close-loop response to both inputs is

$$L(z) = H_s(z)\frac{z}{z-1}l_s + H_d(z)\frac{z}{z-1}D \tag{9.13}$$

To achieve stability and zero steady state error, we design a *Proportional-Integral (PI)* controller for our system:

$$F(z) = \frac{K_1(z-K_2)}{z-1} \tag{9.14}$$

The time-domain form of (9.14) is:

$$u(k) = u(k-1) + K_1e(k) - K_1K_2e(k-1) \tag{9.15}$$

where $K_1$ and $K_2$ are control parameters that can be analytically tuned to guarantee system stability and zero steady state error using standard control design methods.

We first apply the control design to our example application integrated with the CAMRIT framework. The sampling period is $T$=10 sec. The TCP buffer size is $B$ = 4 KB. The period of the Tile Sender task is set to 2.67 sec to fully utilize network bandwidth. The other parameters (including $g$) are the same as for the example given in Section 9.3.2. From (9.5), the control delay in the $k^{th}$ sampling period is $T_d = 4+q(k-1)/27.31$ sec. For example, the control delay is 5.8 sec when $q(k$-1)=50. Compared to a sampling period of 10 sec, the control delay clearly plays a significant role in the system dynamics. From (9.9), the parameters of the controlled system model are A=81.922; C=54.614; D=-3.420.

Using the Root-Locus method, we select our control parameters as $K_1$=0.0068 and $K_2$=0.9. The corresponding closed-loop poles are $0.278 \pm 0.547i$ and 0.887. Since all the poles are in the unit circle, the system is stable. From the final value theorem [24], we have proved that the closed-loop system achieves zero steady state error. That is, the tile buffer level will achieve the set point in steady state: $\lim_{k\to\infty} l(k) = l_s$. If the set point is set to $l_s \geq 1$, the tile buffer will remain non-empty in steady state, and hence the image transmission deadline will be met. Furthermore, by substituting different bandwidths into the system model, we can prove that the system can maintain stability and zero steady-state error with the same control parameters as long as the

network bandwidth remains within the range [4Kbps, 8Kbps]. A detailed analysis is not given here due to space limitations: interested readers are referred to a standard control textbook [24].

In summary, pseudo code for the control algorithm implemented in CAMRIT is as follows:

```
Controller(l_s, K_1, K_2) {
    ℓ = current tile buffer level;
    e = ℓ_s − ℓ;
    u = u + K_1*e − K_1*K_2*e_prev;
    e_prev = e;
    q = 1/u;
    /* enforce constraints on acceptable quality factor */
    /* default range is [1,100] */
    if (q < q_min) q = q_min;
    if (q > q_max) q = q_max;
    UpdateQF(q);
    /* updated q will be used by the Tile Compressor */
}
```

## 9.5 Experimental Evaluation

### 9.5.1 WSOA Scenario

The Weapons System Open Architecture (WSOA) [20] program had a primary objective to provide internet-like connectivity, over Link-16, between legacy embedded mission systems in a fighter aircraft and off-board Command and Control (C2) systems. This capability was designed to support time-sensitive mission re-planning and redirection of attack nodes, as necessary based on situational events, even if a different mission was already underway.

The following high-level sequence of interactions between the C2 and fighter aircraft constitutes a representative WSOA scenario: 1) The C2 node receives information about a higher priority time critical target and requests a planning session with

attack nodes by sending an alert; 2) Upon receiving an alert, a fighter aircraft begins downloading a Virtual Target Folder (VTF). The VTF contains several thumbnail-sized images, each representing a virtual target; 3) Once the fighter receives a folder, the pilot can select a thumbnail image in the folder via a graphical display; 4) A request is then made to the C2 for a larger version of the selected image. The experiments presented in this chapter emulate step 4, which is the most time critical part of the application.

### 9.5.2 Experimental Platform

Our experimental configuration consists of two machines each running RedHat Linux 9.0 with the 2.4.20 kernel. The C2 aircraft and the fighter were simulated using a 2.53GHz Pentium IV and a 400MHz Pentium II, respectively. The following software was used to perform the experiments:

- **A**CE 5.3.5 + TAO 1.3.5 : TAO is a widely used open-source real-time CORBA standard object request broker [18]. TAO also provides a Real Time Event Channel [68] that is integrated with the Kokyu dispatching and scheduling framework [28]. This integrated middleware framework allow us to (re)schedule rates of invocation of application components, while maintaining deadline-feasible scheduling of critical operations.
- **I**mageMagick++ 5.5.7 : We used this library to compress and decompress images.
- **S**haper 1.3 for Linux : Shaper is a linux script for traffic shaping. It allows us to specify the maximum bandwidth for network connection between two hosts.

We used Shaper to control the bandwidth between the two machines, i.e., to simulate the performance of a Link-16 or other bandwidth-constrainted network over an underlying Ethernet connection. We set the range of bandwidth allowed by the traffic shapers to approximate the effective bandwidth of a plausible Link-16 configuration, e.g., with a maximum network capacity of 28.8 Kbps [100], divided between the client

and server. Taking into account the slotted nature of Link-16 communication channels and other Link-16 parameters, and the characteristics of the traffic shaper we used, we chose a maximum bandwidth of 8 Kbps for our experiments.

### 9.5.3 Experimental Parameters

Our experiments used the same parameters as the examples in sections 9.3.2 and 9.4. To test CAMRIT's ability to handle different images, our experiments used two other aerial images than Image 0, whose profile was used to tune the control parameters. These two images are called Image 1 and Image 2 respectively. The number of tiles for each image is set to 64 for our experiments, to achieve a reasonable balance between control granularity and overhead.

The set point for the tile buffer level was $l_s = 5$ in our experiments. Note that there is a tradeoff in the choice of the set point. If the set point is too high, the quality factor for tiles transmitted in the first several sampling periods will be unnecessarily low because system has to fill an initially empty buffer with more tiles (with lower quality factors) before it reaches a steady state. On the other hand, if the set point is too low, a fluctuation in the network bandwidth may cause the buffer level drop to zero.

### 9.5.4 Experimental Results

CAMRIT uses (9.10) to calculate $q(0)$ based on its deadline, the nominal bandwidth (6 Kbps), and the profiled image quality function for Image 0. The resulting initial quality factor is $q(0) = 68$ in all of the following experiments. While $q(0)$ provides a reasonable initial value for the control input, that initial value is usually not correct for meeting the deadline because the actual bandwidth may differ from the nominal one.

The tile buffer level and quality factors during a typical transmission of Image 1 over a 6 Kbps network are shown in Figures 9.6 and 9.7, respectively. The buffer level is recorded by the Image Assembler before everytime it attempts to dequeue a tile.

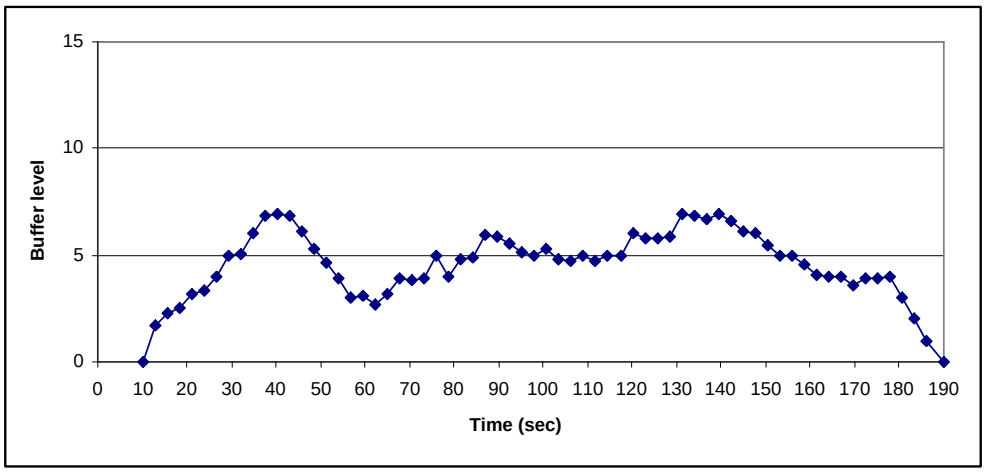



Figure 9.6: Tile buffer levels during typical transmission of Image 1

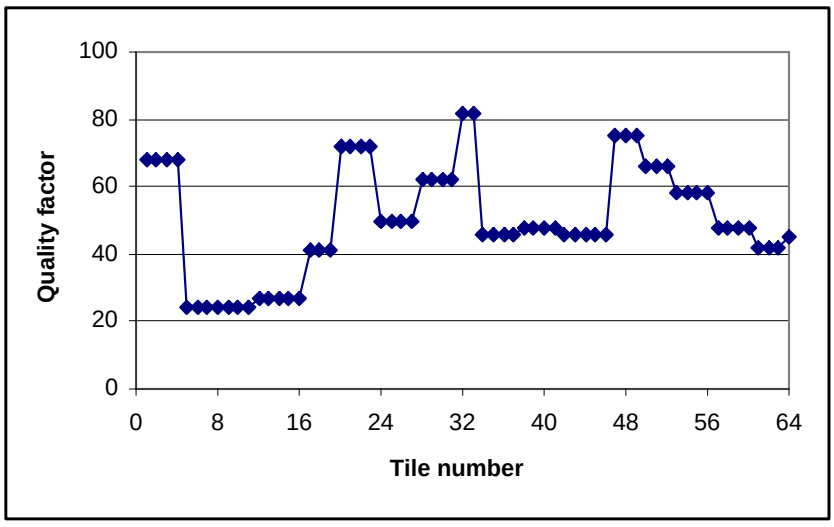



Figure 9.7: Quality factors during typical transmission of Image 1

Time 0 in Figure 9.6 represents the time instant when the image request is sent to the server. The tile buffer is initially empty until the first tile is inserted at around 11 sec. This 11 sec delay includes the time it takes CAMRIT to send the image request to the server, divide the image into tiles on the server, and transmitting the first tile. Since the buffer level is low initially, CAMRIT reduces the quality factor from 68 to about 20 so that the buffer level rises to 5 tiles (the set point) in about 20 sec. The buffer level remains close to 5 tiles until the last image is transmitted to the client near the end of the run. The transmission of the whole image is completed at time 190 sec. This is consistent with our expectation because 190 sec (95% of the deadline) is used to compute the rate of the Image Assembler. Both tile buffer level and quality factor have some oscillation due to system noise. For example, the sizes of different tiles may be different (corresponding to different $g$ values in our model) even if they are compressed using a same quality factor. However, despite the noise the tile buffer is always above 2.5 throughout the transmission. This is important because CAMRIT can guarantee an image transmission deadline is met as long as the tile buffer always contains at least one tile.

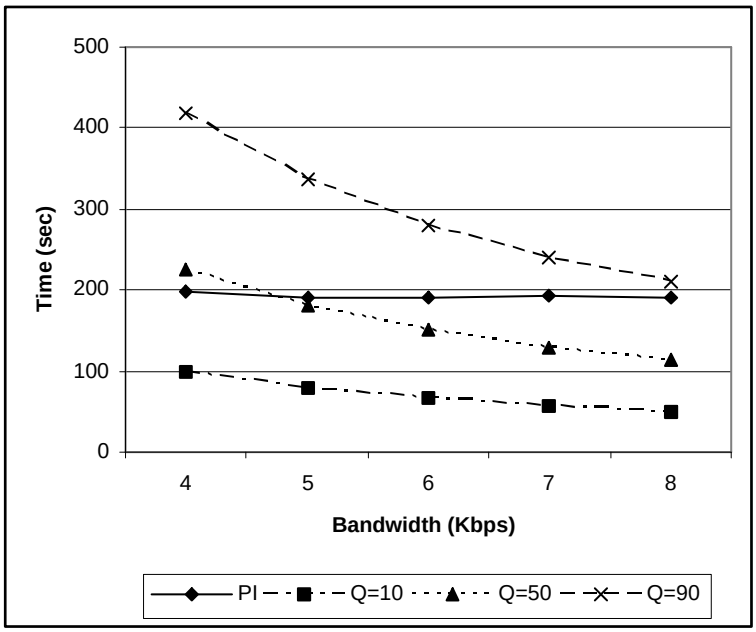



Figure 9.8: Transmission delay under different network bandwidth

The primary goal of CAMRIT is to meet image transmission deadlines. Figure 9.8 shows the transmission delay of Image 1 under different bandwidths. The transmission delay of CAMRIT (with the feedback loop) is measured through experiments. Each data point of CAMRIT in Figure 9.8 is the mean of 10 repeated runs. The standard deviation of each data point is within 2.62 sec. The transmission delay results of Image2 are not shown because they are almost identical to those of Image 1. For comparison purposes, we also plot the estimated transmission delays for Image 1 when a fixed quality factor (10, 50, or 90) is used in each run. The transmission delay for an image with a fixed quality factor is estimated by dividing its total (compressed) tile size by the actual network bandwidth[2].

We can see that the transmission delays for images with fixed quality factors vary significantly as the network bandwidth changes. This result confirms the difficulty in selecting a proper quality factor *a priori* when the network bandwidth is unpredictable. A chosen quality factor may be unnecessarily low when transmission completes much earlier than the deadline, or too high causing a deadline miss.

In contrast, the transmission delay under CAMRIT remains close to 190 sec (95% of the original deadline) as the network bandwidth varies from 4 Kbps to 8 Kbps, and every run meets the deadline of 200 sec. The robust real-time performance is attributed to the feedback control loop that effectively maintains the desired buffer level despite the variation in network bandwidth.

The secondary goal of CAMRIT is to improve the image quality. CAMRIT accomplishes this goal by 1) fully utilizing the network bandwidth and 2) completing the transmission of an image close to the deadline (as shown in Figure 9.8). The combination of both properties means that CAMRIT sends close-to-maximum amounts of data for a requested image, which generally corresponds to a higher image quality.

Figure 9.9 shows the average quality factors of both images when they are transmitted by CAMRIT under different network bandwidths. Each data point is the mean of 10 repeated runs. The standard deviations are also shown. With CAMRIT the average quality factor improves as more network bandwidth becomes available. This result

[2]This estimation is slightly lower than the actual delay because it ignores the overhead of protocol headers.

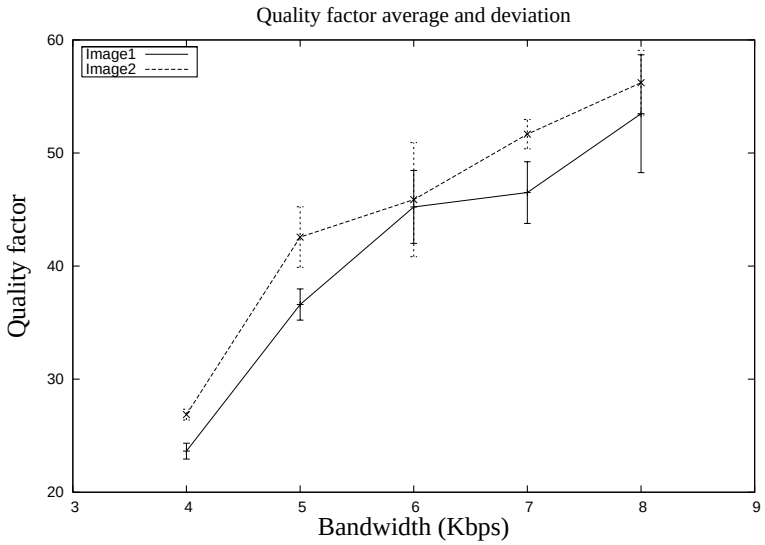



Figure 9.9: Average quality factor under different network bandwidth

determines that CAMRIT can automatically adapt to network bandwidth variations by adjusting the quality factor.

## 9.6 Summary

In this chapter, we have presented the design, modeling, and analysis of CAMRIT based on a control theoretic approach. A key contribution of this work is an analytic model that captures the dynamics of a moderately complex distributed middleware architecture. CAMRIT has been successfully implemented as a CORBA-based middleware service atop the TAO real-time ORB. Our experiments on a representative testbed demonstrate that CAMRIT can provide robust feedback control of image transmission delays across a range of available network bandwidth, by automatically adjusting image tile quality factors.

# Chapter 10

# Conclusions and Future Work

## 10.1 Conclusions

This dissertation has presented an adaptive QoS control framework, which is specifically designed for distributed real-time embedded systems running in unpredictable environments where their workloads are unknown or vary significantly at run-time. The framework includes a set of control design methodologies to provide robust QoS assurance for systems at different scales. In this dissertation, we have applied the framework to the end-to-end CPU utilization control problem which is formulated as a constrained multi-input-multi-output control model. Table 10.1 summaries the algorithms and systems we developed in the framework for utilization control and related problems.

Table 10.1: Adaptive QoS control framework

<table>
<tr><th>System scale</th><th colspan="2">Algorithms or Systems</th></tr>
<tr><td rowspan="2">Small distributed systems<br>(Centralized MIMO control)</td><td>EUCON: end-to-end utilization control<br>FC-ORB: EUCON + robust end-to-end middleware</td><td rowspan="2">Controllability and feasibility</td></tr>
<tr><td>Large distributed systems<br>(Decentralized MIMO control)</td><td>DEUCON: decentralized end-to-end utilization control</td></tr>
<tr><td>Single processor systems<br>and networks<br>(SISO control)</td><td colspan="2">FCS/nORB: utilization and deadline control<br>CAMRIT: real-time transmission delay control<br>Power control for computing servers [95]</td></tr>
</table>

**Small-scale distributed real-time systems**

Many DRE systems (e.g., avionics systems, shipboard computing, and process control systems) depend on server clusters in which several processors connect to each other through a high speed communication interface (e.g., a VME bus backplane). For this class of DRE systems, a centralized QoS control architecture is usually sufficient and more preferable for considerations in security and efficiency.

EUCON (End-to-end Utilization CONtrol) [59] is the first control-theoretic utilization control algorithm designed for this class of DRE systems with end-to-end tasks. EUCON can maintain desired CPU utilizations on multiple processors despite uncertainties in task execution times and coupling among processors. It employs a centralized MIMO model predictive controller to manage and coordinate the adaptation of multiple processors, subject to the constraints on task rates.

FC-ORB is a real-time Object Request Broker (ORB) middleware that employs the EUCON control algorithm to handle fluctuations in application workload and system resources. FC-ORB demonstrates that the integration of adaptive QoS control, end-to-end scheduling and fault-tolerance mechanisms in DRE middleware is a promising approach for enhancing the robustness of DRE applications in unpredictable environments.

**Large-scale distributed real-time systems**

While EUCON and FC-ORB are suitable for small-scale DRE systems, a centralized control scheme has several limitations. Since its communication and computation overhead depends on the size of an *entire* DRE system, it cannot handle large-scale systems (e.g. wide-area power grid management and ubiquitous smart spaces). Furthermore, the processor executing the controller is a single point of failure since the entire system will lose the capability of QoS adaptation if it fails.

DEUCON is a decentralized end-to-end utilization control algorithm that can provide utilization control for large-scale DRE systems. In contrast to centralized control schemes, DEUCON features a novel decentralized control structure that requires

only localized coordination among neighbor processors. DEUCON can effectively distribute the computation and communication cost to different processors and tolerate considerable communication delay between local controllers. Therefore, DEUCON can provide scalable and robust utilization control for large-scale distributed real-time systems executing in unpredictable environments.

Guaranteeing system controllability and feasibility is a fundamental problem of end-to-end utilization control. Neither centralized nor decentralized control algorithms can control a system if the system itself is uncontrollable or infeasible for control. Controllability and feasibility depend crucially on the end-to-end task configuration of a DRE system. Novel task allocation algorithms are developed to ensure that the system is controllable and robustly feasible. Furthermore, we have developed run-time algorithms that maintain controllability and feasibility by reallocating subtasks dynamically in response to task termination and arrival.

**Single processor real-time systems**

While the focus of this dissertation is on distributed real-time systems, our framework also includes algorithms and system implementations for single processor real-time systems. As a starting point for adaptive middleware with end-to-end utilization control, we have developed a single processor utilization control middleware called FCS/nORB. FCS/nORB integrates utilization control with a small-footprint real-time ORB such that it is truly portable in terms of real-time performance and functionality. Our experiments demonstrate that FCS/nORB can provide deadline miss ratio and utilization guarantees in face of changes in the platform and task execution times, while introducing only a small amount of overhead.

As a case study of applying our framework to networking problems, we have developed a control-based adaptive middleware called CAMRIT for real-time image transmission. CAMRIT features a distributed feedback control loop that meets image transmission deadlines by dynamically adjusting the quality of image tiles in response to varying network bandwidth. Experimental results demonstrate that CAMRIT can provide robust real-time delay guarantees for a representative application scenario.

We have also applied the control framework to power control for IBM high-performance blade servers [95] during a summer research internship at IBM Austin Research Lab. In recent years, even high-performance servers are becoming power-constrained due to increasing density and computing capabilities. The situation becomes even worse in the event of a partial power-supply failure. Our design focuses on controlling system-level power consumption by adapting the performance states of the microprocessors. The resultant theoretical controller outperforms both traditional open-loop solutions and a reasonably designed heuristic controller, in terms of better system performance and more accurate power control. This control design resulted from our control framework is now having a big impact at IBM and will be adopted in real blade server products.

This dissertation work has had a broad technology impact because it not only produced 11 research papers (5 conference papers, 2 journal papers and 4 submissions), it also generated three real-time middleware systems and two event-driven simulators, which are being used by other research groups at a variety of universities and research labs. All the software is open-source and is publicly released at:
`http://deuce.doc.wustl.edu/FCS_nORB/`.

## 10.2 Future Work

The successful application of our adaptive QoS control framework to end-to-end utilization control and related problems suggests several interesting future research directions.

First, the QoS control framework presented in this dissertation can be extended to a wide range of real-time and QoS-critical systems. For example, our work on power control for a single computer server [95] has gained us confidence to develop more advanced control algorithms to control both power and thermal for high-performance computing clusters. The centralized EUCON control algorithm can be extended to control all blade servers on a chassis which compose a small-scale distributed system. The decentralized DEUCON control algorithm could be an effective approach to power control in a whole commercial data center which hosts hundreds of blade

servers. Another promising application of the control framework is QoS control in distributed storage systems. Recently, a novel storage architecture is distributed storage systems composed from numerous inexpensive COTS storage disks (e.g. the federated array of bricks (FAB) project at HP Labs [75]). Decentralized control and theoretical guarantees on critical QoS metrics like response time and throughput could be crucial to the success of this new architecture. Other possible applications include peer-to-peer multimedia streaming, content delivery network, distributed data mining and web-services based applications.

Second, one of our ultimate goals is to develop adaptive, resilient and secure middleware for distributed real-time systems. In this dissertation, the integration of the FC-ORB control middleware with traditional fault-tolerance mechanisms is proved to be a step towards self-managing, self-healing and self-tuning distributed computing platforms. Hence, it is interesting to integrate our control algorithms and middleware with more advanced fault-tolerance and security mechanisms, so analytic guarantees can be provided for real-time and QoS-critical applications to tolerate and survive malicious security attacks such as sophisticated DDOS attacks. For example, adaptive and resilient middleware can be used in the supervisory control and data acquisition (SCADA) systems to meet some important challenges, such as maintaining critical real-time performance in power grid management even under cascading failures.

Finally, new control algorithms may have to be designed for DRE systems with special configurations. For example, processors in some networked embedded systems such as large-scale wireless sensor networks have extremely limited computing capacity and storage. As a result, the EUCON and DEUCON control algorithms presented in this dissertation may not be a good fit with those systems, because the two algorithms are developed based on model predictive control theory which requires the processors to solve constrained least squares problems. It would be challenging to simplify the existing control algorithms to provide QoS control for those systems with guaranteed stability. In addition, some distributed real-time systems such as avionics only support *discrete* control variables, so hybrid (continuous/discrete) control algorithms need to be developed to handle end-to-end tasks. Furthermore, adaptive and robust control theory may also be used to improve control performance when system model varies dynamically at runtime.

# Appendix A

# Transformation to Least-Squares Problem

A standard constrained least-squares problem is in the form of

$$\min_{s(k)}(\sum_{i=1}^{P}\|\Theta s(k)-E(k)\|^2_{Q(i)}+\sum_{i=0}^{M-1}s(k)^2_{R(i)}) \tag{A.1}$$

subject to constraints $\Omega s(k) \leq \omega$.

where $s(k)$ denotes the vector of change to the control input in the control horizon. In EUCON, $s(k) = \begin{bmatrix} \Delta r(k|k) - \Delta r(k-1) \\ \vdots \\ \Delta r(k+M-1|k) - \Delta r(k+M-2|k) \end{bmatrix}$.

To transform our control problem to a least-squares problem, we re-write our cost function in (4.5) and constraints (3.1) in the form (A.1). Since the control penalty terms in (4.5) is consistent with (A.1), we only need to transform the tracking error term in (4.5) and the constraints 3.1) to formulations in terms of $s(k)$. First we work on the tracking error term in (4.5). From the plant model (4.4) and (4.7), the predicted utilization for given prediction horizon can be written as:

$$\begin{bmatrix} u(k+1|k) \\ \vdots \\ u(k+M|k) \\ u(k+M+1|k) \\ \vdots \\ u(k+P|k) \end{bmatrix} = \begin{bmatrix} u(k) \\ \vdots \\ u(k) \\ u(k) \\ \vdots \\ u(k) \end{bmatrix} + \begin{bmatrix} F \\ \vdots \\ \sum_{i=0}^{M-1} F \\ \sum_{i=0}^{M} F \\ \vdots \\ \sum_{i=0}^{P-1} F \end{bmatrix} \Delta r(k-1) + \begin{bmatrix} F & 0 & \cdots & 0 \\ F+F & F & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^{M} F & \sum_{i=0}^{M-1} F & \cdots & 2F \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^{P-1} F & \sum_{i=0}^{P-2} F & \cdots & \sum_{i=0}^{P-M} F \end{bmatrix} \tag{A.2}$$

We can rewrite (A.2) as:

$$\mathbf{u}'(k) = \mathbf{u}(k) + \mathbf{\Gamma}\mathbf{\Delta r}(k-1) + \mathbf{\Theta s}(k) \tag{A.3}$$

where

$$\mathbf{u}'(k) = \begin{bmatrix} u(k+1|k) \\ \vdots \\ u(k+M|k) \\ u(k+M+1|k) \\ \vdots \\ u(k+P|k) \end{bmatrix}, \mathbf{\Gamma} = \begin{bmatrix} F \\ \vdots \\ \sum_{i=0}^{M-1} F \\ \sum_{i=0}^{M} F \\ \vdots \\ \sum_{i=0}^{P-1} F \end{bmatrix}, \mathbf{\Theta} = \begin{bmatrix} F & 0 & \cdots & 0 \\ F+F & F & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^{M} F & \sum_{i=0}^{M-1} F & \cdots & 2F \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^{P-1} F & \sum_{i=0}^{P-2} F & \cdots & \sum_{i=0}^{P-M} F \end{bmatrix} \tag{A.4}$$

In addition, we define

$$\mathbf{E}(k) = \mathbf{ref}'(k) - \mathbf{u}(k) - \mathbf{\Gamma}\mathbf{\Delta r}(k-1) \tag{A.5}$$

where $\mathbf{ref}'(k)$ represents the reference trajectory for specified prediction horizon: $\mathbf{ref}'(k) = \begin{bmatrix} ref(k+1|k) \\ \vdots \\ ref(k+P|k) \end{bmatrix}$.

Given $\mathbf{\Theta}$ and $\mathbf{E}(k)$ in (A.3) and (A.4), our cost function (4.5) is equivalent to the one in the least-squares problem (A.1). We now transform the constraints (3.2-3.3) to the linear inequality constraint form as $\Omega s(k) \leq \omega$. Firstly we transform the rate constraint (3.3) in control horizon $M$ as:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ -1 & 0 & \cdots & 0 \\ 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 \end{bmatrix} \begin{bmatrix} r(k) \\ r(k+1) \\ \vdots \\ r(k+M-1) \end{bmatrix} \leq \begin{bmatrix} R_{max} \\ R_{max} \\ \vdots \\ R_{max} \\ -R_{min} \\ -R_{min} \\ \vdots \\ -R_{min} \end{bmatrix}$$

From $r(k) = r(k-1) + \Delta r(k)$, the above inequality is equivalent to

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \\ -1 & 0 & \cdots & 0 \\ -1 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & -1 \end{bmatrix} \begin{bmatrix} \Delta r(k) \\ \Delta r(k+1) \\ \vdots \\ \Delta r(k+M-1) \end{bmatrix} \leq - \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix} r(k-1) + \begin{bmatrix} R_{max} \\ R_{max} \\ \vdots \\ R_{max} \\ -R_{min} \\ -R_{min} \\ \vdots \\ -R_{min} \end{bmatrix}$$

From $\Delta r(k) = \Delta r(k-1) + (\Delta r(k) - \Delta r(k-1))$, we can transform the rate constraints to the following linear inequality constraints:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ M & M-1 & \cdots & 1 \\ -1 & 0 & \cdots & 0 \\ -2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -M & -(M-1) & \cdots & -1 \end{bmatrix} s(k) \leq - \begin{bmatrix} 1 \\ 2 \\ \vdots \\ M \\ -1 \\ -2 \\ \vdots \\ -M \end{bmatrix} \Delta r(k-1) - \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix} r(k-1) + \begin{bmatrix} R_{max} \\ R_{max} \\ \vdots \\ R_{max} \\ -R_{min} \\ -R_{min} \\ \vdots \\ -R_{min} \end{bmatrix} \tag{A.6}$$

Now we consider the utilization bound constraints (3.2). From (3.2) and (A.3) the utilization bound constraints are equivalent to the following linear inequality

$$\mathbf{\Theta s}(k) \leq -\mathbf{u}(k) - \mathbf{\Gamma \Delta r}(k-1) + \mathbf{B} \tag{A.7}$$

We have transformed our MPC formulation to a constrained least-square formulation described by (A.1, A.3-A.6). Since the constraints (A.6-A.7) depend on $u(k)$, $\Delta r(k-1)$, and $r(k-1)$, both of them are known at time $k$. We can use any standard least-squares solver to solve this control problem now.

# Appendix B

# Detailed Stability Analysis in EUCON

**Example:** We now apply the stability analysis approach to the example system described in the end of Section V. The system has 3 tasks and 2 processors. We set the prediction horizon $P = 2$ and the control horizon $M = 1$. According to the MPC theory, the system is also stable with any longer prediction horizon and control horizon if it is stable with shorter horizons. The time constant of the reference trajectory is $T_{ref}/T_s = 4$. The weights assigned to all terms are 1. The cost function can be transformed to the following formula in scalar form:

$$V(k) = \sum_{j=1}^{2}\sum_{i=1}^{2}(u_j(k+i|k) - ref_j(k+i|k))^2 + \sum_{j=1}^{3}(\Delta r_j(k) - \Delta r_j(k-1))^2 \quad \text{(B.1)}$$

Substituting the model parameters to (4.4) and (4.7), we have

$$\begin{bmatrix} u_1(k+1) \\ u_2(k+1) \\ u_1(k+2) \\ u_2(k+2) \end{bmatrix} = \begin{bmatrix} u_1(k) \\ u_2(k) \\ u_1(k+1) \\ u_2(k+1) \end{bmatrix} + \begin{bmatrix} c_{11} & c_{21} & 0 \\ 0 & c_{22} & c_{31} \\ c_{11} & c_{21} & 0 \\ 0 & c_{22} & c_{31} \end{bmatrix} \begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \\ \Delta r_3(k) \end{bmatrix} \quad \text{(B.2)}$$

For simplicity, we use $u_i$ and $\Delta r_i$ to represent $u_i(k)$ and $\Delta r_i(k)$, respectively, in the rest of this section. Substitute (B.2) and the reference trajectory (4.6) in (B.1), the

cost function becomes

$$\begin{aligned} V(k) &= [u_1 + c_{11}\Delta r_1 + c_{21}\Delta r_2 - (B_1 - \lambda(B_1 - u_1))]^2 \\ &+ [u_2 + c_{22}\Delta r_2 + c_{31}\Delta r_3 - (B_2 - \lambda(B_2 - u_2))]^2 \\ &+ [u_1 + 2c_{11}\Delta r_1 + 2c_{21}\Delta r_2 - (B_1 - \lambda^2(B_1 - u_1))]^2 \\ &+ [u_2 + 2c_{22}\Delta r_2 + 2c_{31}\Delta r_3 - (B_2 - \lambda^2(B_2 - u_2))]^2 \\ &+ [\Delta r_1 - \Delta r_1(k-1)]^2 + [\Delta r_2 - \Delta r_2(k-1)]^2 \\ &+ [\Delta r_3 - \Delta r_3(k-1)]^2 \end{aligned} \tag{B.3}$$

where $\lambda = e^{-T_s/T_{ref}}$. We then perform partial differentiation on $V(k)$ with respect to $\Delta r_1, \Delta r_2$ and $\Delta r_3$, respectively. The derivatives are set to zero to compute the control input vector $\mathbf{\Delta r(k)}$ that minimize the cost function. This gives us the following equations:

$$\begin{cases} (10c_{11}^2 + 2)\Delta r_1 + 10c_{11}c_{21}\Delta r_2 + O = 0 \\ (10c_{31}^2 + 2)\Delta r_3 + 10c_{22}c_{31}\Delta r_2 + P = 0 \\ 10c_{11}c_{21}\Delta r_1 + (10c_{21}^2 + 10c_{22}^2 + 2)\Delta r_2 + 10c_{22}c_{31}\Delta r_3 + Q = 0 \end{cases} \tag{B.4}$$

where

$$\begin{aligned} O &= 6c_{11}u_1 - 6c_{11}B_1 + (2c_{11}\lambda + 4c_{11}\lambda^2)(B_1 - u_1) - 2\Delta r_1(k-1) \\ P &= 6c_{31}u_2 - 6c_{31}B_2 + (2c_{31}\lambda + 4c_{31}\lambda^2)(B_2 - u_2) - 2\Delta r_3(k-1) \\ Q &= 6c_{21}u_1 + 6c_{22}u_2 - 6c_{21}B_1 - 6c_{22}B_2 + (2c_{21}\lambda + 4c_{21}\lambda^2)(B_1 - u_1) \\ &\quad + (2c_{22}\lambda + 4c_{22}\lambda^2)(B_2 - u_2) - 2\Delta r_2(k-1) \end{aligned}$$

We compute $\mathbf{\Delta r}(k)$ by solving (B.4), and then substitute it to the actual system model (4.3-4.4). The closed-loop model is a function of the system gains $(g_1, g_2)$.

$$\begin{aligned} u_1(k+1) &= (g_1(\frac{2\lambda + 4\lambda^2}{6} + \frac{2c_{21}}{3}\frac{12c_{21} - 2c_{21}(2\lambda + 4\lambda^2)}{32c_{21}^2 + 32c_{22}^2} - 1) + 1)u_1(k) \\ &\quad + g_1\frac{2c_{21}}{3}\frac{12c_{22} - 2c_{22}(2\lambda + 4\lambda^2)}{32c_{21}^2 + 32c_{22}^2}u_2(k) + M \\ u_2(k+1) &= g_2\frac{2c_{22}}{3}\frac{12c_{21} - 2c_{21}(2\lambda + 4\lambda^2)}{32c_{21}^2 + 32c_{22}^2}u_1(k) \end{aligned}$$

$$+(g_2(\frac{2\lambda+4\lambda^2}{6}+\frac{2c_{22}}{3}\frac{12c_{22}-2c_{22}(2\lambda+4\lambda^2)}{32c_{21}^2+32c_{22}^2}-1)+1)u_2(k)+N$$

where $M, N$ are independent of $u_1(k)$ or $u_2(k)$. Hence the matrix $A$ in (4.8) is

$$A=\begin{bmatrix} g_1(\frac{c}{6}+\frac{2c_{22}a}{3}-1)+1 & g_1\frac{2c_{21}b}{3} \\ g_2\frac{2c_{22}a}{3} & g_2(\frac{c}{6}+\frac{2c_{22}b}{3}-1)+1 \end{bmatrix} \tag{B.5}$$

where $a=\frac{12c_{21}-2c_{21}c}{32c_{21}^2+32c_{22}^2}, b=\frac{12c_{22}-2c_{22}c}{32c_{21}^2+32c_{22}^2}$, and $c=2\lambda+4\lambda^2$.

Since $\mathbf{A}$ is not a function of $c_{11}$ or $c_{31}$, the stability of the closed-loop system only depends on the values of $c_{21}$ and $c_{22}$. This is because both $T_1$ and $T_3$ are local tasks, i.e., each of them only has one subtask and hence only runs on a single processor. The controller can adjust the rates of $T_1$ and $T_3$ to control the utilization on a processor without affecting the other one. Therefore, only the parameters of the end-to-end task, $T_2$, affect system stability.

The closed-loop system is stable if the eigenvalues of $\mathbf{A}$ locate inside the unit circle in the complex space. The eigenvalues of $\mathbf{A}$ are

$$\rho=\frac{\frac{c}{6}(g_1+g_2)+\frac{2}{3}(c_{21}ag_1+c_{22}bg_2)-g_1-g_2+2\pm\omega^{1/2}}{2}$$

where $\omega=(\frac{c-6}{6})(g_1-g_2)^2+\frac{4}{3}(c_{21}ag_1-c_{22}bg_2)\frac{c-6}{6})(g_1-g_2)+\frac{4}{9}(c_{21}ag_1-c_{22}bg_2)^2$

Following Step 3, we can establish the condition in terms of $(g_1, g_2)$ that guarantees the stability of the closed-loop system. For example, in the special case when $g_1=g_2=g$,

$$\begin{cases} \rho_1=\frac{2\lambda^2+\lambda-3}{3}g+1 \\ \rho_2=\frac{2\lambda^2+\lambda-3}{4}g+1 \end{cases} \tag{B.6}$$

To guarantee stability, we need to guarantee $-1<\rho_1,\rho_2<1$. Substituting the values of $\lambda, T_s$,and $T_{ref}$, the stability condition of the closed-loop system is $0<g<5.95$. Therefore, EUCON can maintain stability even if the execution time of every subtask becomes as high as 5.95 times its estimated one.

# Appendix C

# Parameters of MEDIUM used in Section 4.4

The execution time of every subtask $T_{ij}$ in MEDIUM follows a uniform distribution in a range $[Min_{ij}, Max_{ij}] * etf$, where $etf$ is the current execution time factor used in the experiment. The second column (*Proc*) represents the processor where a subtask is located.

Table C.1: Parameters of the MEDIUM workload

| $\mathbf{T_{ij}}$ | **Proc** | $\mathbf{Min_{ij}}$ | $\mathbf{Max_{ij}}$ | $\mathbf{1/R_{max,i}}$, | $\mathbf{1/R_{min,i}}$ | $\mathbf{1/r_i(0)}$ | **phase** |
|---|---|---|---|---|---|---|---|
| $T_{11}$ | $P_1$ | 25 | 35 | 55 | 3000 | 300 | 0 |
| $T_{12}$ | $P_2$ | 45 | 55 | | | | |
| $T_{13}$ | $P_3$ | 45 | 55 | | | | |
| $T_{14}$ | $P_4$ | 35 | 45 | | | | |
| $T_{21}$ | $P_4$ | 45 | 55 | 55 | 5000 | 500 | 100 |
| $T_{22}$ | $P_2$ | 35 | 45 | | | | |
| $T_{31}$ | $P_1$ | 55 | 65 | 65 | 4000 | 400 | 0 |
| $T_{32}$ | $P_3$ | 35 | 45 | | | | |
| $T_{41}$ | $P_1$ | 25 | 35 | 45 | 6000 | 600 | 200 |
| $T_{42}$ | $P_4$ | 35 | 45 | | | | |
| $T_{43}$ | $P_2$ | 15 | 25 | | | | |
| $T_{51}$ | $P_4$ | 105 | 115 | 115 | 10000 | 1000 | 200 |
| $T_{52}$ | $P_2$ | 65 | 75 | | | | |
| $T_{53}$ | $P_3$ | 55 | 65 | | | | |
| $T_{61}$ | $P_1$ | 25 | 35 | 55 | 4000 | 400 | 0 |
| $T_{62}$ | $P_2$ | 45 | 55 | | | | |
| $T_{63}$ | $P_1$ | 35 | 45 | | | | |
| $T_{71}$ | $P_4$ | 55 | 65 | 105 | 6000 | 600 | 100 |
| $T_{72}$ | $P_3$ | 95 | 105 | | | | |
| $T_{81}$ | $P_2$ | 65 | 75 | 75 | 5000 | 500 | 0 |
| $T_{82}$ | $P_1$ | 35 | 45 | | | | |
| $T_{91}$ | $P_1$ | 35 | 45 | 45 | 5000 | 500 | 0 |
| $T_{10,1}$ | $P_2$ | 35 | 45 | 45 | 6000 | 600 | 0 |
| $T_{11,1}$ | $P_3$ | 35 | 45 | 45 | 4000 | 400 | 0 |
| $T_{12,1}$ | $P_4$ | 35 | 45 | 45 | 6500 | 650 | 0 |

# Appendix D

# Detailed Stability Analysis in DEUCON

**Example** To illustrate our method for stability analysis, we now apply the stability analysis approach to the example system described in Figure 5.2. The system has 21 tasks and 10 processors. We set the prediction horizon $P = 2$ and the control horizon $M = 1$. The time constant of the reference trajectory is $T_{ref}/T_s = 4$. The weights on all terms are 1. The parameters in the model for the controller on Processor $P_1$ are

$$
\begin{aligned}
\mathbf{nu_1'(k)} &= \left[ \begin{array}{ccc} u_1'(k) & u_2'(k) & u_3'(k) \end{array} \right]^T \\
&= \left[ \begin{array}{ccc} u_1(k) & e^{-\frac{T_s}{T_{ref}}} u_2(k) + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) B_2 & e^{-\frac{T_s}{T_{ref}}} u_3(k) + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) B_3 \end{array} \right]^T \\
\mathbf{G_1} &= \left[ \begin{array}{ccc} g_1 & 0 & 0 \\ 0 & g_2 & 0 \\ 0 & 0 & g_3 \end{array} \right] \\
\mathbf{F_1} &= \left[ \begin{array}{cccccccc} c_{11} & c_{21} & c_{31} & c_{42} & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{32} & c_{41} & c_{51} & c_{62} & 0 & 0 \\ 0 & c_{22} & c_{33} & 0 & 0 & c_{61} & c_{71} & c_{83} \end{array} \right] \\
\mathbf{\Delta r_1(k)} &= \left[ \begin{array}{cccccccc} \Delta r_1(k) & \Delta r_2(k) & \Delta r_3(k) & \Delta r_4(k) & \Delta r_5(k) & \Delta r_6(k) & \Delta r_7(k) & \Delta r_8(k) \end{array} \right]^T \\
\mathbf{B_1} &= \left[ \begin{array}{ccc} B_1 & B_2 & B_3 \end{array} \right]^T
\end{aligned}
$$

The solution for the controller on $P_1$ is of the form

$$\mathbf{\Delta nr_1^1}(\mathbf{k}) = \begin{bmatrix} k_{11}^1 & k_{12}^1 & k_{13}^1 \\ \vdots & \vdots & \vdots \\ k_{81}^1 & k_{82}^1 & k_{83}^1 \end{bmatrix} \mathbf{nu_1'}(\mathbf{k}) + \begin{bmatrix} h_{11}^1 & \cdots & h_{18}^1 \\ \vdots & \ddots & \vdots \\ h_{81}^1 & \cdots & h_{88}^1 \end{bmatrix} \mathbf{\Delta nr_1}(\mathbf{k-1}) + \begin{bmatrix} e_{11}^1 & e_{12}^1 & e_{13}^1 \\ \vdots & \vdots & \vdots \\ e_{81}^1 & e_{82}^1 & e_{83}^1 \end{bmatrix} \mathbf{B_1} \tag{D.1}$$

The superscript 1 denotes that the solution is for the controller on $P_1$.

Following Step 2, we construct the feedback and feed-forward matrices for (9). Since controller $C_1$ manipulates the control variables $\Delta r_1$, $\Delta r_2$ and $\Delta r_3$, the first three rows of the matrices $\mathbf{K}$ and $\mathbf{L}$ are constructed by the first three rows of $\mathbf{K_1}$ as

$$\begin{bmatrix} k_{11}^1 & 0 & 0 & 0 & \cdots & 0 \\ k_{21}^1 & 0 & 0 & 0 & \cdots & 0 \\ k_{31}^1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

and

$$\begin{bmatrix} 0 & e^{-\frac{T_s}{T_{ref}}} k_{12}^1 & e^{-\frac{T_s}{T_{ref}}} k_{13}^1 & 0 & \cdots & 0 \\ 0 & e^{-\frac{T_s}{T_{ref}}} k_{22}^1 & e^{-\frac{T_s}{T_{ref}}} k_{23}^1 & 0 & \cdots & 0 \\ 0 & e^{-\frac{T_s}{T_{ref}}} k_{32}^1 & e^{-\frac{T_s}{T_{ref}}} k_{33}^1 & 0 & \cdots & 0 \end{bmatrix},$$

respectively. The first three rows of the matrix $\mathbf{E}$ are constructed by the first three rows of $\mathbf{E_1}$ and $\mathbf{K_1}$ as

$$\begin{bmatrix} e_{11}^1 & e_{12}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{12}^1 & e_{13}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{13}^1 & 0 & \cdots & 0 \\ e_{21}^1 & e_{22}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{22}^1 & e_{23}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{23}^1 & 0 & \cdots & 0 \\ e_{31}^1 & e_{32}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{32}^1 & e_{33}^1 + \left(1 - e^{-\frac{T_s}{T_{ref}}}\right) k_{33}^1 & 0 & \cdots & 0 \end{bmatrix}.$$

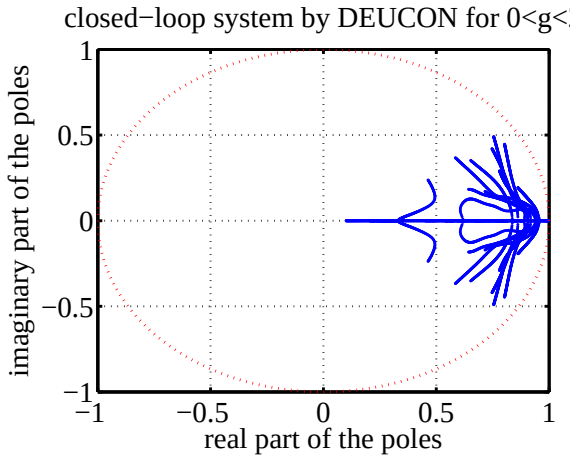



Figure D.1: The root locus of the closed-loop system

The first three rows of the matrix $\mathbf{H}$ are constructed by the first three rows of the matrix $\mathbf{H_1}$ as

$$\begin{bmatrix} h_{11}^1 & \cdots & h_{18}^1 & 0 & \cdots & 0 \\ h_{21}^1 & \cdots & h_{28}^1 & 0 & \cdots & 0 \\ h_{31}^1 & \cdots & h_{38}^1 & 0 & \cdots & 0 \end{bmatrix}.$$

The matrices $\mathbf{K}$, $\mathbf{H}$ and $\mathbf{E}$ can be completed by the corresponding matrices from controllers on other processors. Then, we can derive the composite system (10).

The poles are functions of the system gains in $\mathbf{G}$. The closed-loop system has 31 poles. Our MATLAB program allows us to analyze the system stability under any $\mathbf{G}$. For example, Figure D.1 shows the root locus of the closed-loop system by DEUCON for the case that all non-zero elements of $\mathbf{G}$ have the same value, denoted by $g$. Root locus is the trajectory of the poles of the closed-loop system as $g$ varies. The dotted circle is the unit circle. It shows that all poles are within the unit circle for $0 < g < 2$. Furthermore, the DC gain of the closed-loop system is the identity matrix for $0 < g < 2$. Therefore, the system is stable. Our analysis proves that DEUCON can provide robust utilization guarantees to the example system even when actual execution times deviate significantly from the estimation. For instance, our results indicate that DEUCON can converge to the desired utilizations on all processors even if the execution time of every task is 90% lower ($g = 0.1$) or 90% higher ($g = 1.9$) than the estimation as long as the range of task rates are not violated.

# References

[1] Sherif Abdelwahed, Sandeep Neema, Joseph P. Loyall, and Richard Shapiro. A hybrid control design for QoS management. In *RTSS*, 2003.

[2] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in sofware services. *IEEE Control Systems*, 23(3), June 2003.

[3] Tarek F. Abdelzaher, Vivek Sharma, and Chenyang Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. Comput.*, 53(3):334–350, 2004.

[4] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[5] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *IEEE RTSS*, December 2002.

[6] Shoukat Ali, Jong-Kook Kim, Yang Yu, Shriram B. Gundala, Sethavidh Gertphol, Howard Jay Siegel, and Anthony A. Maciejewski. Utilization-based techniques for statically mapping heterogeneous applications onto the hiper-d heterogeneous computing system, 2003.

[7] Mehdi Amirijoo, Nicolas Chaufette, Jörgen Hansson, Sang Hyuk Son, and Svante Gunnarsson. Generalized performance management of multi-class real-time imprecise data services. In *RTSS*, pages 38–49, 2005.

[8] Sanjoy Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. *rtss*, 00:37–46, 2004.

[9] Lotfi Benmohamed and Semyon M. Meerkov. Feedback control of congestion in packet switching networks: the case of a single congested node. *IEEE/ACM Trans. Netw.*, 1(6):693–708, 1993.

[10] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *IEEE RTSS*, December 1998.

[11] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Syst.*, 23(1-2):7–24, 2002.

[12] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. Comput.*, 51(3):289–302, 2002.

[13] M. Caccamo, G. Buttazzo, and L. Sha. Elastic feedback control. In *Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[14] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Handling execution overruns in hard real-time control systems. *IEEE Trans. Comput.*, 51(7):835–849, 2002.

[15] E. F. Camacho and C. Bordons. *Model Predictive Control.* Springer Verlag, London, 1999.

[16] E. Camponogara, D. Jia, B.H. Krogh, and S.N. Talukdar. Distributed model predictive control. *Control Systems Magazine*, 22(1):44–52, February 2002.

[17] R. Carlson. Sandia SCADA program high-security SCADA LDRD final report. *SANDIA Report SAND2002-0729*, 2002.

[18] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[19] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Arzen. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1):25–53, July 2002.

[20] D. Corman. WSOA-Weapon Systems Open Architecture Demonstration-Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution. In *Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2001.

[21] Y Diao, N Gandhi, JL Hellerstein, S Parekh, and DM Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management*, 2002.

[22] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay S. Parekh, and Christian Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 376, 2004.

[23] J. Eker. *Flexible Embedded Control Systems. Design and Implementation.* PhD thesis, Lund Institute of Technology, Sweden, December 1999.

[24] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems, 3rd edition.* Addition-Wesley, 1997.

[25] Gene F. Franklin, Abbas Emami-Naeini, and J. David Powell. *Feedback Control of Dynamic Systems, 3rd edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[26] Felix C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[27] Sethavidh Gertphol, Yang Yu, Shriram B. Gundala, Viktor K. Prasanna, Shoukat Ali, Jong-Kook Kim, Anthony A. Maciejewski, and Howard Jay Siegel. A metric and mixed-integer-programming-based approach for resource allocation in dynamic real-time systems. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 16, Washington, DC, USA, 2002. IEEE Computer Society.

[28] C.D. Gill, D.C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[29] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 20(2), March 2001.

[30] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London, UK, 1981.

[31] A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *IEEE RTAS*, 2004.

[32] Aniruddha S. Gokhale, Balachandran Natarajan, Douglas C. Schmidt, and Joseph K. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing*, 7(4):331–346, 2004.

[33] T.H. Harrison, D.L. Levine, and D.C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997.

[34] D. Henriksson and T. Olsson. Maximizing the use of computational resources in multi-camera feedback control. In *IEEE RTAS*, May 2004.

[35] C.V. Hollot, V. Misra, D. Towsley, , and W. Gong. A control theoretic analysis of RED. In *Proceedings of INFOCOM 2001*, April 2001.

[36] Chao-Ju Hou and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *IEEE Trans. Comput.*, 46(12):1338–1356, 1997.

[37] J. Huang, Y. Wang, and F. Cao. On developing distributed middleware services for qos-and criticality-based resource negotiation and adaptation. *Real-Time Syst.*, 16(2-3):187–221, 1999.

[38] S. Hutchinson, G. Hager, and P. Corke. A tutorial on visual servo control. *IEEE Trans. on Robotics and Automation*, 12(5):651–670, October 1996.

[39] Information Networks Division, Hewlett-Packard, Cupertino, CA. *Netperf: A Network Performance Benchmark*, March 1993. Edition B.

[40] Liu J., Lin K. J., Shih W. K., and Yu A. C. Algorithms for Scheduling Imprecise Computations. In *ONR Third Annual Workshop on the Foundations of Real-Time Computing*, 1990.

[41] J. Huang and R. Jha and W. Heimerdinger and M. Muhammad and S. Lauzac and B. Kannikeswaran and K. Schwan and W. Zhao and R. Bettati. RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications. In *Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*, San Francisco, California, 1997. IEEE.

[42] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Trans. Parallel Distrib. Syst.*, 8(12):1268–1274, 1997.

[43] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, 2005.

[44] Magnus Karlsson, Christos T. Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *TOS*, 1(4):457–480, 2005.

[45] Srinivasan Keshav. A control-theoretic approach to flow control. In *SIGCOMM*, pages 3–15, 1991.

[46] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. Towards highly configurable real-time object request brokers. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 437–447, 2002.

[47] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[48] Xenofon Koutsoukos, Radhika Tekumalla, Balachandran Natarajan, and Chenyang Lu. Hybrid supervisory utilization control of real-time systems. In *IEEE RTAS*, 2005.

[49] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jeff Hansen. A scalable solution to the multi-resource qos problem. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 315, Washington, DC, USA, 1999. IEEE Computer Society.

[50] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadline. In *IEEE RTSS*, 1990.

[51] F.L. Lewis and V.L. Syrmos. *Optimal Control, Second Edition.* John Wiley & Sons, Inc., 1995.

[52] B. Li and K. Nahrstedt. A Control-based Middleware Framework for QoS Adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, September 1999.

[53] Suzhen Lin and G. Manimaran. Double-loop feedback-based scheduling approach for distributed real-time systems. In *HiPC*, pages 268–278, 2003.

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

[55] Jane W. S. Liu. *Real-Time Systems.* Prentice Hall, 2000.

[56] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 625–634. IEEE, April 2001.

[57] 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, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2):85–126, July 2002.

[58] C. Lu, X. Wang, and C.D. Gill. Feedback control real-time scheduling in ORB middleware. In *IEEE RTAS*, May 2003.

[59] C. Lu, X. Wang, and X. Koutsoukos. End-to-end utilization control in distributed real-time systems. In *ICDCS 2004*, Tokyo, Japan, March 2004.

[60] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):550–561, June 2005.

[61] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 21, Berkeley, CA, USA, 2002. USENIX Association.

[62] Ying Lu, Chanyang Lu, Tarek Abdelzaher, and Gang Tao. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, Miami Beach, FL, May 2002.

[63] J.M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002.

[64] Pau Marti, Gerhard Fohler, Pep Fuertes, and Krithi Ramamritham. Improving quality-of-control using flexible timing constraints: metric and scheduling. In *IEEE RTSS*, 2002.

[65] J.-F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Trans. Computers*, 29(8):720–731, August 1980.

[66] M. Di Natale and J. Stankovic. Dynamic end-to-end guarantees in distributed real-time systems. In *IEEE RTSS*, 1994.

[67] Object Management Group. *Real-Time CORBA Specification*, 1.1 edition, August 2002.

[68] C. O'Ryan, D.C. Schmidt, and J.R. Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2), March 2002.

[69] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[70] William K. Pratt. *Digital image processing*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[71] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7), July 2003.

[72] R. Collins. JTIDS: Joint Tactical Information Distribution System. www.rockwellcollins.com/ecat/gs/JTIDS.html.

[73] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE RTAS*, December 1988.

[74] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 320, Washington, DC, USA, 1997. IEEE Computer Society.

[75] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 48–58, New York, NY, USA, 2004. ACM Press.

[76] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[77] D. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: AddisonWesley.*, 1995.

[78] D. Schmidt. Acceptor and connector: Design patterns for initializing communication services. *Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley.*, 1997.

[79] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* $12^{th}$ *Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, December 1993. SUG.

[80] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[81] Douglas C. Schmidt and et al. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[82] Douglas C. Schmidt and Fred Kuhns. An overview of the real-time CORBA specification. *IEEE Computer*, 33(6):56–63, 2000.

[83] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control system. In *IEEE RTSS*, Washington, D.C., December 1996.

[84] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, Kevin Skadron, and Zhijian Lu. Power-aware QoS management in web servers. In *IEEE RTSS*, 2003.

[85] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 4(4):382–397, 1993.

[86] R. W. Shields and J. B. Pearson. Structural controllability of multiinput linear systems. *IEEE Transactions on Automatic Control*, AC-21:203–212, 1976.

[87] Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 17, Washington, DC, USA, 2002. IEEE Computer Society.

[88] John A. Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son, and Cenyan Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, 2001.

[89] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158, 1999.

[90] Venkita Subramonian, Guoliang Xing, Christopher D. Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. In *RTAS*, 2004.

[91] Jun Sun and Jane W.-S. Liu. Synchronization protocols in distributed real-time systems. In *ICDCS*, 1996.

[92] G. K. Wallace. The jpeg still image compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[93] Xiaorui Wang, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu, and Christopher D. Gill. CAMRIT: Control-based adaptive middleware for real-time image transmission. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 296–305, 2004.

[94] Xiaorui Wang, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos. Decentralized utilization control in distributed real-time systems. In *IEEE RTSS*, 2005.

[95] Xiaorui Wang, Charles Lefurgy, and Malcolm Ware. Managing Peak System-level Power with Feedback Control. Technical Report IBM RC23835, IBM Research, 2005.

[96] Xiaorui Wang, Chenyang Lu, and Xenofon Koutsoukos. Enhancing the robustness of distributed real-time middleware via end-to-end utilization control. In *IEEE RTSS*, 2005.

[97] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In *DSOM*, pages 133–144, 2005.

[98] L. Welch. Adaptive resource management for scalable, dependable real-time systems. In *Work-In Progress Session of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, 1998.

[99] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman. DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable Real-Time Systems. In *IFACs 15th Symposium on Distributed Computer Control Systems (DCCS98)*. IFAC, 1998.

[100] W. J. Wilson. Applying layering principles to legacy systems: Link 16 as a case study. In *IEEE International Military Communications Conference (MILCOM)*, 2001.

[101] A. D. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10):54–62, 2002.

[102] Qiang Wu, Philo Juang, Margaret Martonosi, Li-Shiuan Peh, and Douglas W. Clark. Formal control techniques for power-performance management. *IEEE Micro*, 25(5):52–62, 2005.

[103] Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, Inc., 1997.

[104] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002. IEEE.

[105] Feng Zhao, Xenofon D. Koutsoukos, Horst W. Haussecker, James Reich, Patrick Cheung, and Claudia Picardi. Distributed monitoring of hybrid systems: A model-directed approach. In *IJCAI*, 2001.

[106] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *IEEE RTAS*, 2004.

[107] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

# Vita

Xiaorui Wang

**Degrees**

Bachelor of Science in Computer Science, July 1995
Master of Science in Computer Science, May 1998
Master of Science in Computer Science, August 2002
Doctor of Science in Computer Science, August 2006

**Industrial Experience**

- Research Intern, IBM Austin Research Lab, Power-Aware Systems Department, Austin, TX, Summer 2005.

- Senior Software Engineer, Huawei Technologies Co., Ltd, Central R&D, Optical Networks Department, Shenzhen, China, 1998-2001.

**Publications**

- **Xiaorui Wang**, Yingming Chen, Chenyang Lu, and Xenofon Koutsoukos, *FC-ORB: A Robust Distributed Real-time Embedded Middleware with End-to-End Utilization Control,* to appear in the Elsevier Journal of Systems and Software, Special Issue on Dynamic Resource Management in Distributed Real-Time Systems.

- **Xiaorui Wang**, Dong Jia, Chenyang Lu and Xenofon Koutsoukos, *Decentralized Utilization Control in Distributed Real-Time Systems,* the 26th IEEE Real-Time Systems Symposium (RTSS 2005), Miami, Florida, December 2005.

- **Xiaorui Wang**, Chenyang Lu and Xenofon Koutsoukos, *Enhancing the Robustness of Distributed Real-Time Middleware via End-to-End Utilization Control,* the 26th IEEE Real-Time Systems Symposium (RTSS 2005), Miami, Florida, December 2005.

- Chenyang Lu, **Xiaorui Wang**, and Xenofon Koutsoukos, *Feedback Utilization Control in Distributed Real-Time Systems with End-to-End Tasks,* IEEE Transactions on Parallel

and Distributed Systems (IEEE TPDS), 16(6):550-561, June 2005.

- Guoliang Xing, **Xiaorui Wang**, Yuanfang Zhang, Chenyang Lu, Robert Pless, and Christopher Gill, *Integrated Coverage and Connectivity Configuration for Energy Conservation in Sensor Networks,* ACM Transactions on Sensor Networks (ACM TOSN), 1(1): 36-72, August 2005.

- **Xiaorui Wang**, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu, and Christopher Gill. *CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission,* IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), Toronto, Canada, May 2004.

- Chenyang Lu, **Xiaorui Wang**, and Xenofon Koutsoukos, *End-to-End Utilization Control in Distributed Real-Time Systems,* The 24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, March 2004.

- **Xiaorui Wang**, Guoliang Xing, Yuanfang Zhang, Chenyang Lu, etc. *Integrated Coverage and Connectivity Configuration in Wireless Sensor Networks,* The First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), Los Angeles, CA, November 2003.

- Chenyang Lu, **Xiaorui Wang**, and Christopher Gill, *Feedback Control Real-Time Scheduling in ORB Middleware,* IEEE RTAS 2003, Washington DC, May 2003.

- Octav Chipara, Zhimin He, Guoliang Xing, Qin Chen, **Xiaorui Wang**, Chenyang Lu, John Stankovic, and Tarek Abdelzaher, *Real-time Power Aware Routing in Wireless Sensor Networks,* IEEE International Workshop on Quality of Service (IWQoS 2006), New Haven, CT, June 2006.

**Under Review**

- **Xiaorui Wang**, Charles Lefurgy, and Malcolm Ware, *Performance Optimization within a Power Constraint using Precision Measurement and Feedback Control,* IBM Tech Report RC23835, submitted to the International Symposium on High-Performance Computer Architecture (HPCA 2007).

- **Xiaorui Wang**, Dong Jia, Chenyang Lu, and Xenofon Koutsoukos, *DEUCON: A Decentralized Utilization Control Service for Distributed Real-Time Systems,* submitted to IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS).

- **Xiaorui Wang**, Chenyang Lu and Christopher Gill, *A Feedback Control Real-Time Scheduling in Object Request Broker Middleware,* submitted to Real-Time Systems Journal (RTSJ).

August 2006