New Jersey Institute of Technology
Digital Commons @ NJIT

Theses

Theses and Dissertations

Fall 1996

Real-time implementation of an adaptive control system for a 3-zone rapid thermal processing station

David Hur
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Electrical and Electronics Commons

Recommended Citation
Hur, David, "Real-time implementation of an adaptive control system for a 3-zone rapid thermal processing station" (1996). *Theses*. 1055.
https://digitalcommons.njit.edu/theses/1055

This Thesis is brought to you for free and open access by the Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

REAL-TIME IMPLEMENTATION OF AN ADAPTIVE CONTROL SYSTEM FOR A 3-ZONE RAPID THERMAL PROCESSING STATION

by
David Hur

In this thesis, the implementation details of a real time adaptive control system for a TI 3-zone RTP station, as well as the simulation and the experimental results are presented. Extensive simulation of the system is performed in order to ensure proper operation of the system. The experimental results are used to verify proper operation of the closed loop control system. Initial experiments were conducted using two thermocouples. Further experiments were conducted using one thermocouple for the purpose of testing the performance of the system using Extended Kalman Filter as the state estimator.

The implementation of the control system is carried out on an IBM compatible PC hosting a Transputer parallel-processing system. The motivation for utilizing the parallel processing system is to ensure future extensibility of the system. The eventual incorporation of a remote temperature sensing method such as Multi-Wavelength Imaging Pyrometer (M-WIP) will require great deal of computing power from the system. The implementation of the software for the system is also carried out with goal of providing ease of maintenance and extensibility. The implementation of graphical user environment also provides to the user point and click operation of the system as well as real time plotting capability.

REAL-TIME IMPLEMENTATION OF AN ADAPTIVE CONTROL SYSTEM FOR A 3-ZONE RAPID THERMAL PROCESSING STATION

by
David Hur

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

October 1996

APPROVAL PAGE

REAL TIME IMPLEMENTATION OF AN ADAPTIVE CONTROL SYSTEM FOR A 3-ZONE RAPID THERMAL PROCESSING STATION

David Hur

Dr. Bernard Friedland, Thesis Advisor — Date
Distinguished Professor of Electrical and Computer Engineering, NJIT

Dr. Timothy N. Chang — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. N. M. Ravindra — Date
Associate Professor of Physics, NJIT

BIOGRAPHICAL SKETCH

Author: David Hur

Degree: Master of Science

Date: October 1996

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology,
  Newark, NJ, 1996

- Bachelor of Science in Computer Engineering,
  New Jersey Institute of Technology,
  Newark, NJ, 1994

Major: Electrical Engineering

ACKNOWLEDGMENT

I express my sincere gratitude to Dr. Bernard Friedland for his support and guidance. I also express my deep gratitude to Sergey Belikov for his expert guidance in experimenting with the RTP system, and on whose theoretical work much of this thesis is based on. Without their support and guidance, this thesis would not be possible.

I also like to thank National Science Foundation for their support. The entirity of this experiment was supported by the National Science Foundation under Grant ECS-9312451.

I would also like to express my appreciation to Dr. Timothy Chang and Dr. N. M. Ravindra for serving as committee members. Also a thanks is in order to M. Kaplinsky for teaching us the operations procedure of the RTP station.

This thesis is dedicated to my family

TABLE OF CONTENTS

Chapter	Pages
1 INTRODUCTION	1
1.1 Rapid Thermal Processing	1
1.2 Performance Criteria	2
1.3 Description of Experimental RTP Station	3
1.4 Experiment Objective	5
2 MODEL BASED ADAPTIVE CONTROL ALGORITHM	6
2.1 Introduction	6
2.2 Dynamic Model of Heat Transfer in the Experimental RTP system	6
2.2 Adaptive Control Algorithm for the Experimental RTP Station	10
3 IMPLEMENTATION OF ADAPTIVE CONTROLLER	13
3.1 Transputer System	13
3.2 Programming for Transputer System	15
3.3 Hardware Architecture of the Adaptive Controller	18
3.4 Software Architecture of the Adaptive Controller	20
3.5 Simulation System	24
3.6 Experimental System	25
4 SIMULATION AND EXPERIMENTAL RESULTS	27
4.1 Preliminary Simulation Results	27

TABLE OF CONTENTS

(Continued)

Chapter **Pages**

4.2 Preliminary Experimental Results 31

5 STATE ESTIMATION USING EXTENDED KALMAN FILTER 37

5.1 Motivation 37

5.2 Extended Kalman Filter 38

5.3 Simulation Results 41

5.4 Experimental Results 46

6 ESTIMATION OF EMISSIVITY 56

6.1 Background 56

6.2 Theory 56

6.3 Results of the Experiment 59

7 CONCLUSION 62

APPENDIX A Adaptive Controller Host Program 64

APPENDIX B Adaptive Controller Transputer Program 82

APPENDIX C Operational Procedure for RTP System 106

BIBLIOGRAPHY 109

LIST OF TABLES

Table **Page**

2.1 Temperature dependent specific heat and thermal conductivity for silicon 9

2.2 Temperature dependent emissivity for silicon 9

LIST OF FIGURES

Figure		Page
1.1	Schematic of a TI 3zone RTP station	3
1.2	Photo of Lamp Rings	4
1.3	Photo of RTP Chamber	4
3.1	Block Diagram of a Transputer Module [14]	13
3.2	Default Transputer Module Configuration on the Motherboard [12]	14
3.3	Block Diagram of the Transputer Based Implementation of the Adaptive Controller	19
3.4	Graphical User Interface for the Adaptive Controller	22
3.5	Block Diagram of the Simulation System	25
3.6	Block Diagram of the Experimental RTP System	26
4.1	Single Computer Simulation Result	28
4.2	Simulation with Estimator Gain of 2.33e6	29
4.3	Simulation Result with Filtered Temperature Measurements	30
4.4	Experimental Results. Ramp Rate is 20K/s, Parameter Estimator Gains are Set Identically to 1.33e3	33
4.5	Comparison of Measured Temperatures vs. Reference Trajectory. The Ramp Rate is 20K/s, Parameter Estimator Gains are Set to 1.33e3	34
4.6	Experimental Result. Ramp Rate is 10K/s, Parameter Estimator Gains Set Identically to 1.33e3	35
4.7	Comparison of Measured Temperature Against Reference Trajectory. Ramp Rate is 10K/s, Parameter Estimator Gains are Set to 1.33e3	36

LIST OF FIGURES

(Continued)

FIGURE PAGE

5.1 Simulation Result Using Extended Kalman Filter as the State Estimator. Ramp Rate is Set at 10K/s, the Parameter Estimator Gains are Set to 1.33e3 43

5.2 Simulation Result. Difference Between the Measured and Estimated Temperatures. The Ramp Rate is 10K/s, the Parameter Estimator Gains are 1.33e3. 44

5.3 Simulation Result. Difference Between the Measured Temperatures. The Ramp Rate is 10K/s, the Parameter Estimator Gains are 1.33e3 45

5.4 Preliminary Experimental Result of Using EKF as the State Estimator. The Ramp Rate is 10K/s, the Parameter Estimator Gains are 1.33e2 48

5.5 Plot of Reference Tracking Error and Uniformity. The Ramp Rate is 10K/s, the Parameter Estimator Gains are 1.33e2 49

5.6 Experimental Results with Parameter Estimator Gains Set at 2.33e4 50

5.7 Plot of Reference Tracking Error and Uniformity with Parameter Estimator Gains of 2.33e4 51

5.8 Experimental Results with Parameter Estimator Gains Set at 1.33e4 52

5.9 Plot of Reference Tracking Error and Uniformity with Parameter Estimator Gains at 1.33e4 53

5.10 Experimental Results with Parameter Estimator Gains Set at 1.0e4, 2.5e4, 1.33e4 54

5.11 Plot of Reference Tracking Error and Uniformity with Parameter Estimator Gains Set at 1.0e4, 2.5e4, 1.33e4 55

6.1 Temperature Trajectory Due to Impulse Control Function Generated From Experimental Data 59

6.2 Estimation of Parameters in Simulation 60

6.3 Emissivity Function of Estimated Parameters 60

LIST OF FIGURES

(Continued)

FIGURE PAGE

6.4 Estimated Parameters Using Experimental Data 61

6.5 Emissivity Function of Estimated Parameters 61

CHAPTER 1

INTRODUCTION

1.1 Rapid Thermal Processing

Rapid Thermal Processing (RTP) is an advanced semi-conductor wafer processing method promising many improvements over the conventional batch processing methods. The conventional method is to process a batch of wafers simultaneously in a temperature controlled furnace. The thermal processing of the wafers by this procedure is slow, usually taking of minutes or even hours. Also, due to large thermal mass of the furnace, the thermal cost of processing is high. The processing steps such as oxidation occurs only at high temperature, at which the problem of diffusion and thermal shock becomes a factor. Therefore, the length of the time under which the wafers are subjected to high temperature presents a problem as the circuit feature size continues to shrink as is the current trend in the semiconductor industry [1].

By comparison, in the RTP method, each wafer is processed individually. The RTP station performs multiple thermal processing steps in a single chamber by specifying a process temperature trajectory. The rapid thermal processing of the wafer reduces the processing time to in terms of seconds, and also reduces the overall thermal cost of processing a wafer due to smaller thermal mass of the reaction chamber [1]. Another advantage of rapid thermal processing is the inherent flexibility associated with the method. By being able to specify temperature trajectory, many different types of processing operations can be accomplished at a single RTP station. This flexibility makes the RTP method all the more suitable for the programmable factory environment [2].

1.2 Performance Criteria

The two main performance criteria of the RTP station is (1) its ability to closely follow a given process temperature trajectory, and (2) maintaining uniformity of the temperature across the surface of the wafer while following the process temperature trajectory. The close proximity of the wafer being processed to the heat source, usually a set of high power quartz lamps, and the inherent lack of thermal stability inside the processing chamber makes the uniformity issue a difficult one to solve [1].

The measurement of the wafer temperature also presents a challenge to the development of a practical RTP system. For a control strategy to be implemented using feedback, accurate measurement of the temperature at a multitude of points across the surface of the wafer becomes an important issue. Two of the most popular method of measuring the temperature is to use thermocouples and pyrometers. Using thermocouples is unrealistic for manufacturing environment, since the thermocouples must maintain contact with the wafer being processed. Using pyrometers also presents problems: Since pyrometers are a device which measure radiative power rather than direct temperature, the physical property of the wafer under process, namely the emissivity of the wafer, becomes an important parameter. However, the emissivity of a silicon wafer changes as the processing temperature changes. The emissivity will also differ from one wafer to another. Without proper strategy to account for emissivity, the pyrometer measurement can not provide accurate temperature readings to be useful in the temperature control strategy.

1.3 Description of Experimental RTP Station

The issue of temperature uniformity, as mentioned previously, is a critical issue in the design of a RTP station. An approach to solve the problem, presented in [1], is to implement the heating lamp configuration such that each cluster of lamps are independently controlled. A feedback control strategy is then implemented to control the heat flux of each ring such that the overall heat flux profile follows a certain optimal shape desired [2].

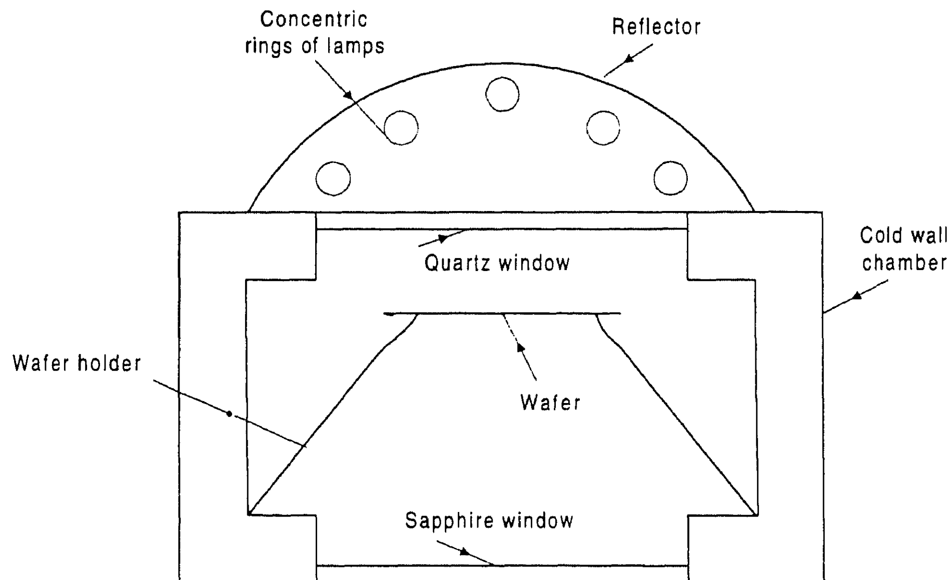

Figure 1.1. Schematic of a TI-3 zone RTP station

The TI-3zone RTP station, shown schematically in Figure 1.1, consists of three rings of tungsten-halogen lamps, each ring controlled by an independent power supply, and surrounded by a reflector. The central ring consist of a single 2kW lamp. The second ring consist of twelve 1kW lamps. The third ring consists of twenty four 1kW lamps. A

quartz window, 0.5" thick, separates the lamp rings from the wafer to be processed. The distance between the top surface of the quartz window to the central lamp ring is 0.56", and to the second lamp ring is 2.13", and to the third lamp ring is 1.25". The wafer is positioned 0.085" below the bottom surface of the quartz window and is held in position at three points by a graphite susceptor. The edges of the quartz window are water cooled to the room temperature, and the surface of the window are cooled by room temperature air blown across the surface. The reaction chamber of the TI 3-zone RTP station is 7.375" in diameter. The walls of the chamber are made of stainless steel and are water cooled to room temperature. The ambient of the chamber may be either a vacuum or a process gas depending on the processing need. The chamber have provisions for a remote infrared temperature sensor. The sapphire window at the bottom of the chamber allows for measurement of radiative energy of the wafer. A remote sensor, such as M-WIP [3] which is under investigation by a team of researchers at NJIT, may be used in the future to provide a solution to the sensor problem mentioned previously.

1.4 Experiment Objective

The research efforts at NJIT during past several years have resulted in, among others, a dynamic heat transfer model of the TI 3 zone RTP station. Using TI 3-zone RTP station, extensive experiments have identified various physical parameters associated with the RTP station. Based on the parameters, the direct model-based adaptive control algorithm for the temperature control strategy is developed.

The objective of the experiment is to implement the control algorithm in an real-time system and to evaluate the performance of the system with respect to the performance criteria as stated previously. A typical goal may be to control the temperature tracking error to be within 1K and to maintain the temperature uniformity across the surface of the wafer within 1K. In the present implementation thermocouples, located along a diameter of the wafer at three points, are used to measure the temperatures. A digital computer hosting a parallel processing system is used to implement the control algorithm and to close the control loop.

CHAPTER 2

MODEL BASED ADAPTIVE CONTROL ALGORITHM

2.1 Introduction

In this chapter a brief summary of theories underlying the dynamic model of the RTP station and the adaptive control algorithm is presented. The theoretical work is described in full detail in [4], [5], and [6], and is presented here for reference only.

2.2 Dynamic Model of Heat Transfer in the Experimental RTP system

The heat transfer mechanism in RTP depends on both the thermal conductivity of the wafer material and the heat radiation mechanism inside the chamber. The radiation mechanism depends on the parameters due to lamp radiation as well as environmental and disturbance heat flux [7]. The heat transfer parameters due to thermal conductivity can be calculated from the known physical properties of the wafer material. However the parameters due to the radiation must be either experimentally determined or estimated on line during the processing steps [5].

The model of heat transfer for a thin silicon wafer in an axi-symmetric RTP system with three rings of heating lamps is represented by the following partial differential equation [4].

$$\rho c(T(r,t))\frac{\partial T(r,t)}{\partial t} = k(T(r,t))\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T(r,t)}{\partial r}\right) - 2h^{-1}\left[E(T(r,t)) + \sum_{l=1}^{3} G_l(r,U_l(t)) + S(r,t)\right] \tag{2.1}$$

$$T(r,0) = T_0(r), \quad 0 \le r \le R \tag{2.2}$$

$$\left. \frac{\partial\, T(r,t)}{\partial\, r} \right|_{r=R} = 0 \tag{2.3}$$

where R is the radius of the wafer, h is the wafer thickness, r is the radial position along the wafer, ρ is the density of the wafer material, and $T(r,t)$ is the temperature of the wafer at point r at time t, $c(T(r,t))$ is the specific heat of the wafer material per unit volume, $k(T(r,t))$ is the thermal conductivity of the wafer material per unit volume. The function $E(T(r,t))$ in (2.1) is the emission power per unit area of the surface of the wafer [5][8]. The emission function $E(T(r,t))$ for a non-black body is shown to be [6][9]

$$E(T) = \sum_{l=l_{\min}}^{l_{\max}} \alpha_l p_l(T) T^{4-l} \tag{2.4}$$

where $p_l(T)$ are the coefficient of the following spectral emissivity model [4][3]

$$\varepsilon\,(\lambda, T) = \sum_{l=l_{\min}}^{l_{\max}} p_l(T) \lambda^{\,l} \tag{2.5}$$

$$l_{\min} = 0, -1, -2, \quad l_{\max} = 0, 1, 2$$
$$\alpha_0 = 5.67 \cdot 10^{-8} \; [W/(m^2 K^4)]$$
$$\alpha_1 = 3.02 \cdot 10^{-10} \; [W/(mK^3)]$$
$$\alpha_2 = 2.97 \cdot 10^{-12} \; [W/K^2]$$
$$\alpha_{-1} = 1.51 \cdot 10^{-5} \; [W/(m^3 K^5)]$$
$$\alpha_{-2} = 5.15 \cdot 10^{-3} \; [W/(m^4 K^6)]$$

The function $U_l(t)$ in (2.1) is the control voltage to the lth ring of the heating lamp at time t in the range from *0* to *5* volts. The function $G_l(r,U)$ of (2.1) is the lth ring radiation function relating the voltage supplied to the lamp ring to the heat power

reaching the surface of wafer [4]. The function $S(r,t)$ denotes unmodeled disturbances such as heat flux to and from the edge of wafer, convection, and other disturbance flux.

The investigation of [8] have shown that the temperature distribution in a semiconductor wafer $T(r,t)$ and heat fluxes $G_l(r,U)$, $S(r,t)$ can be expressed by a 3-coefficient Bessel function expansion. The RTP system with 3 rings of heating lamps therefore have the following representation for the temperature distribution [6]

$$T(r,t) = x_0(t) + x_2(t)\ J_0(\mu_1\, r / R) + x_3(t)\ J_0(\mu_2\, r / R) \tag{2.6}$$

where $x_0(t)$ is the average temperature along the wafer, $x_n(t), n = 2,3$ are the $(n-1)^{th}$ coefficients of the Bessel function expansion of the difference between the temperature of the wafer and the reference temperature $\overline{T}(t)$, and μ_1, μ_2 are solutions of the equation

$$dJ_o(\mu) / d\mu = 0 \quad . \tag{2.7}$$

The state variables $x_n(t), n = 1,2,3$ are the coefficients in Bessel expansion of (2.6) and satisfy the following ordinary differential equations [4].

$$\rho\, c(x_0)\, \dot{x}_0 = -2h^{-1}\ E(x_0) + P_1 \tag{2.8}$$

$$\rho\, c(x_0)\, \dot{x}_2 = -[(\mu_1 / R)^2\ k(x_0) + 2h^{-1}\ E(x_0)]\, x_2 + P_2 \tag{2.9}$$

$$\rho\, c(x_0)\, \dot{x}_3 = -[(\mu_2 / R)^2\ k(x_0) + 2h^{-1}\ E(x_0)]\, x_3 + P_3 \tag{2.10}$$

where

$$P_n = 2h^{-1}[\sum_{l=1}^{3} G_{ln}(U_l) + S_n(t)] \quad , n = 1, 2, 3 \tag{2.11}$$

It was shown in [5] that the function $G_l(r,U)$ can be adequately approximated by the products of "shape" functions $G_l(r)$ which depend only on the radial position r and the "amplitude" function $\tilde{g}_l(r,U)$ that depend only on the voltage U;

$$G_l(r,U) \approx \tilde{G}_l(r)\,\tilde{g}_l(U), \quad l = 1, 2, 3 \tag{2.12}$$

The control signals must be nonnegative and bounded. For control signal to be admissible it must satisfy the following condition [6];

$$\mathbf{0} \le \mathbf{u}(t) \le \mathbf{U}^{\max} = [U_1^{\max}, \cdots, U_3^{\max}]^T \tag{2.13}$$

For the experiment, the silicon wafer used has radius of *R=76.2 mm* and thickness *h=0.635 mm* with three thermocouples imbedded at the center of the wafer, and at points of radii *23 mm*, and *46 mm*. The density of the silicon is $\rho = 2330\ Kg/m^3$ [4]. The temperature dependent specific heat *c(T)* and thermal conductivity $k(T)$ are given in table 1, with emissivity given in table 2.

Table 1.1. Temperature dependent specific heat *c* and thermal conductivity *k* for silicon [12]

$T[K]$	200	400	600	800	1000	1200
$c\,[J/(Kg.K)]$	549	780	856	900	93.4	955
$k\,[W/(m.K)]$	264	98.9	61.2	42.2	31.2	25.7

Table 1.2. Temperature dependent emissivity for silicon [13]

$T[K]$	300	550	700	740	1000
ε	0.2	0.2	0.5	0.7	0.7

2.2 Adaptive Control Algorithm for the Experimental RTP Station

In order to generate control voltages to the lamp ring power supplies, the control algorithm needs the knowledge of the state variables as well as the un-modeled

disturbance parameters of the system. The state variables $x_n(t), n = 1,2,3$ are calculated from the equation (2.6) using the three temperature measurements from the thermocouples. The disturbance parameter vector S is then estimated on line using the parameter-estimation algorithm of [10].

According to the separation principle [11], the controller design can be achieved in two separate steps. First, the controller is designed with the assumption that all the state variables and the parameters are either measurable or known a priori. The second step involves the estimation of the unmeasured state variables and the estimation of the unknown parameters by whatever means available. In the design of the adaptive control algorithm for the experimental RTP system, the controller is initially designed with an assumption that all the state variables are measurable and that the parameter vector S is known. The parameter estimation algorithm is then used to estimate the disturbance parameter vector S.

Assuming that the disturbance parameter vector $\mathbf{S} = \mathbf{S}(t)$ is known, the control function $\bar{\mathbf{u}}(t) = [\bar{u}_1(t), \bar{u}_2(t), \bar{u}_3(t)]^T$ can be calculated from (2.11). This control function makes the system described by (2.1) to follow a given uniform reference trajectory $\bar{T}(t)$. In vector matrix notation the control function is [6];

$$\bar{\mathbf{u}}(t) = \mathbf{G}^{-1}[\bar{\mathbf{P}}(t) - \mathbf{S}(t)] = \mathbf{G}^{-1}\begin{bmatrix} c(\bar{T}(t))[d\bar{T}(t)/dt] + E(\bar{T}(t)) - S_1(t) \\ -S_2(t) \\ -S_3(t) \end{bmatrix} \quad (2.14)$$

The admissibility of the control function is defined by (2.13).

For the feedback control law, let;

$$\Delta \mathbf{u}(t) = \mathbf{u}(t) - \overline{\mathbf{u}}(t) \tag{2.15}$$

and

$$\Delta \mathbf{P}(t) = \mathbf{G}\Delta \mathbf{u}(t) \tag{2.16}$$

In order to obtain the control law, the "feedback linearization" technique is applied to (2.8), (2.9), and (2.10). This reduces to choosing $\Delta \mathbf{P}(t)$ such that after it is substituted into the equations (2.8), (2.9), and (2.10), we get $\dot{x}_n = -\lambda_n x_n \quad ,n = 1, 2, 3$. The corresponding $\Delta \mathbf{u}(t)$ is calculated by (2.16). Thus we have;

$$\mathbf{u}(t,\mathbf{x}) = \overline{\mathbf{u}}(t) + \Delta \mathbf{u}(t) = \mathbf{G}^{-1}(\overline{\mathbf{P}} + \Delta \mathbf{P} - \mathbf{S}) \tag{2.17}$$

In order to account for the constraint (2.13), (2.17) is written in the form;

$$\mathbf{u}(t,\mathbf{x}) = \mathbf{a}_1(\mathbf{x})\lambda_1 + \mathbf{a}_2(\mathbf{x})\lambda_2 + \mathbf{a}_3(\mathbf{x})\lambda_3 + \mathbf{b}(\mathbf{x}) \tag{2.18}$$

The feedback control $\mathbf{u}(t,\mathbf{x})$ is admissible if and only if

$$\mathbf{0} \le \mathbf{u}(t,\mathbf{x}) \le \mathbf{U}^{\max} \tag{2.19}$$

For stability, the eigenvalues λ_i must be positive. Also the physical constraint limits the maximum possible value of the eigenvalues. A set of eigenvalues can be obtained by linear programming.

In order to estimate the disturbance parameter vector $\boldsymbol{S}$, a simple form of parameter estimation algorithm is applied. The estimator is described by the following equations [4];

$$\begin{aligned} \mathbf{P}_n &= \alpha_n \mathbf{x}_n + z_n \\ \dot{z}_n &= \alpha_n \lambda_n(t,\mathbf{x})\mathbf{x}_n \quad ,n = 1, 2, 3 \end{aligned} \tag{2.20}$$

The parameter estimator is a dynamic process with state vector as the input and the estimated vector $\boldsymbol{P}$ as the output. The dynamics of the estimator is described by the following equations;

$$\hat{P}_n(t) = \alpha_n\, x_n(t) + z_n(t), \;\; n = 0,1,2 \tag{2.21}$$

where

$$\rho\, c(x_o(t))\, \dot{z}_o(t) = -\alpha_o[-E(x_o(t))x_o^4(t) + \hat{P}_o(t)] \tag{2.22}$$

$$\rho\, c(x_o(t))\dot{z}_n(t) = -\alpha_n[-[(\mu_n / R)^2 k(x_o(t)) + 4E(x_o(t))x_n^3]x_n(t) + \hat{P}_n(t)]\,, n = 1,2$$

The disturbance vector $\boldsymbol{S}$ can then be calculated from the estimated $\boldsymbol{P}$ using (2.11).

CHAPTER 3

IMPLEMENTATION OF ADAPTIVE CONTROLLER

3.1 Transputer System

For the purpose of implementing the control algorithm, an IBM compatible PC hosting a transputer motherboard is used. Transputers are general purpose microprocessors with added provisions for connectivity to other transputer microprocessors. Typically, the transputers are packaged as a module, called a TRAM, each containing one or more transputers with additional memory and electronics necessary for interfacing to other transputer modules. A TRAM need not contain a transputer microprocessor; it must however conform to the electronic standard set forth by INMOS® with respect to connectivity to other TRAMs on the motherboard [14].

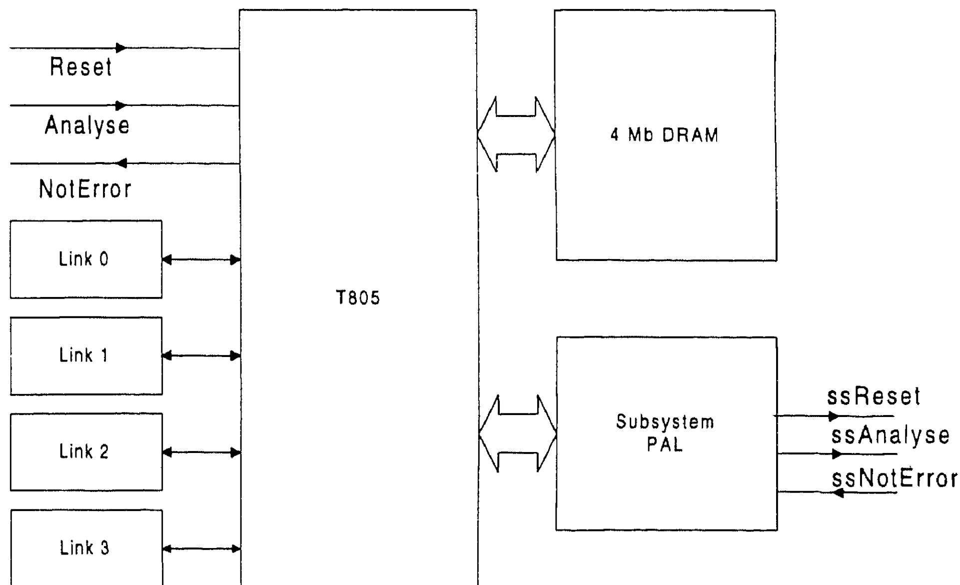

Figure 3.1. Block diagram of a transputer module [14]

As shown in Figure 3.1, the transputer modules used for the experiment consists of a T805 32-bit microprocessor with a 4 MB static external memory, system signals interface, and 4 serial link interfaces. The system signals, (Reset, Analysis, NotError), are generated from the motherboard and propagate (ssReset, ssAnalysis, ssNotError) through the TRAM network. The main use of the system signals are to initialize the transputer at boot-up time, and to check for error status while the transputer is in operation [16].

The TRAM motherboard used for the experiment conforms to a standard PC AT bus architecture. The board can house up to 10 standard (size 1) transputer modules. The interface between the board and the PC is controlled by a 16 bit transputer microprocessor (IMS T222) with serial link interface adapter (IMS C012). The serial link interface adapter is by default connected to link 0 of TRAM on slot 0 on the transputer side, and interfaces to the host PC via set of registers conforming to the AT bus standard.

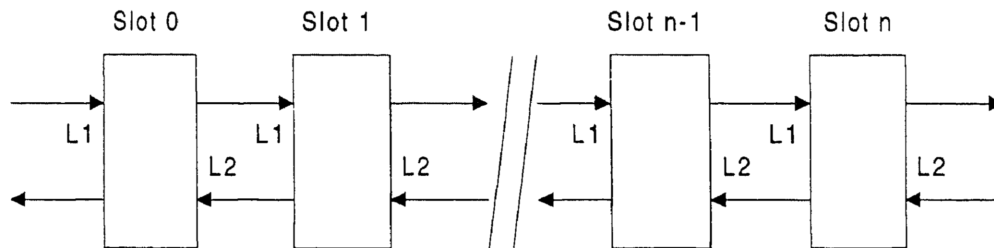

Figure 3.2. Default transputer module configuration on the motherboard [16]

Each TRAM has 4 link interfaces. Links 1 and 2 are hardwired to links 1 and 2 of the adjacent TRAMs on the motherboard resulting in a pipelined connection between the transputers when all the available slots are populated with a size 1 TRAM (see Figure 3.2). The remaining links 0 and 3 are connected to a programmable cross-bar switches

(IMS C004) controlled by the T222 microprocessor. Thus Links 0 and 3 can be softwired to links 0 and 3 of any other TRAMs (except the root TRAM) on board enabling the user to configure overall transputer network. All communication between the TRAMs are serial in nature. The speed of the communication can be set at either 10Mb/s or 20Mb/s by using jumpers on the motherboard [16].

The interface between the RTP system and the controller requires an analog to digital converter module (ADT) for the purpose of reading the temperature measurements and a digital to analog converter module (DAT) to output control signals to the lamp ring power supplies. The ADT used is a size 4 TRAM consisting of a 12 bit analog to digital converter and other electronics necessary in order to ensure proper communication with other transputer based TRAM.[17] The DAT is an eight channel, 12 bit, size 2 TRAM [18].

3.2 Programming for Transputer System

Programming for the transputer system is a two step procedure. First the network configuration file must be defined. The configuration file has an extension of *.cfs* and must be included in the make file or explicitly named during the command line compilation of the program modules. It must be noted that, prior to this step the hardware configuration utility (NCS) must be run in order to program the IMS-C004 cross bar switches with the desired configuration of the link connections [17]. The configuration file is used to map software network configuration to the hardware TRAM network configuration. The software network consists of processes which uses *channels* to communicate with other processes. The processes may be located within the same

TRAM, or in an adjacent TRAM. Given that each physical link may be used to define multiple logical channels, the configuration file defines on which physical processor (TRAM) the process is to be located, and what the channel interface of the process is with respect to its inputs and outputs.

The second step in programming for the transputer system involves writing of the program modules to be run on the transputer system. Because the program will be distributed over a transputer network consisting of multiple transputers, modularity of the program becomes an important issue in the software design. The idea of declaring global variables and allowing access to the variables among the program modules must be discarded. Also, since each transputer module has a separate memory space, the passing of pointer variables among program modules is not possible. The program modules instead must communicate with each other by passing copies of parameters in the form of messages. An example of a basic transputer program is shown below.

```
/* master process */
#include <stdio.h>
#include <misc.h>
#include <channel.h>

int main(int argc, char* argv[])
{
    /* declare channels */
    Channel *from_W1;
    Channel *to_W1;
    /* local variables */
    int i, val;

    /* initialize channels */
    from_W1 = (Channel*) get_param(3);
    to_W1 = (Channel*) get_param(4);

    val = 0;
    for (i = 0; i < 10; ++i)
    {
         /*output integer parameter to worker process */
         ChanOutInt (to_W1, i);
         /* read integer parameter from worker process */
         val = ChanInInt (from_W1);
    }
    /* send terminate sign to worker process */
    ChanOutInt (to_W1, -1);

    /* terminate program */
    exit_terminate (EXIT_SUCCESS);
}/* end of program master,c */
```

```
/* worker1.c */
#include <stdio.h>
#include <misc.h>
#include <channel.h>

int main(int argc, char* argv[])
{
    /* declare channels */
    Channel* from_master;
    Channel* to_master;
    int val;

    /* initialize channels */
    from_master = (Channel*) get_param(1);
    to_master = (Channel*) get_param(2);

    /* receive input from master */
    val = ChanInInt (from_master);

    /* while input is not terminate sign ... */
    while (val >= 0)
    {
         /* do something complicated ... */
         if (val == 0) val = 1;
         val = val*val;
         /* output to master */
         ChanOutInt(to_master, val);
         /* receive new input from master */
         val = ChanInInt(from_master);
    }
    /* terminate program */
    exit_terminate (EXIT_SUCCESS);
}/* end of program worker1.c */
```

The program modules *master* and *worker1* could be running on a same transputer or two separate transputers. In the latter case, the physical link connections over which the channels are declared must be specified in the network configuration file. The program declares *channel* variables and assigns to it the interface specification as defined in the network configuration file. Assuming that the configuration file had declared the following process interfaces;

```
process (interface( input in, output out,
                    input from_p1, output to_p1)) master;
process (interface( input from_master, output to_master,
                    input from_p2, output to_p2)) w1;
process (interface( input from_p1, output to_p1,
                    input from_p3, output to_p3)) w2;
process (interface( input from_p2, output to_p2)) w3;
```

the *get_param( )* command assigns to the channel variables the input and output channels to and from the worker process *w1*. The *ChanOutInt* and *ChanInInt* functions are used to pass and receive integer parameters between the master and the worker process. The *exit_terminate* command tells the transputer system that the program has terminated. The most important thing to notice from this simple example is that the program is basically similar to any regular *C* program except for the ways in which the variables are exchanged between the program modules. As can be seen, the communication between the program modules are carried out with a set of communication primitives. The communication primitives are blocking calls, that is a call to *ChanIn* will wait forever until the corresponding *ChanOut* is called in the called process and vice versa. The blocking nature of the communication protocol makes the process synchronization a simple task.

3.3 Hardware Architecture of the Adaptive Controller

The block diagram of a transputer based implementation of the controller is shown in figure 3.3. As mentioned previously, the links 1 and 2 of the TRAMs are automatically connected to the links 1 and 2 of the adjacent TRAMs by the motherboard to form a pipelined configuration. Given that we have four size 1 TRAMs as well as a size 4 ADT and a size 2 DAT, one of the ways the configuration of figure 3.3 can be achieved is by inserting size 1 TRAMs at TRAM locations 0, 1, 3, and 4. The TRAM at location 0 would be the root transputer. The link 0 of the root transputer is hardwired to the serial bus interface adapter and thus to the host PC. In this configuration the link 2 of the TRAM 0 will be connected to the link 1 of the TRAM 1, and link 2 of the TRAM 3 will

be connected to the link 1 of the TRAM 4. The ADT is inserted into slot 6. Since the ADT is of size 4, it will also occupy slots 8, 7, and 9. The DAT is of size 2. Insertion of DAT into slot 5 will also occupy slot 2.

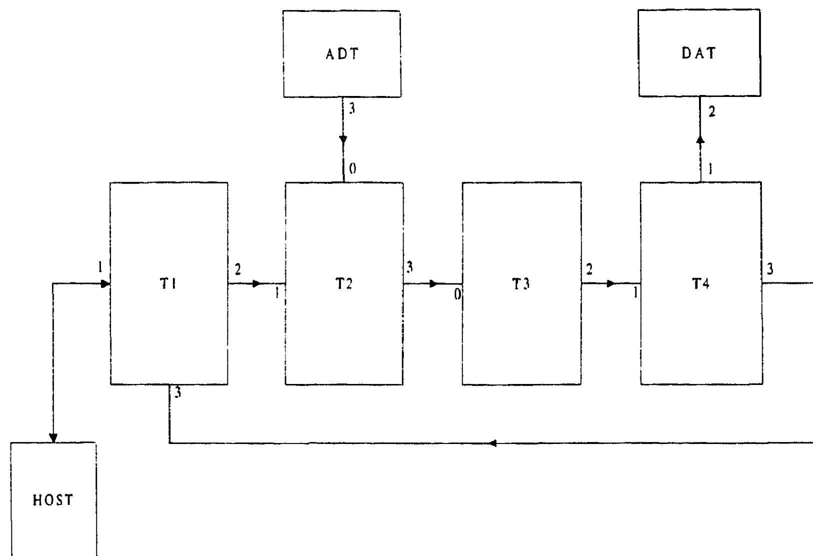

Figure 3.3. Block diagram of the transputer based implementation of the adaptive controller

The size of ADT and DAT effectively breaks the pipelined configuration. In order to obtain the configuration shown in figure 3.3, the link 0 of the T2 is connected to link 3 of ADT. Also link3 of T2 is connected to the link 0 of T3, and so on as shown in the figure.

The '*iserver*' utility supplied with the transputer motherboard allows interaction between the transputer and the operating system of the host PC. However, in order to enable communication between the program running on the root transputer and the program running on the host PC, a custom device driver is needed. The transputer motherboard used for the implementation of the controller conforms to the PC AT bus

interface standard. Therefore, the transputer can be thought of as a set of registers in the I/O space of the host accessible to the host program using standard DOS device driver commands *ioctl*.

The controller is timer interrupt driven. In each interrupt cycle, the system must read in the temperature readings from the thermocouples, generate control outputs, output control signals to the lamp ring power supplies, and update graphics on the user interface. The source of the timer interrupt is the host PC through the use of an IBM DACA board. The host PC itself is an IBM compatible with Intel 486 CPU and 8 Mb of memory.

3.4 Software Architecture of the Adaptive Controller

The software for the controller system consists of program modules running on the transputer and program modules running on the host PC. An interface program between the transputer and the host PC allows the exchange of the data between the two program modules. The transputer program consists of four modules each running on a separate transputer. The master program, running on the root transputer T1, is responsible for communicating with the host and also for initiating the sequence of actions to be performed by the worker processes. The worker process 1, running on transputer T2, interfaces with the ADT module and performs scaling of temperature readings as well as filtering of the temperature readings. The worker process 2, running on T3, is responsible for generating the reference temperature trajectory and interfacing with the control algorithm. The control algorithm is run as a sub-function entirely in transputer T3. The worker process 3, running on T4, is responsible for output of control signals to the DAT

module and for communicating with the root transputer. Scaling and bound checking of the control signals are also performed.

The sequence of actions performed during an interrupt cycle is as follows. The generation of interrupt timer signal causes the host PC to send a message packet to the master transputer. The message packet contains information regarding the status of the user input. Upon receiving the message packet, the master transputer forms another message packet and sends it to T2. The transputer message packet consists of pertinent control system data as well as the status of the user interface. Transputer T2 initiates ADT to read the temperature measurements from the thermocouples. The measurement readings are copied to the message packet and sent to T3. T3 strips off the temperature measurement data and calls the control algorithm sub-function. The control algorithm generates the control output which is then copied to the message packet. The message packet is subsequently sent to T3. T3 strips off the control output information and sends it to the lamp ring power supplies via DAT. The message packet is then sent back to the master transputer T1 where the format of the message is transferred to the host message data structure. The message then is sent to the host, completing a single control loop sequence. From this moment on until the generation of next timer interrupt, the host displays the information on the screen as a strip chart, and scans for any changes in the user input status.

The host program running on the PC consists of 3 modules and is implemented using C++. The modules defined in the host program are defined by the *Class* abstract data type which encapsulates variables and functions associated with the modules. Two main modules are the LabWindow user interface object and the transputer interface

object. The LabWindow is a DOS-based graphical user interface generator providing graphics capability as well as mouse driven user environment. Figure 3.4 shows a sample screen shot of the user interface while the experiment is in progress.

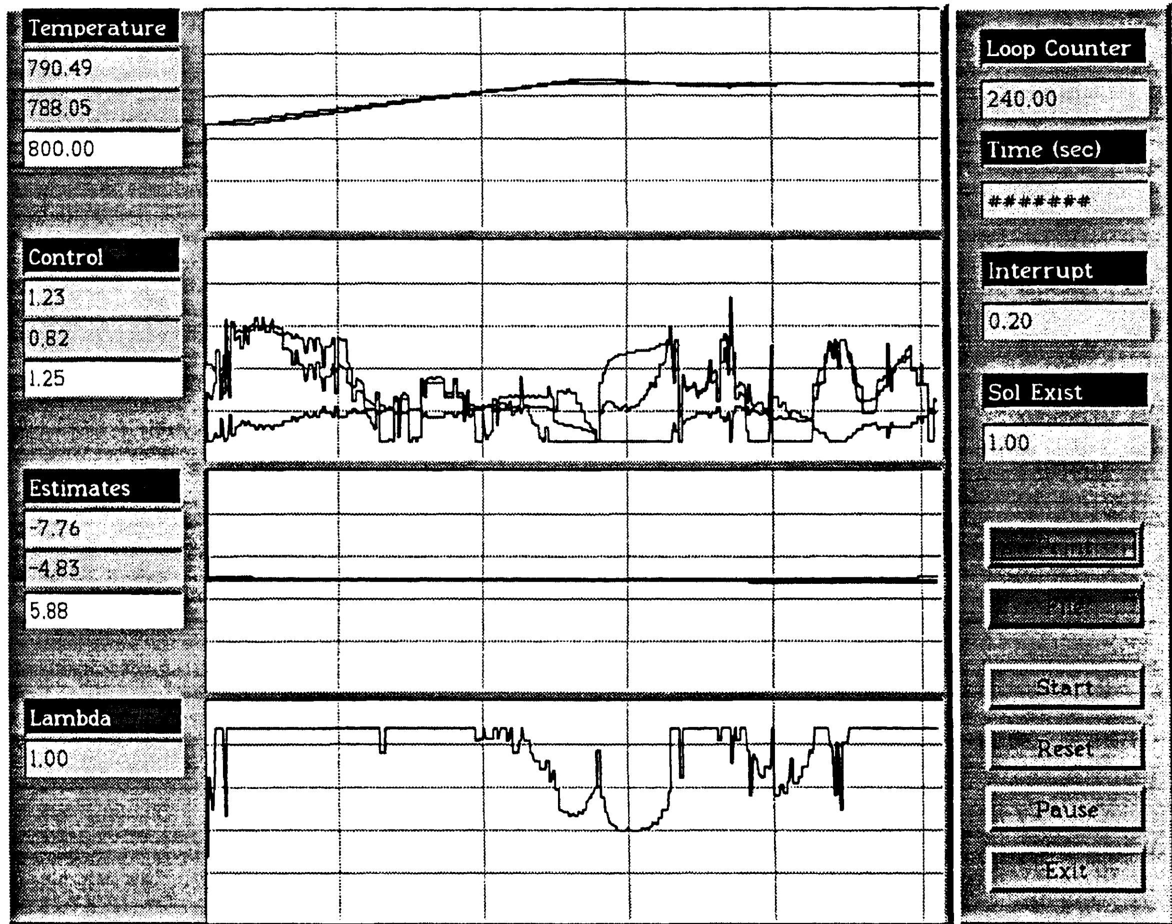

Figure 3.4. Graphical user interface for the adaptive controller

The LabWindow object receives as an input the message data structure from the transputer through the transputer interface object. The message structure contains an updated information of the control system such as temperature measurements and control

outputs. The main task of the LabWindow object, other than interfacing with the user, is to display these data on screen. The data is therefore updated at every interrupt cycle.

The communication between the LabWindow object and the Transputer object is by way of message passing. The intermediary between the LabWindow object and the transputer interface object is the main module. The main module acts as a main entry point of the program, and invokes interrupt service routine in response to the hardware timer interrupt.

The transputer interface object consists of functions which act on the transputer through a standard DOS device driver calls. In programming sense, the transputer motherboard can be seen as two 16 bit registers in the I/O space of the host PC [16]. These registers are mapped into the memory space of the T222 transputer on the transputer motherboard. This memory space is used as a buffer during the read and write operations between the transputer motherboard and the host PC. On the transputer side, the host interface software resides in a flash ROM and access to the host is accomplished via standard *ChanIn* and *ChanOut* commands. On the host side, the interface software utilizes standard DOS device driver through *ioctl* function calls [17]. The communication between the host and the transputer is by way of message passing. The message is defined by a data structure, thereby enabling communication of mixed data types. The size of the data structure is limited to *512* bytes in length. This is due to the limitation of the DOS device driver. For messages larger than *512* bytes, multiple *read* calls can be made.

The transputer interface object on the host is defined by the C++ abstract data type *Class*. The class defines a set of public and private methods thereby clearly defining its interface to other program modules. The private methods defines functions used in

initializing the transputer and to boot the transputer with the desired executable program. The public functions provide to the application modules the functions to read and write transputer as well as check for the error conditions (*NotError* signal on the TRAM).

Upon boot up, the transputer goes through self analysis mode in order to reset and check for errors on the transputer network. During this process, the transputer sends a series of bytes to the host PC. For successful boot up of the transputer, these series of bytes must be read correctly by the host PC. The message data structure defined for the transputer interface modules *read* operation defines as the first element an integer number in order to facilitate the reading of these bytes. The rest of data elements are identical to the message data structure defined in the transputer programs. For transputer *write* operation, the first byte of the message structure also must be the byte count of the message structure. The complete listing of the program is included in the appendix for reference.

3.5 Simulation System

The purpose of the simulation system is two fold. First, the simulation system is used to verify correct operation of the adaptive control algorithm and the controller system. Second, we wanted the simulation system to be as close to the actual experimental system in terms of the hardware and software interfaces necessary so that the transition from the simulation system to the experimental system would be accomplished with least amount of complication. The implementation method of the simulation system provided added benefit in that the system automatically took into account the sampling and quantization

noises into the simulation. A block diagram of the simulation system is shown in Figure 3.5.

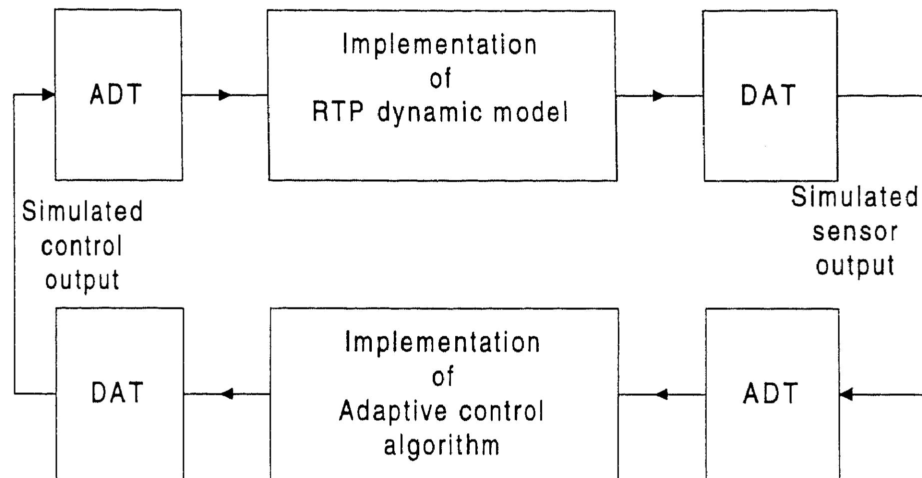

Figure 3.5. Block diagram of the simulation system

For the implementation of the simulation system two PCs, each hosting a transputer motherboard, are utilized. On one of the PCs, the controller is implemented as has been described in the previous sections. The dynamics of the RTP model is implemented on the second PC to simulate the RTP station. The two PCs are connected by a 26 pin ribbon cable via ADT and DAT pair of each system. A simple communication protocol is established to synchronize the activity in both systems.

3.6 Experimental System

The adaptive controller from the simulation system, with minor changes, used for the experimental system. The controller is interfaced with the RTP station through the ADT and DAT modules on the transputer motherboard. The input to the controller is the

thermocouple readings. The thermocouples used for the experiments are a K-type thermocouples for which the operating temperature range is defined to be within 0 and 1000K. An Analogic® DCP5B47 thermocouple signal conditioner is used to acquire linear readings in the range of 0 and 5 volts. The signal conditioner is connected to an ADT module via a 26 pin ribbon cable. The controller output from the DAT is connected to the switch box which is connected to the lamp ring power supplies. The switch box is necessary in order to cutoff the output of the DAT during the boot time since the DAT output at this time is undefined. The switch box also serve as a kill switch in case of an emergency situation. The schematic of an experimental system is shown in Figure 3.6.

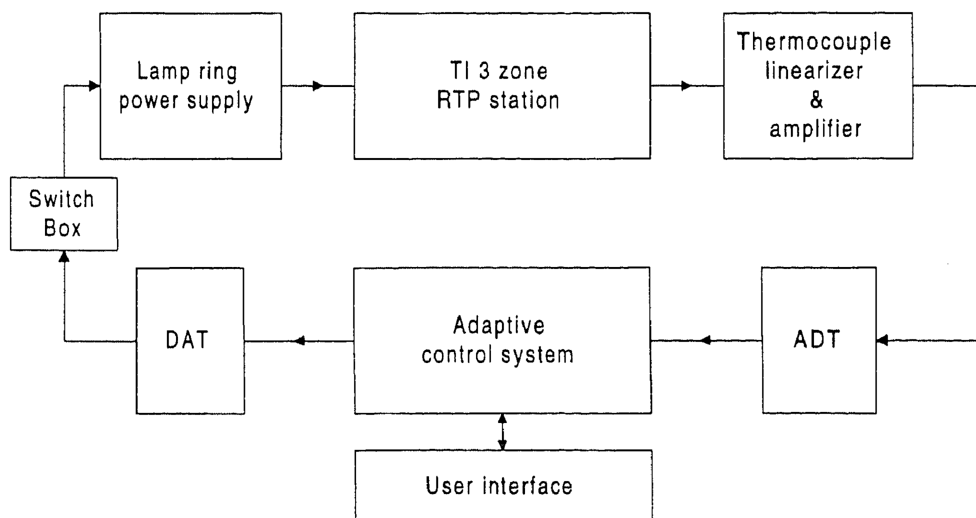

Figure 3.6. Block diagram of the experimental RTP system

CHAPTER 4

SIMULATION AND EXPERIMENTAL RESULTS

4.1 Preliminary Simulation Results

The general control strategy in controlling the temperature tracking as well as uniformity of the temperature along the surface of the wafer is to independently control the amount of heat radiation by three lamp rings based on the measurement of the temperature at three points along the diameter of the wafer. The control algorithm assumes a model of the system. Any unknown, or unmodeled parameters of the system are estimated using the parameter estimation algorithm.

The purpose of first simulation run was to evaluate the performance of the parameter estimator as well as to evaluate the correct operation of the control system. The limiting of output voltages to within 0 and 5 volts, reliable operation of the user interface, the data logging capability, the reliable communication between the transputer motherboard and the host, possibility of any deadlock situations in communication among the processes, correct operation of ADT and DAT modules are all carefully monitored and evaluated to ensure proper operation of the control system.

For evaluating the performance of the parameter estimator, the three estimator gain factors are set identically to 2.33e6. In previous single computer simulations, these value of gains have resulted in fast convergence of the estimates to the true values as shown in Figure 4.1. The simulation results indicate sensitivity of the parameter estimator to the presence of sampling noise in the temperature measurement as shown in Figure 4.2.

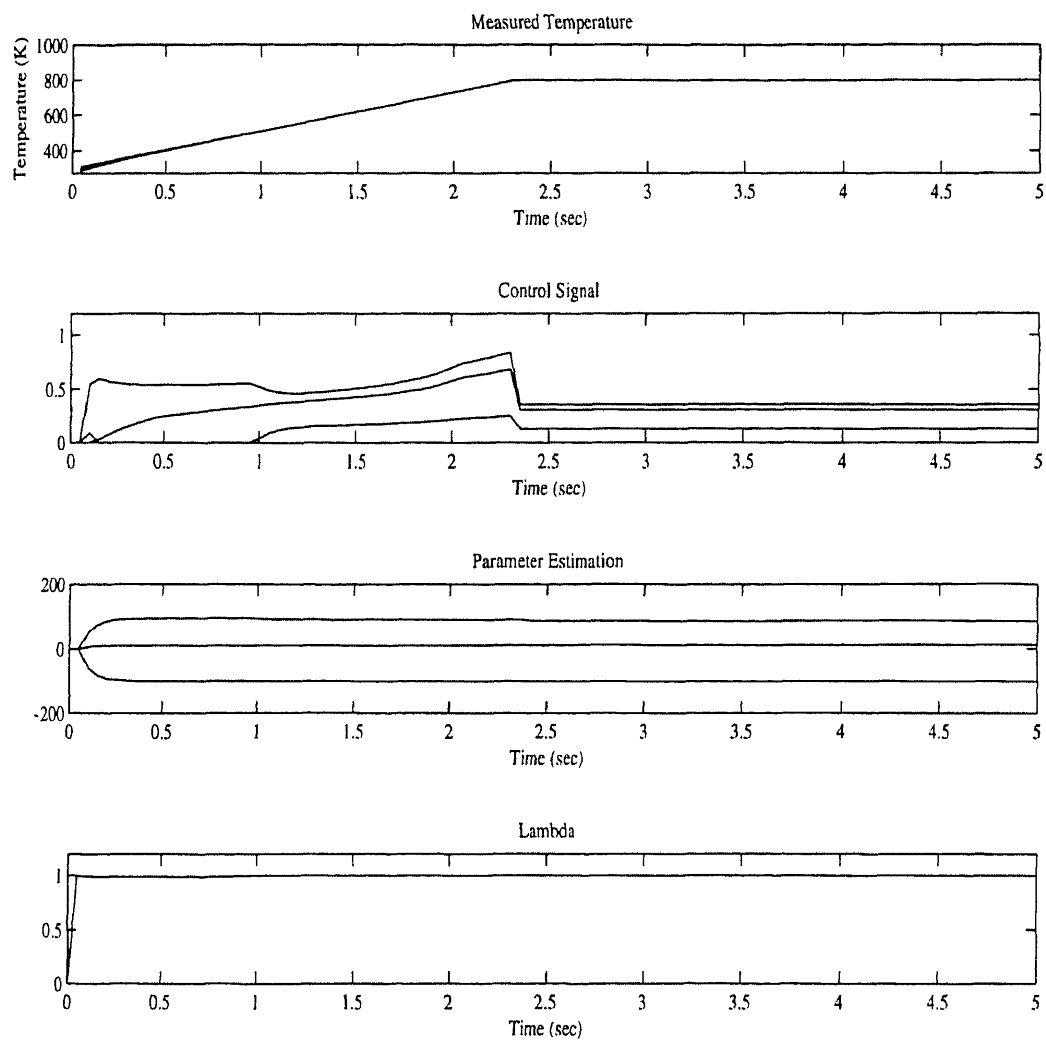

Figure 4.1. Single computer simulation result.

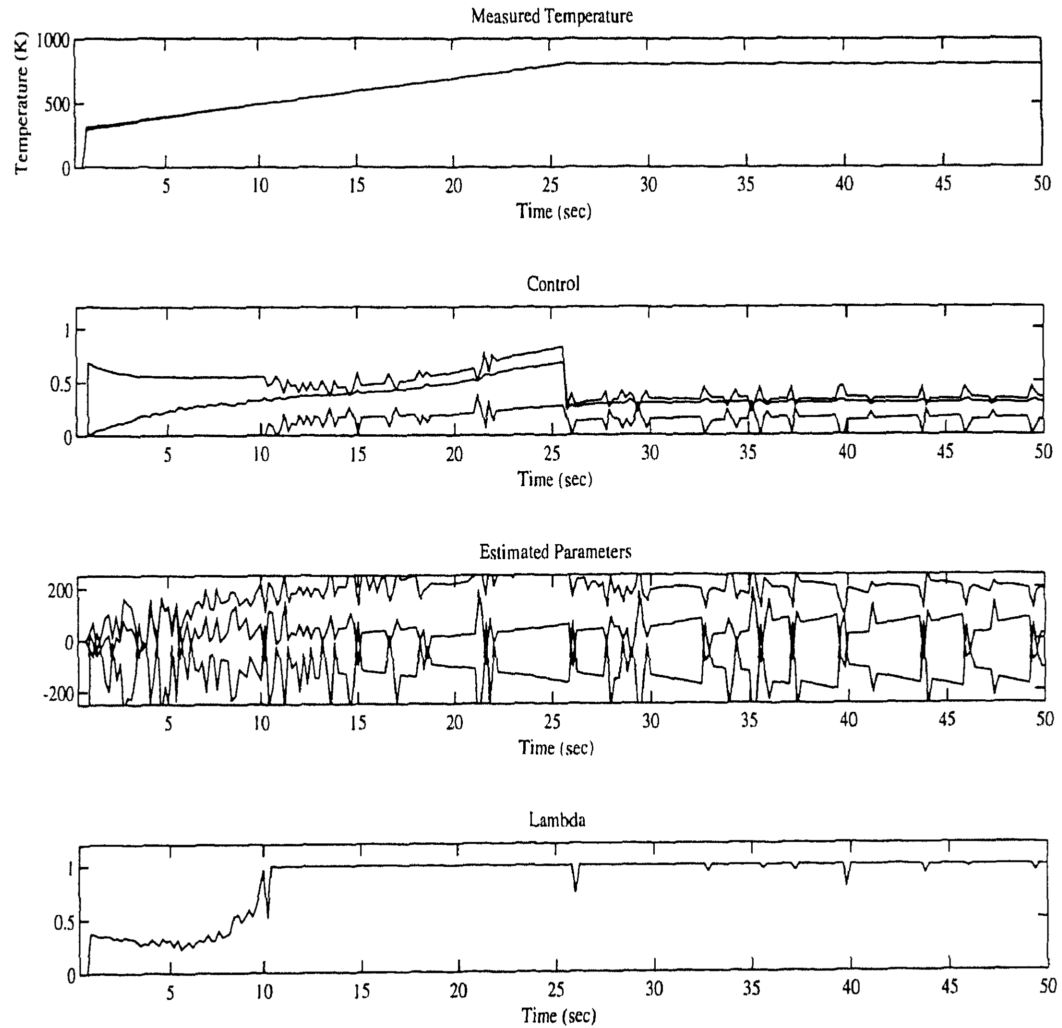

Figure 4.2. Simulation with parameter estimator gains set identically to 2.33e6.

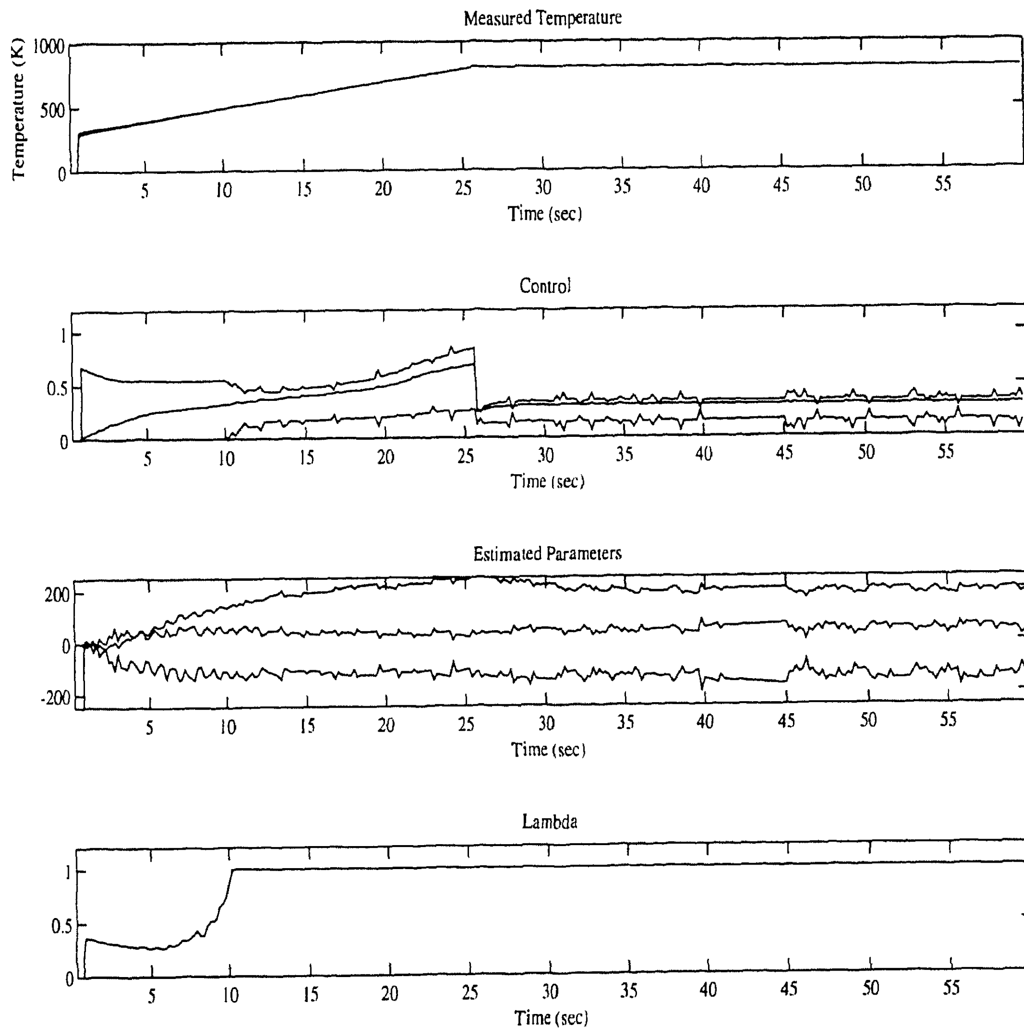

Figure 4.3. Simulation result with filtered temperature measurements.

Given that the temperature measurements range from around 300K to 1000K, and that the measurements are sampled using a 12 bit analog to digital converter, the maximum accuracy achievable in temperature measurement, in the absence of any other noise, is 0.17K. Numerous simulation results have shown the maximum tolerable sampling error at near 0.1K. One way to alleviate the effect of the sampling noise is to reduce the gain of the estimator at a cost of slower convergence rate. However, filtering of the measured temperature using some averaging technique in conjunction with moderate lowering of the estimator gain results in acceptable parameter estimation as shown in Figure 4.3. The result indicates convergence of estimated parameters to a constant values, however with steady state errors.

4.2 Preliminary Experimental Results

After verifying that the control system was running properly, the controller was attached to the actual RTP station. One of the problems encountered in implementing the experimental system was that of the three necessary thermocouple measurements, only two were deemed reliable. Since the nonlinear state observer requires three temperature readings, the third reading was "manufactured" as an average of the two reliable readings. Initially, several runs of open-loop control experiments were run in order to evaluate correct input and output operation of the experimental system.

Having verified the correct operation of the control system, the control loop was closed and the apparatus was operated. Initially the control system was run with temperature trajectory ramp rate of 20K/s and three estimator gains identically set to

1.33e3. The results indicate, as shown in figure 4.4, lagging effect, unaccounted for during the simulation runs. The primary source of the lag is seen to be associated with the lamp rings. The simulations assumed, rather unrealistically, that the lamp rings could be turned on and off instantaneously. There is a lag however, between the control signal and the lamp power output. The result of the lag manifests itself as the lagging of the controlled temperature with respect to the reference trajectory as well as significant amount of overshoot.

Figure 4.5 shows plots comparing the measured temperatures against the reference trajectory (*T0-Tref, T1-Tref*) as well as comparison of two measured temperatures against one another (*T0-T1*). The plots show that during the ramp-up period and during the transient period the measured temperatures deviate from the reference trajectory by up to 100K. But as the transients begin to diminish, the measured temperature converges to the reference trajectory. The uniformity of the measured temperature is also quite poor during the period when there is an overshoot. The difference in the measured temperature, thus the uniformity of the temperature, however is also seen to converge to near zero as the transient dies out.

Reducing the trajectory ramp rate to 10K/s, while keeping estimator gains at 1.33e3, results in better tracking performance as shown in Figures 4.6 and 4.7. The plots indicate significantly less overshoot and better transient response than in Figures 4.4 and 4.5. The tracking of the reference trajectory, shown in Figure 4.7, during the ramp up period appears to be also quite improved. The uniformity of the measured temperatures also appears to be quite reasonable. In all cases, the estimated parameters also converge to some steady state values.

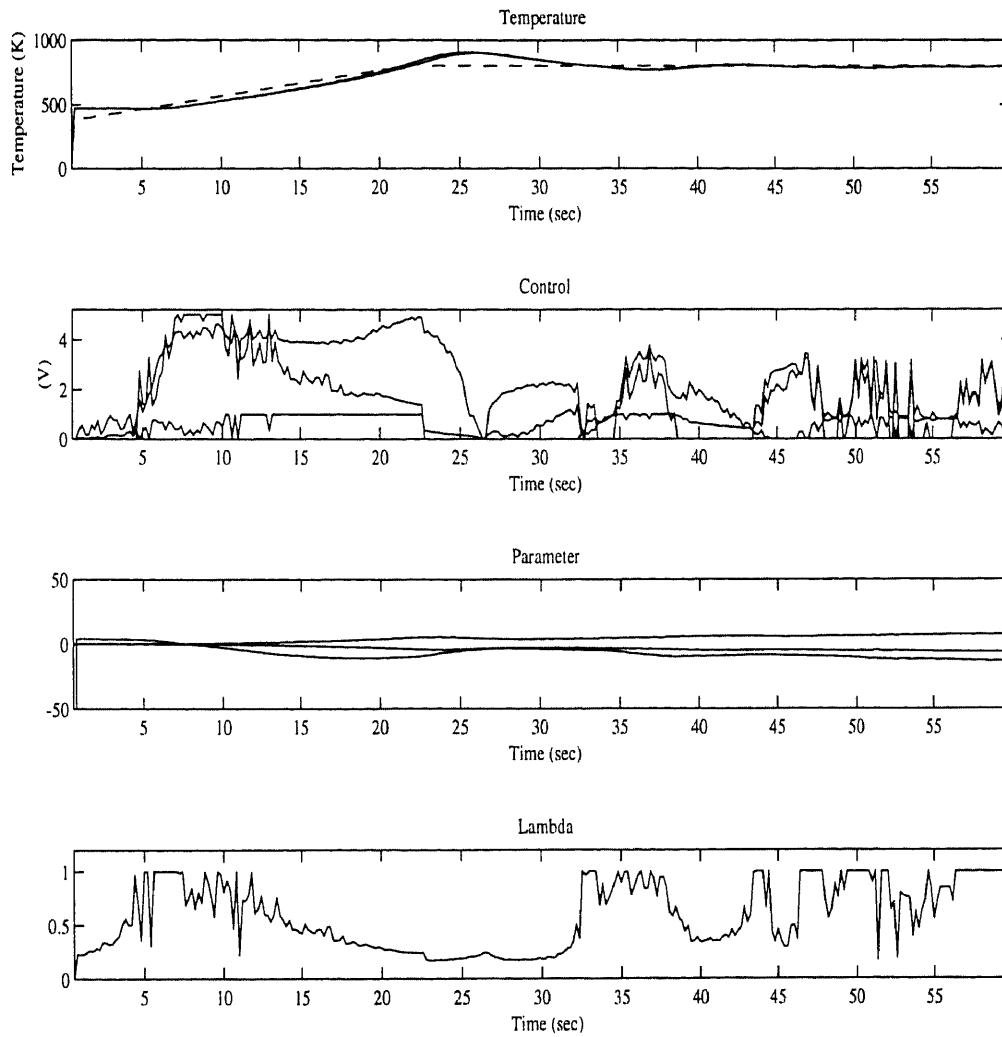

Figure 4.4. Experimental results. Ramp rate is 20K/s, parameter estimator gains are set identically to 1.33e3.

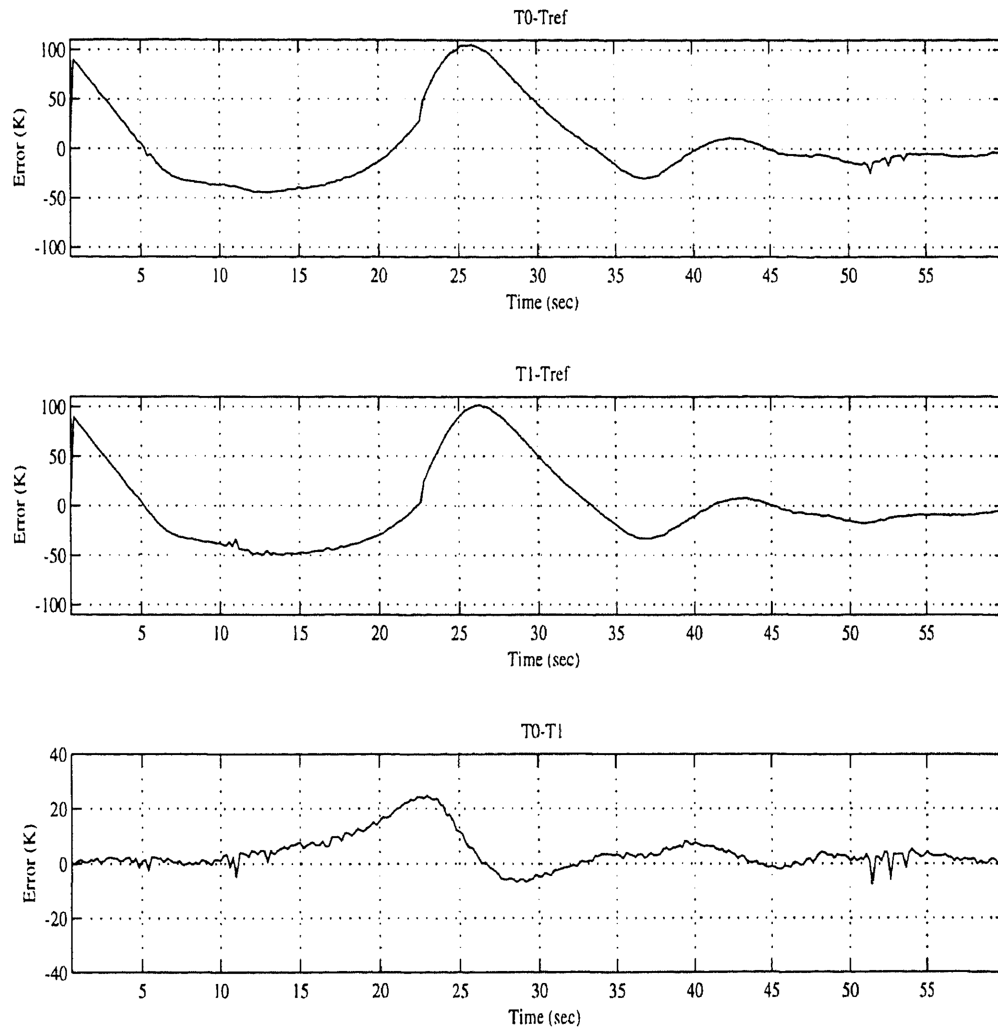

Figure 4.5. Comparison of measured temperatures vs. Reference trajectory. The ramp rate is 20K/s, parameter estimator gains are set to 1.33e3.

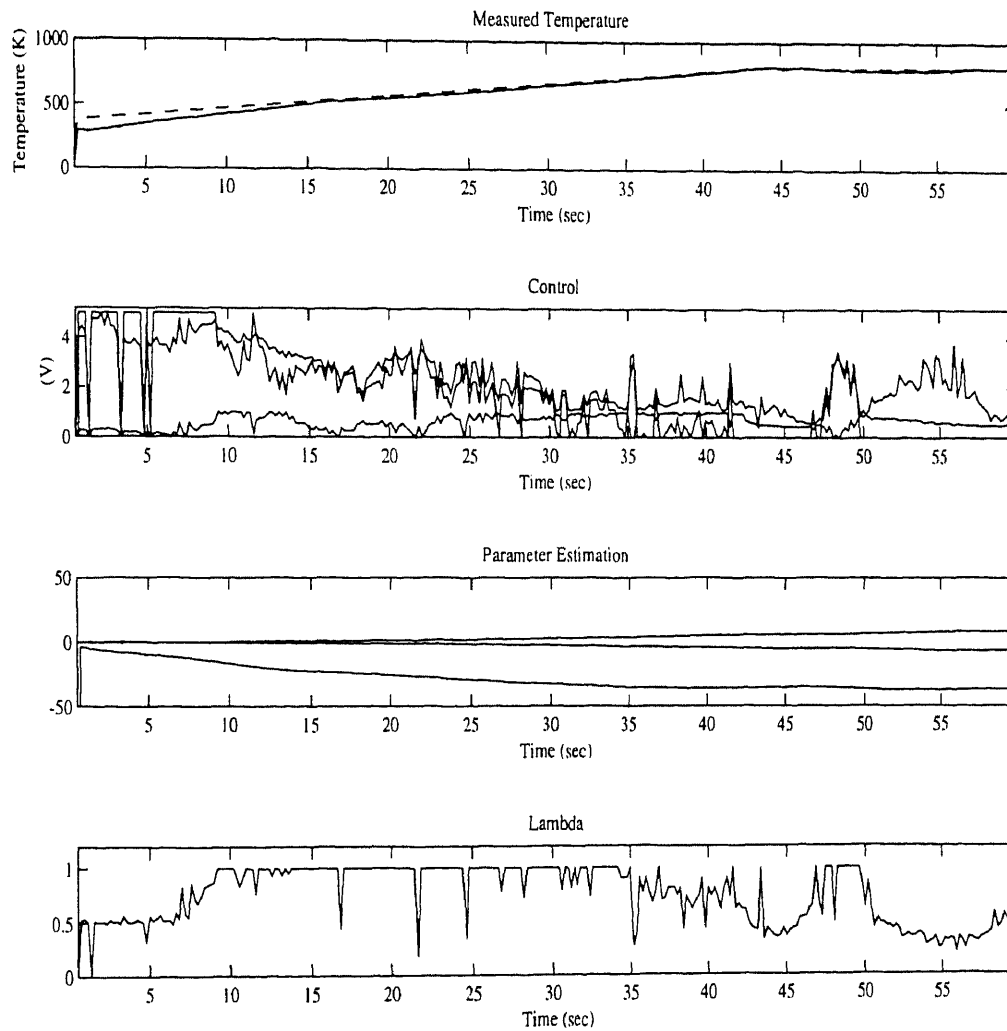

Figure 4.6. Experimental result. Ramp rate is 10K/s, parameter estimator gains set identically to 1.33e3.

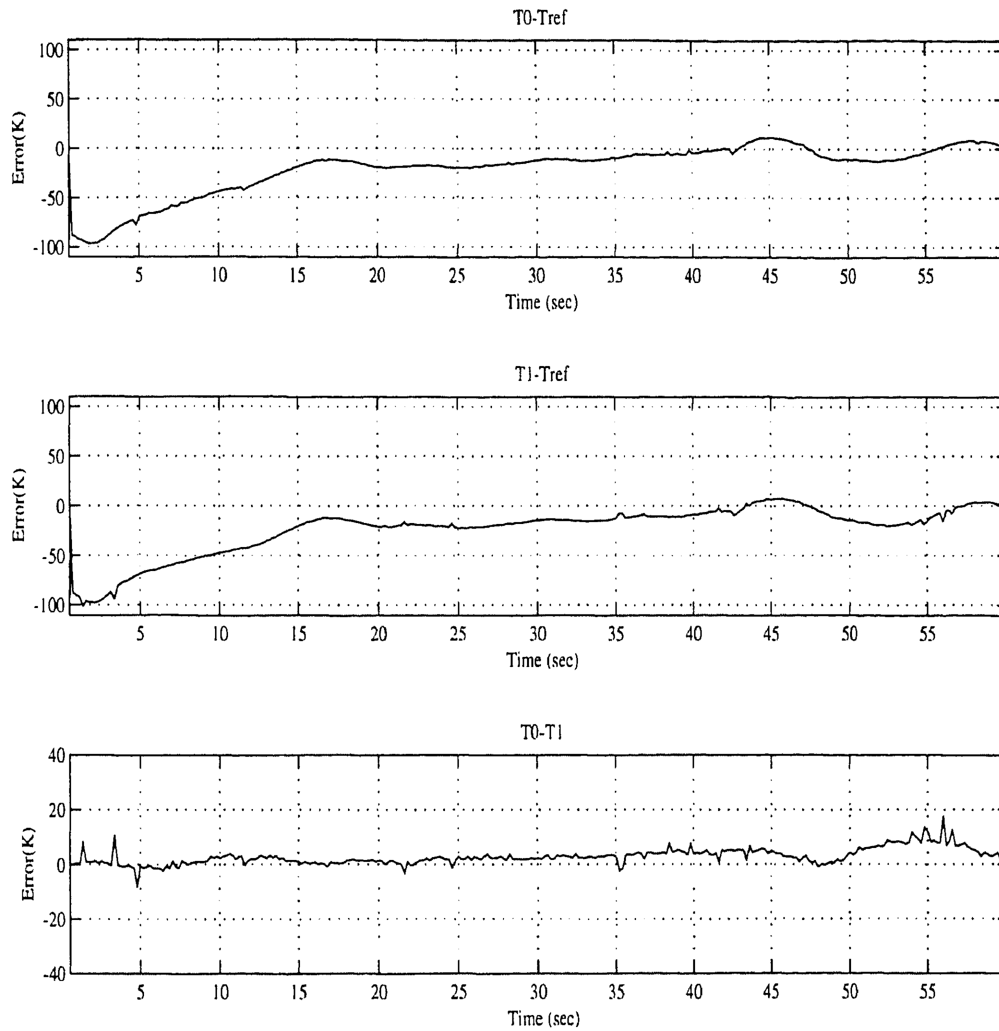

Figure 4.7. Comparison of measured temperature against reference trajectory. Ramp rate is 10K/s, parameter estimator gains are set to 1.33e3.

CHAPTER 5

STATE ESTIMATION USING EXTENDED KALMAN FILTER

5.1 Motivation

The state estimator for the adaptive control algorithm solves the temperature distribution equation (2.6) in terms of the state variables given three temperature measurements. During the experiment, with only two of the measurements available, the third reading of the temperature was fabricated as the average of the two reliable temperature measurements. Under this circumstance, other means of estimating the state variables using fewer than three temperature measurements are desirable. Also, the results of the previous experiments have shown significant amount of noise present in the control signal. This noise is most likely due to the noisy measurement of the temperatures. The noise contributes to the limit cycle like oscillation during the steady state, which is very much undesirable. Filtering of the temperature measurement by the state estimator should reduce the noise content of the temperature measurements. In this chapter, the method of estimating the state variables using a full-order Extended Kalman Filter as the state estimator is explored. The simulation results and experimental results of implementing the Extended Kalman Filter as the state estimator for the adaptive control algorithm is presented.

5.2 Extended Kalman Filter

Given a nonlinear process described by;

$$\begin{aligned} \dot{x} &= a(x,u,t) + G(t)w \\ z &= h(x,t) + v \end{aligned} \tag{5.1}$$

where w and v are the white noise process, and initial conditions $x(0) \approx (\bar{x}_0, P_0)$, $w \approx (0,Q)$, and $v \approx (0,R)$, assuming that $E\begin{bmatrix} w & v \end{bmatrix} = 0$, the equations for the continuous time Extended Kalman Filter is described by [20];

$$\dot{\hat{x}} = a(\hat{x},u,t) + K[z - h(\hat{x},t)] \tag{5.2}$$

$$\dot{P} = AP + PA^T + GQG^T - PH^T R^{-1} HP \tag{5.3}$$

where;

$$A = \left.\frac{\partial a}{\partial x}\right|_{x=\hat{x}} \tag{5.4}$$

$$H = \left.\frac{\partial h}{\partial x}\right|_{x=\hat{x}} \tag{5.5}$$

and K is the Kalman gain;

$$K = PH^T R^{-1} \tag{5.6}$$

In this study the assumption is made that

$$\begin{aligned} G &= I \\ Q &= 25 \\ R &= 25 \end{aligned} \tag{5.7}$$

The system dynamics are given by the equations (2.8), (2.9), (2.10), and (2.11). These equations can be expressed for convenience as follows;

$$\dot{x}_0 = -\alpha(a_0 k(x_0) + b_0 E(x_0) x_0^3) x_0 + P_0 \tag{5.8}$$

$$\dot{x}_1 = -\alpha(a_1 k(x_0) + b_1 E(x_0) x_0^3) x_1 + P_1 \tag{5.9}$$

$$\dot{x}_2 = -\alpha(a_2 k(x_0) + b_2 E(x_0) x_0^3) x_2 + P_2 \tag{5.10}$$

where

$$\begin{aligned} \alpha &= \frac{1}{\rho c(x_0)} \\ a_0 &= 0 \\ a_1 &= (\mu_1 / R)^2 \\ a_2 &= (\mu_2 / R)^2 \\ b_0 &= 1 \\ b_1 &= b_2 = 4 \end{aligned} \qquad 5.11)$$

and

$$P_n = 2h^{-1}[\sum_{l=1}^{3} G_{ln}(U_l) + S_n(t)] \quad ,n = 1, 2, 3 \tag{5.12}$$

where S_n represents the unmodeled parameters of the heat transfer dynamics inside the RTP reaction chamber. The P_i of (5.8) through (5.12) is not related to the P of the variance equation (5.3).

From (2.6) the temperature at the center of the wafer $T(0,t)$ is;

$$T(0,t) = x_1 + x_2 + x_3 \tag{5.13}$$

where $x_1 = \overline{T} - x_0$ is the deviation of the measured temperature from the reference temperature, and x_2, x_3 are the coefficients of the Bessel function expansion. Also the

functions $E(x), k(x), c(x)$ are assumed to be constants with their average values $\overline{E}, \overline{k}, \overline{c}$ respectively. Therefore the observation $h(x,t)$ is;

$$h(x,t) = T(0,t) = x_1 + x_2 + x_3 \tag{5.14}$$

and;

$$A = \begin{bmatrix} a_1 & 0 & 0 \\ a_2 & a_3 & 0 \\ a_4 & 0 & a_5 \end{bmatrix} \tag{5.15}$$

where,

$$\begin{aligned} a_1 &= -\alpha(4\overline{E}x_0^3) \\ a_2 &= -12\alpha(4\overline{E}x_0^2)x_1 \\ a_3 &= -\alpha(A_1\overline{k} + 4\overline{E}x_0^3) \\ a_4 &= -12\alpha(4\overline{E}x_0^2)x_2 \\ a_5 &= -\alpha(A_2\overline{k} + 4\overline{E}x_0^3) \end{aligned}$$

and

$$H = \left.\frac{\partial h}{\partial x}\right|_{x=\hat{x}} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{5.16}$$

Therefore, solving (5.3) in terms of its elements:

$$\dot{P} = \begin{bmatrix} \dot{P}_1 & \dot{P}_2 & \dot{P}_3 \\ \dot{P}_2 & \dot{P}_4 & \dot{P}_5 \\ \dot{P}_3 & \dot{P}_5 & \dot{P}_6 \end{bmatrix} \tag{5.17}$$

where,

$$\dot{P}_1 = 2a_1P_1 + w - R^{-1}(P_1 + P_2 + P_3)^2$$
$$\dot{P}_2 = a_2P_1 + P_2(a_1 + a_3) - R^{-1}(P_1 + P_2 + P_3)(P_2 + P_4 + P_5)$$
$$\dot{P}_3 = a_4P_1 + P_3(a_1 + a_5) - R^{-1}(P_1 + P_2 + P_3)(P_3 + P_5 + P_6)$$
$$\dot{P}_4 = 2(a_2P_2 + a_3P_2) + w - R^{-1}(P_2 + P_4 + P_5)^2$$
$$\dot{P}_5 = a_2P_3 + a_4P_2 + P_5(a_3 + a_5) - R^{-1}(P_2 + P_4 + P_5)(P_3 + P_5 + P_6)$$
$$\dot{P}_6 = 2(a_4P_3 + a_5P_6) + w - R^{-1}(P_3 + P_5 + P_6)^2$$

The Kalman gain is:

$$K = R^{-1}\begin{bmatrix} P_1 + P_2 + P_3 \\ P_2 + P_4 + P_5 \\ P_3 + P_5 + P_6 \end{bmatrix} \tag{5.18}$$

Thus, the equation for the state estimator is:

$$\begin{aligned} \dot{\hat{x}}_1 &= -\alpha(A_0\bar{k} + b_0\bar{E}\hat{x}_1^3)\hat{x}_1 + R^{-1}K_1(T(0,t) - (\hat{x}_1 + \hat{x}_2 + \hat{x}_3)) \\ \dot{\hat{x}}_2 &= -\alpha(A_1\bar{k} + b_1\bar{E}\hat{x}_1^3)\hat{x}_2 + R^{-1}K_2(T(0,t) - (\hat{x}_1 + \hat{x}_2 + \hat{x}_3)) \\ \dot{\hat{x}}_3 &= -\alpha(A_2\bar{k} + b_2\bar{E}\hat{x}_1^3)\hat{x}_3 + R^{-1}K_3(T(0,t) - (\hat{x}_1 + \hat{x}_2 + \hat{x}_3)) \end{aligned} \tag{5.19}$$

5.3 Simulation Results

The simulation results of the adaptive control system using Extended Kalman Filter as the state estimator is shown in Figures 5.1, 5.2 and 5.3. For simulation purpose, the ramp speed is set to 10K/s and the parameter estimator gains are set identically at 1.33e3. Figure 5.1 shows that, in the simulated environment, the convergence of the estimated temperature, based on the estimated states, to the measured temperatures are quite good. Figure 5.2 verifies this by showing the temperature difference between the measured temperatures and the estimated temperatures. In order to check the uniformity of the

measured temperatures, the Figure 5.3 shows the difference between the three measured temperatures. The simulation results indicate that, using Extended Kalman Filter as the state estimator, in conjunction with the parameter estimator, and given that the system model is well known, a single sensor at the center of the wafer would be enough to generate good estimates of the state, and therefore good control of the temperatures along the surface of the wafer.

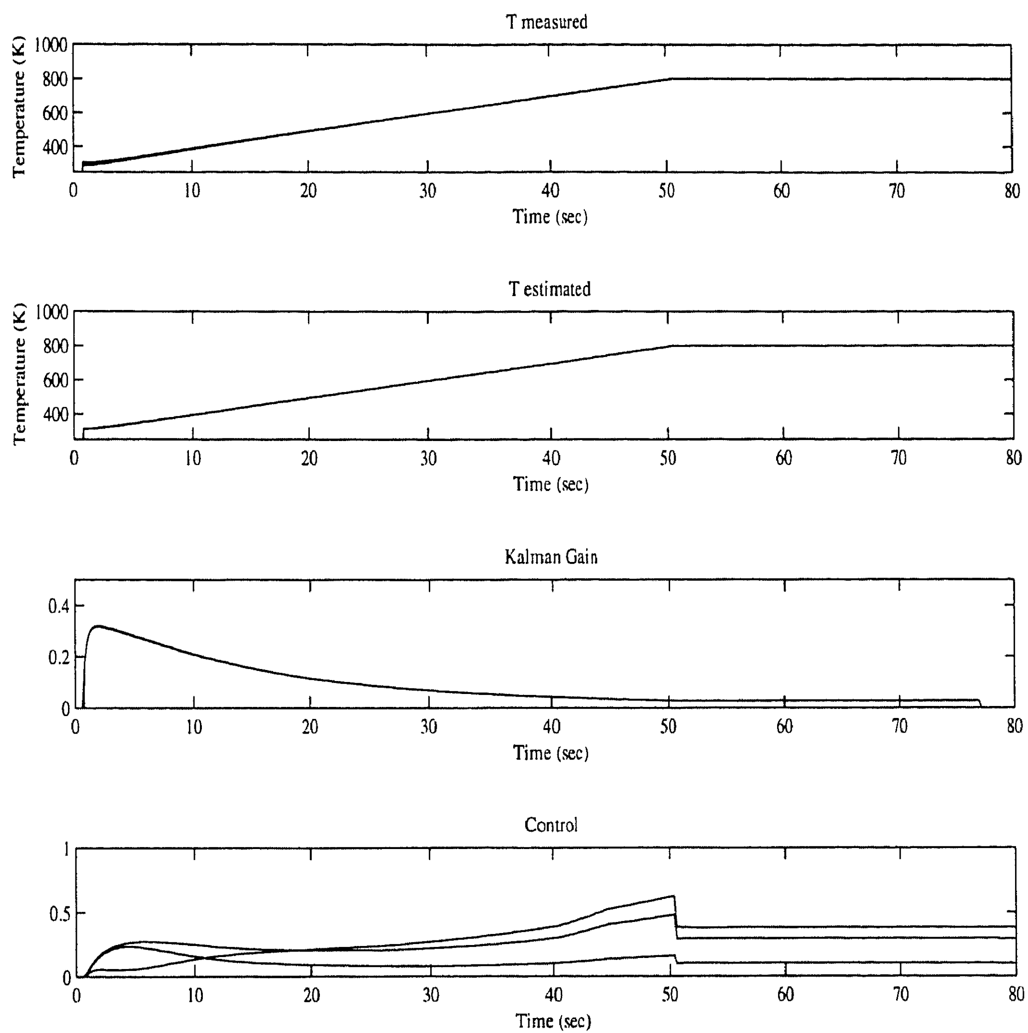

Figure 5.1. Simulation result using Extended Kalman Filter as the state estimator. Ramp rate is set at 10K/s, the parameter estimator gains are set identically to 1.33e3.

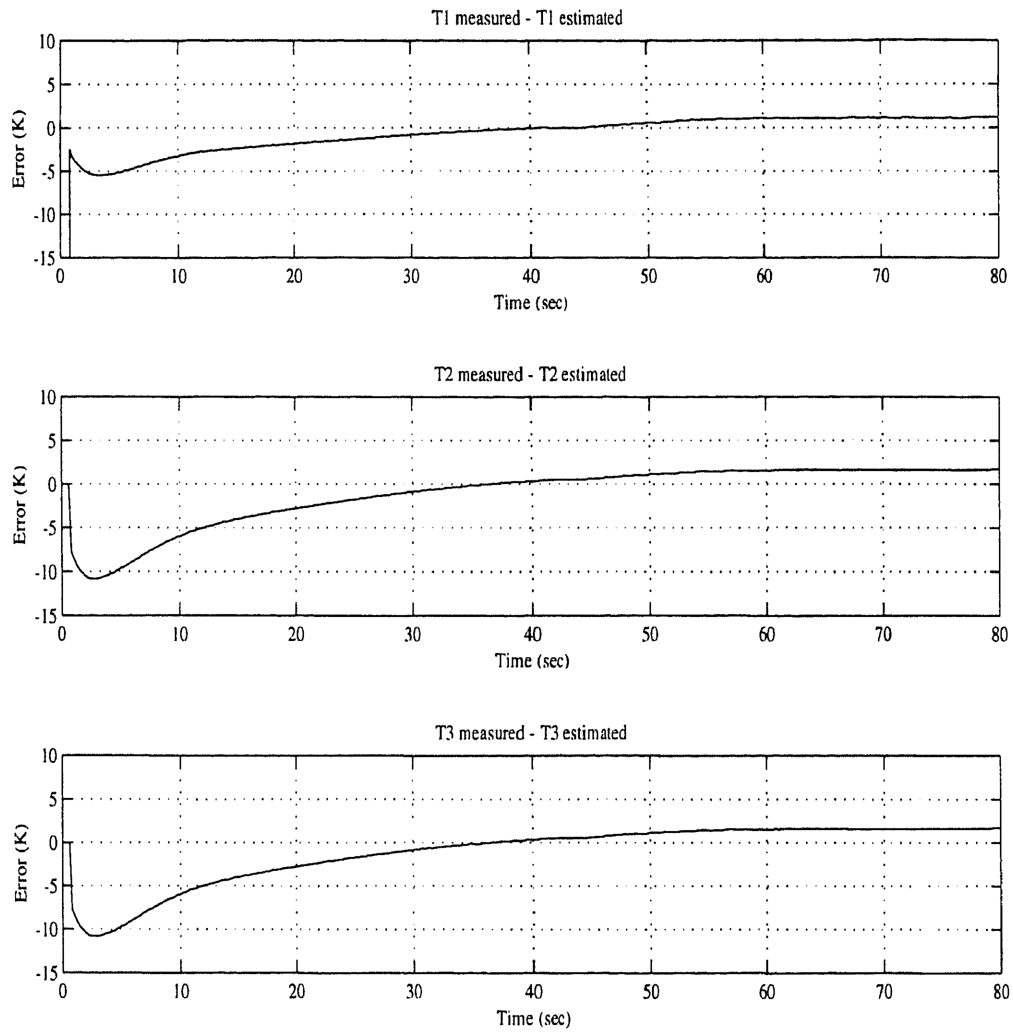

Figure 5.2. Simulation Result. Difference between the measured and estimated temperatures. The ramp rate is 10K/s, the parameter estimator gains are 1.33e3.

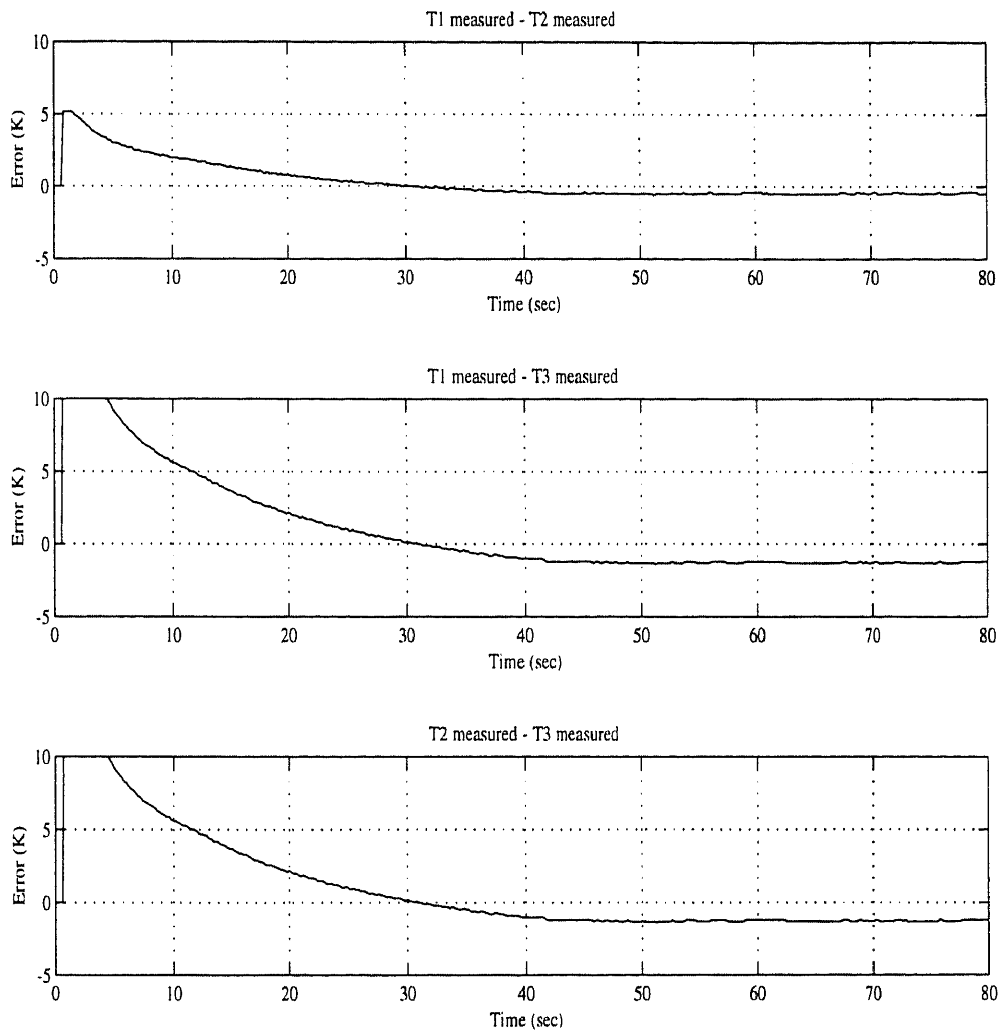

Figure 5.3. Simulation Result. Difference between the measured temperatures. The ramp rate is 10K/s, the parameter estimator gains are 1.33e3.

5.4 Experimental Results

Preliminary experimental results using Extended Kalman Filter as the state estimator is shown in Figure 5.4. Figure 5.5 shows the plot of the differences between the measured temperatures and the reference temperature, as well as the plot of uniformity between the measured temperatures. The ramp rate of the reference trajectory is 10K/s and the gain of the parameter estimator is set uniformly at 1.33e2. The performance of the system, in comparison with the previous results, indicate poorer steady state tracking of the reference trajectory. However, there is less overshoot and, more importantly, the transient behavior of the system using EKF as the state estimator is much more improved over that of the previous experiments. As can be seen from the Figure 5.4, the control signals generated reach a steady state as the temperature trajectories reach their steady state. Absent is the significant amount of noise present in the control signals that were present in the previous experiments. Also, the effect of the lamp hysteresis appears to be less severe than in the previous experiments.

Given that the system does indeed reach steady state, with constant tracking error, further experiments were performed by changing the gains of the parameter estimator. In Figure 5.6, the experimental result obtained with gains set uniformly at 2.33e4 is shown. Figure 5.7 shows the plot of reference tracking error and the plot of uniformity between the measured temperatures. The tracking error of T2 appears to be acceptable, however the tracking error of T1 is in the order of 15K. The temperature uniformity plot reflects this discrepancy between the T1 and T2. Also both T1 and T2 are greater than the reference temperature. Given the results, the experiment is performed with parameter

estimator gain set uniformly at 1.33e4. The results of the experiment are shown in Figures 5.8 and 5.9. Now the T2 is below the reference temperature, which is more preferable. Next, the gains of the parameter estimator are set to 1.0e4, 2.5e4, and 1.33e4. The experimental results using these values of gains are shown in Figures 5.10 and 5.11. The reference tracking error is seen to have reduced, and the uniformity appears also to have improved.

The experimental results so far obtained show improvements over the previous results without Kalman Filtering. The steady state and transient response of the system is much improved. Further optimization of the gain parameters and improvements in the accuracy of the system model should produce even better results. However, the results of this experiment must be tempered by the fact that the results cannot be verified yet. Without being able to physically measure the temperatures at various locations along the surface of the wafer, it is difficult to tell if the temperatures at locations far from the center of the wafer is indeed behaving in the manner similar to as would be expected from the system model being used. The availability of a wafer, with thermocouples imbedded at multiple locations, should allow us to verify the results. Also unclear at this point is the overall stability of the system incorporating both the state and the parameter estimator. Currently there are no theoretical results available from which the stability criteria can be derived.

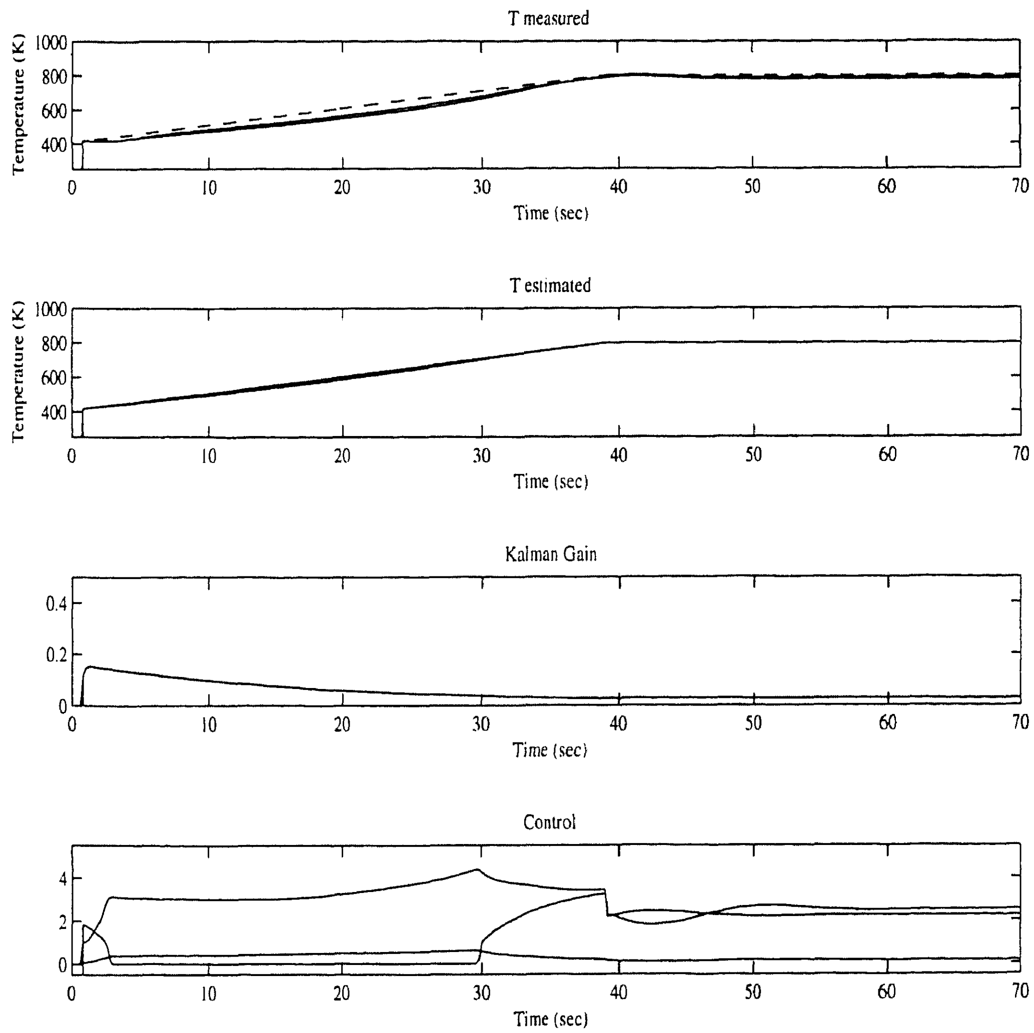

Figure 5.4. Preliminary experimental result of using EKF as the state estimator. The ramp rate is 10K/s, parameter estimator gains are 1.33e2.

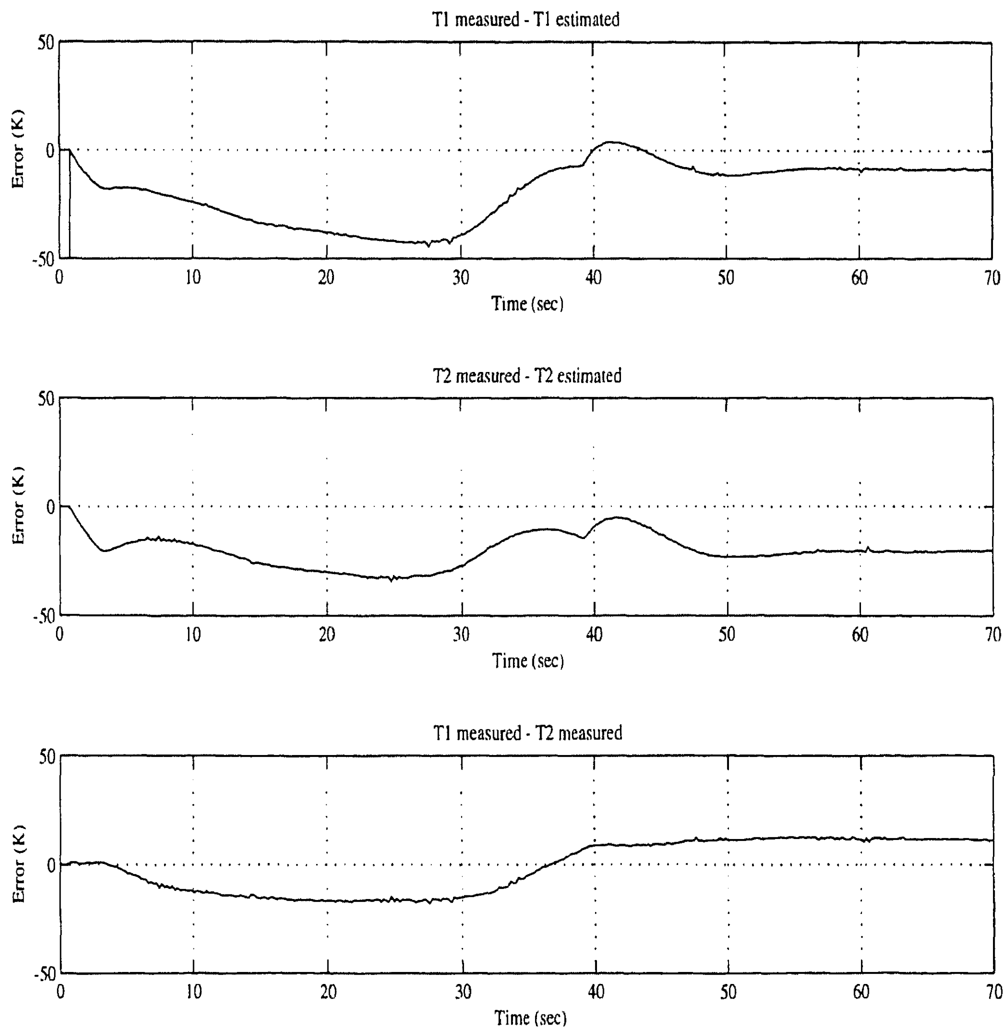

Figure 5.5. Plot of reference tracking error and uniformity. The ramp rate is 10K/s, parameter estimator gains are 1.33e2.

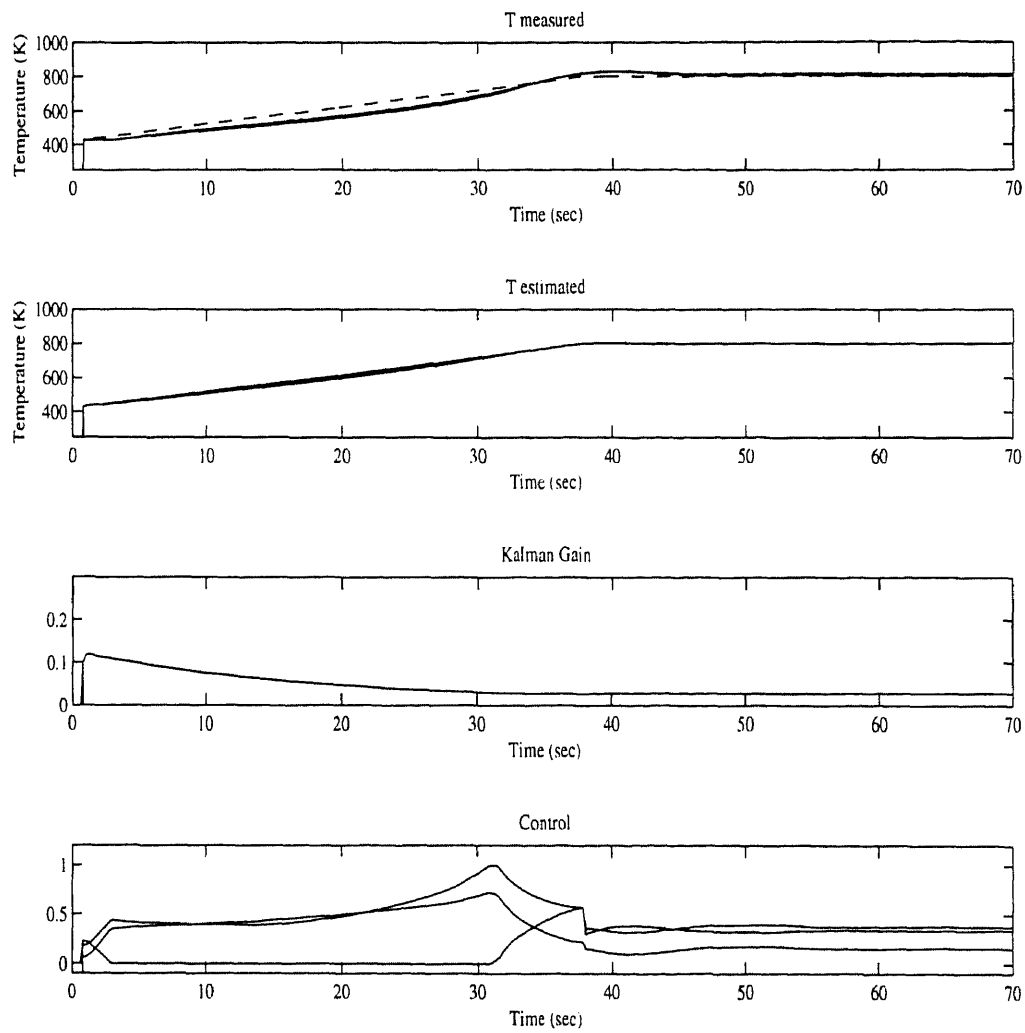

Figure 5.6. Experimental results with parameter estimator gains set at 2.33e4

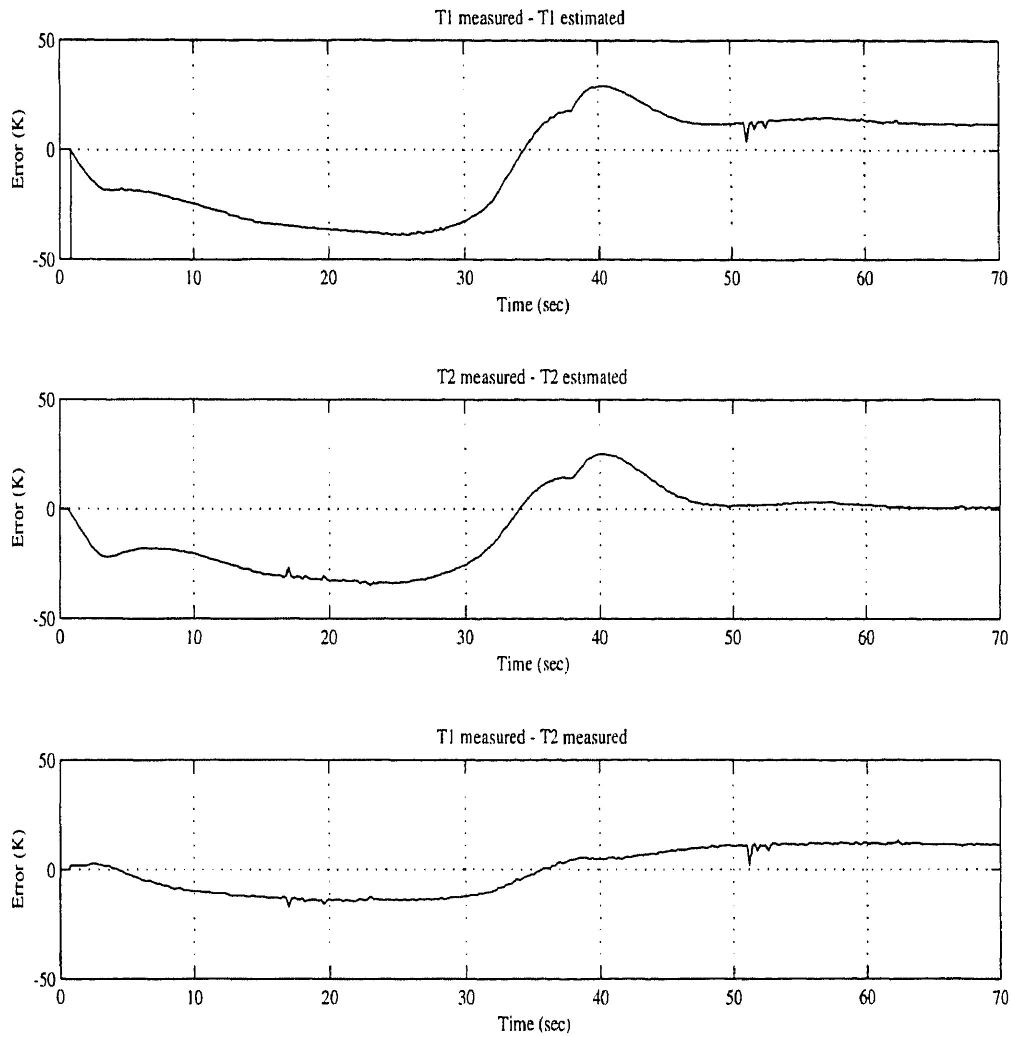

Figure 5.7. Plot of reference tracking error and uniformity with parameter estimator gains of 2.33e4.

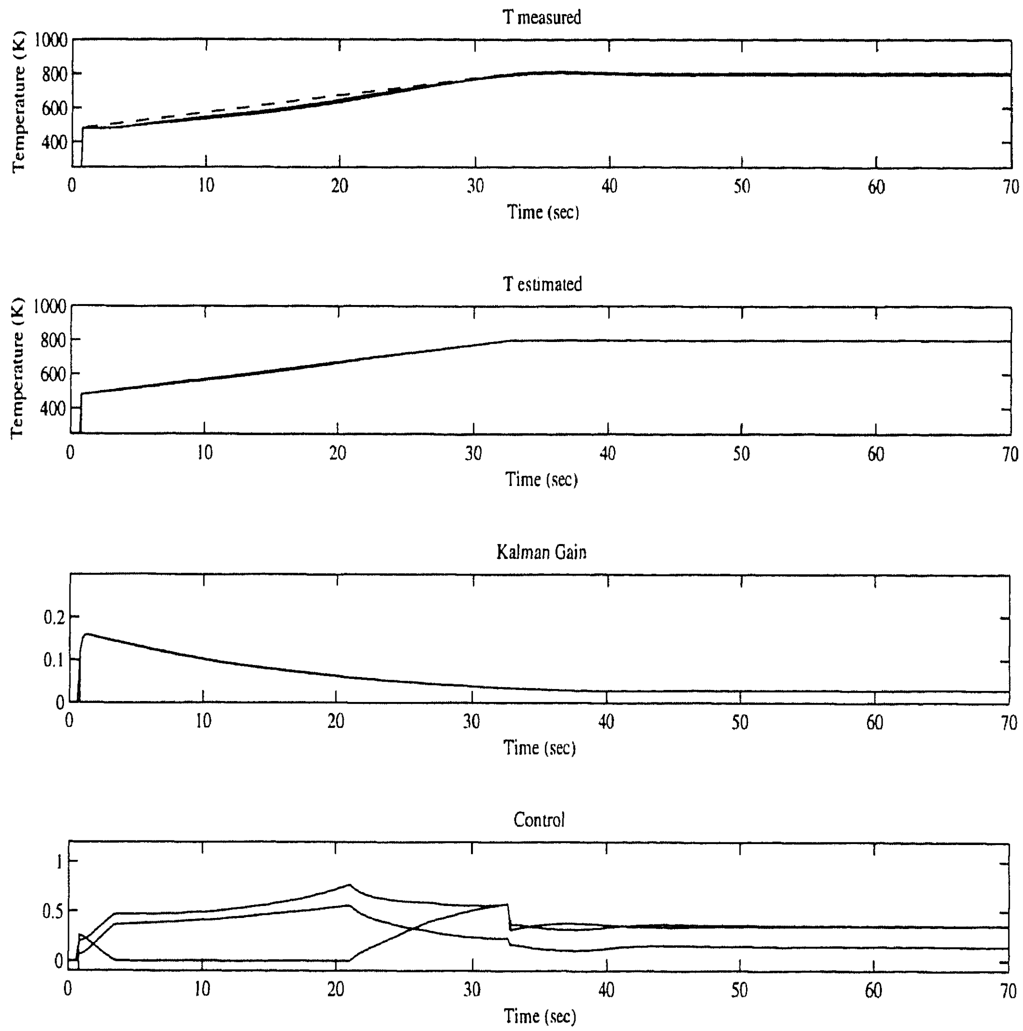

Figure 5.8. Experimental results with parameter estimator gains set at 1.33e4.

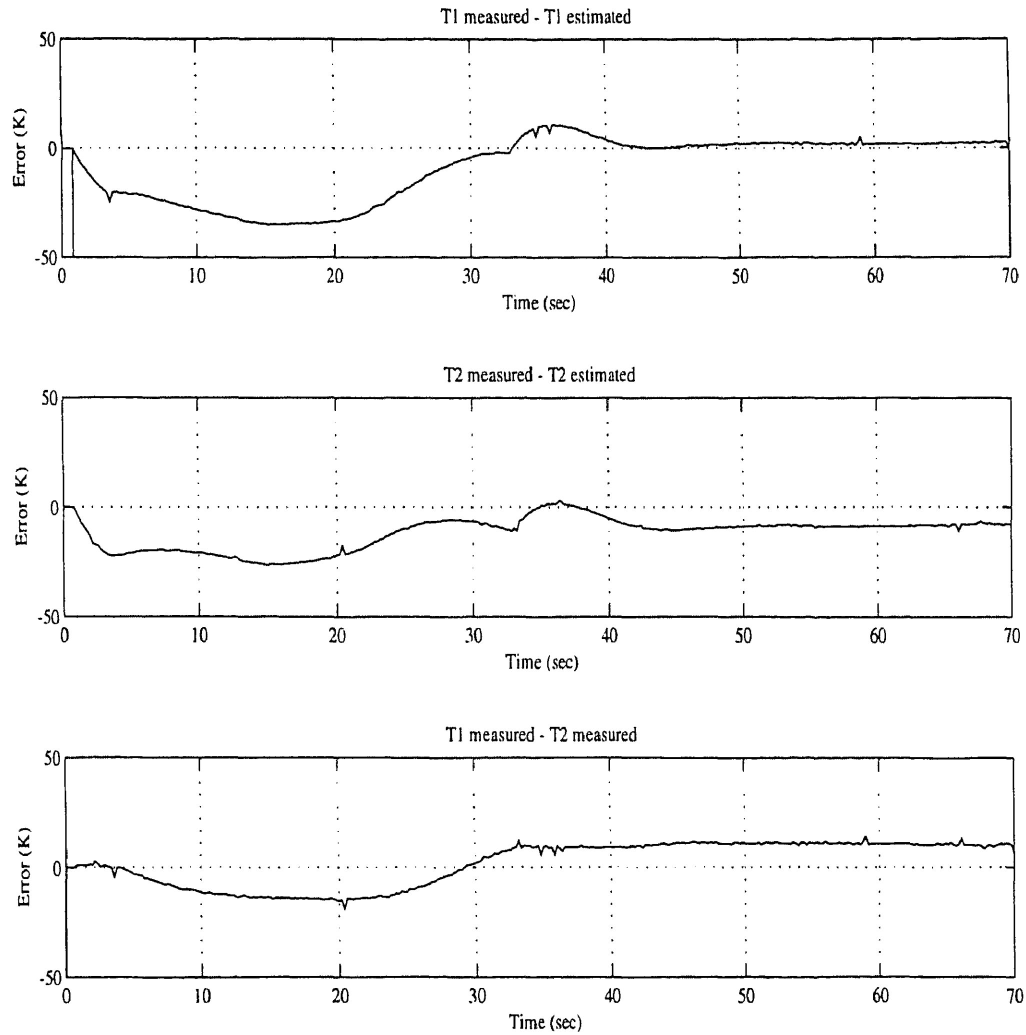

Figure 5.9. Plot of reference tracking error and uniformity with parameter estimator gains at 1.33e4.

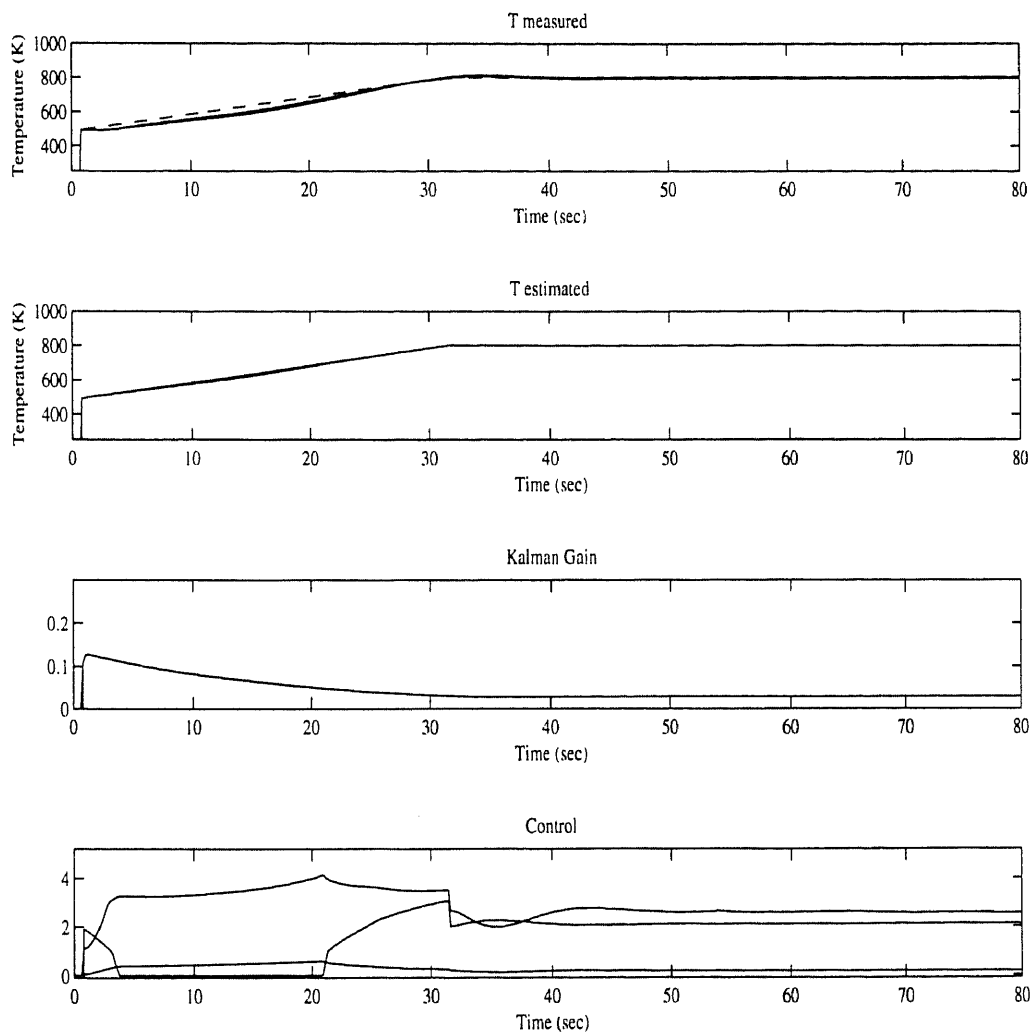

Figure 5.10. Experimental results with parameter estimator gains set at 1.0e4, 2.5e4, 1.33e4.

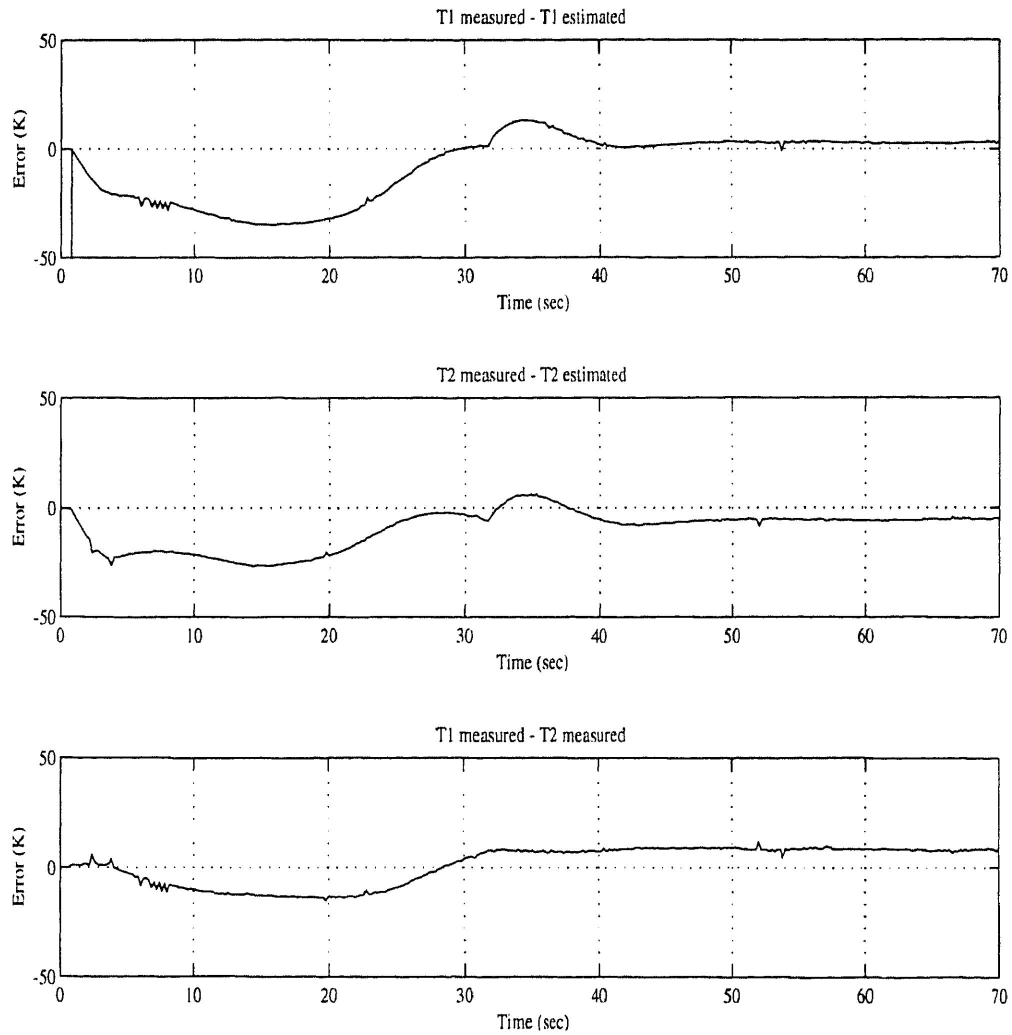

Figure 5.11. Plot of reference tracking error and uniformity with parameter estimator gains set at 1.0e4, 2.5e4, 1.33e4.

CHAPTER 6

ESTIMATION OF EMISSIVITY

6.1 Background

Emissivity is a quantity which relates the amount of power radiated by a heated body to the amount of power per unit area radiated by a black body at the same temperature [1]. As stated previously, in order to implement a practical RTP system, some method of remotely measuring the surface temperature of the wafer is needed. The most popular approach is to use pyrometry. The pyrometer measures the amount of radiative power generated at a point on the wafer over a narrow wave length [1]. In order to calculate the point temperature from the measured radiation, the emissivity parameter is needed. In this chapter the results of the experiments conducted with the experimental RTP system in estimating the emissivity [19] is presented. The theoretical work, presented in [19], is briefly summarized here for reference purpose.

6.2 Theory [19]

The equation for the emission function $E(T(r,t))$ for a non-gray body (2.4) and spectral emissivity model of (2.5) were used in developing the heat transfer model of the RTP system and are repeated below for convenience;

$$E(T) = \sum_{l=l_{\min}}^{l_{\max}} \alpha_l p_l(T) T^{4-l} \tag{2.4}$$

$$\varepsilon(\lambda,T) = \sum_{l=l_{\min}}^{l_{\max}} p_l(T)\lambda^l \quad (2.5)$$

By estimating the temperature dependent parameters $p_l(T)$, the emissivity at the given temperature T_o can be estimated. In order to estimate the parameters, a uniform and persistently-excited temperature trajectory near T_o is generated. The reference trajectory so generated can be described by the following equations;

$$\begin{aligned} T(t) &= T_o x(t) \\ x(t) &= 1 + \Delta x(t), |\Delta x(t)| \approx 10^{-2} \end{aligned} \quad (6.1)$$

The equation for $x(t)$ can then be expressed as follows;

$$\dot{x} = \theta_o - \theta_4 x^4 - \theta_5 x^5 - \theta_6 x^6 + g(\mathbf{U}) \quad (6.2)$$

By using approximation

$$(1+\Delta x)^n = 1 + n\Delta x + \frac{n(n+1)}{2}\Delta x^2 + O \quad (6.3)$$

the equation for $\Delta x(t)$ can be represented as follows;

$$\Delta\dot{x} = -a_0 - a_1\Delta x - a_2\Delta x^2 + g(\mathbf{U}) \quad (6.4)$$

where parameters a_n are related to θ s by the following equations;

$$\begin{aligned} \theta_4 &= 15a_0 - 5a_1 + a_2 + 15\theta_0 \\ \theta_5 &= -24a_0 + 9a_1 - 2a_2 - 24\theta_0 \\ \theta_4 &= 10a_0 - 4a_1 + a_2 + 10\theta_0 \end{aligned} \quad (6.5)$$

Thus the problem of estimating emissivity has been reduced to identification of parameters a_n of (6.4) and θ_0. The dynamic observer used for estimating parameters a_n is ;

$$\hat{a}_n = -K_n \frac{\Delta x^{n+1}}{n+1} + z_n, \quad n = 0,1,2$$
$$\dot{z}_n = K_n \Delta x^n (-\hat{a}_0 - \hat{a}_1 \Delta x - \hat{a}_2 \Delta x^2 + g(\mathbf{U}) \tag{6.6}$$

where $\hat{a}_n$ are the estimated parameters, $K_n > 0$ are the gains, and z_n are the internal states of the observer.

The inclusion of control function $g(\mathbf{U})$ in the observer equation (6.6) is undesirable due to lack of accuracy in the dynamics of the lamp radiation. Therefore, in order to remove dependency on control function $g(\mathbf{U})$ from the observer equation, pseudo-impulse function is generated as the control function. The pseudo-impulse function is generated by allowing the wafer temperature to increase to, say, $T_o + 100K$, and then allow the wafers to cool. When the temperature becomes lower than $T_o - 10K$ the impulse control function should change the temperature to $T_o + 10K$. In order to obtain such temperature measurements, the experiment is conducted using a square wave reference trajectory. During the cooling down period, the control function is zero given that the uniformity of temperatures is maintained. By recording the temperature measurements during the cooling down period near 600K, the artificial temperature trajectory of Figure 6.1 is produced. This trajectory is equivalent to the temperature measurements that would be measured when the control input is a series of impulses. Using impulse control function as $g(\mathbf{U})$, the state equation for the observer can be represented as follows;

$$z_0(t+0) = z_0(t-0) + K_0 A$$
$$z_1(t+0) = z_1(t-0) + K_1(\Delta x(t-0)A + A^2/2) \tag{6.7}$$
$$z_2(t+0) = z_2(t-0) + K_2((\Delta x(t-0))^2 A + \Delta x(t-0)A^2 + A^3/3)$$

where $A = (T(t+0)+T(t-0))/T_0$ is the amplitude of the impulse function, t is the time at which the impulse function is applied, $T(t-0)$ is the temperature reading just before the impulse function is applied, and $T(t+0)$ is the temperature reading just after the impulse function is applied.

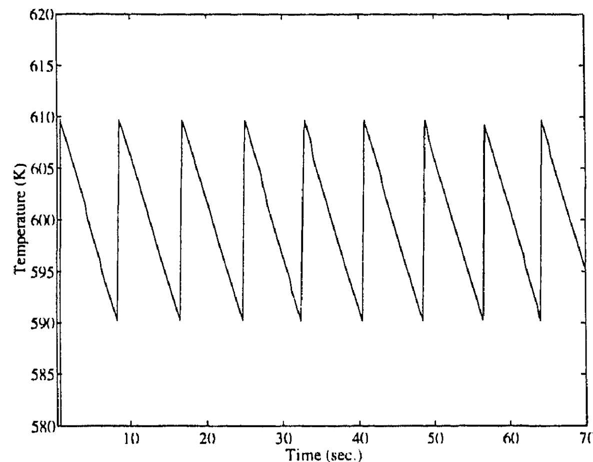

Figure 6.1. Temperature trajectory due to impulse control function generated from experimental data.

6.3 Results of the Experiment [19]

Using formula $\varepsilon(\lambda) = 0.72 - 0.96[\mu m]\lambda^{-1} + 0.72[\mu m]\lambda^{-2}$ at temperature $T_0 = 600K$ the parameters of (6.4) and (6.5) are $a_0 \approx 2.62\cdot 10^{-3}, a_1 \approx 8.93\cdot 10^{-3}, a_2 \approx 1.13\cdot 10^{-2}$. The simulation run showing convergence of estimated parameters to the above values are shown in Figure 6.2. Figure 6.3 shows the emissivity function calculated from the estimated parameters. Figure 6.4 shows the result of parameter estimation using the experimentally measured temperatures showing convergence of parameters. Figure 6.5 shows a plot of emissivity function using the estimated parameters.

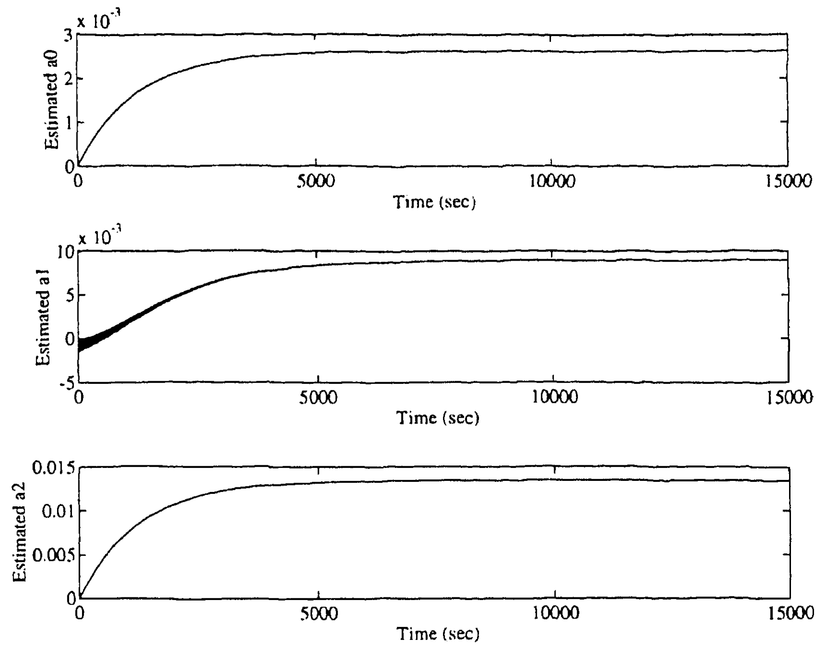

Figure 6.2. Estimation of parameters in simulation.

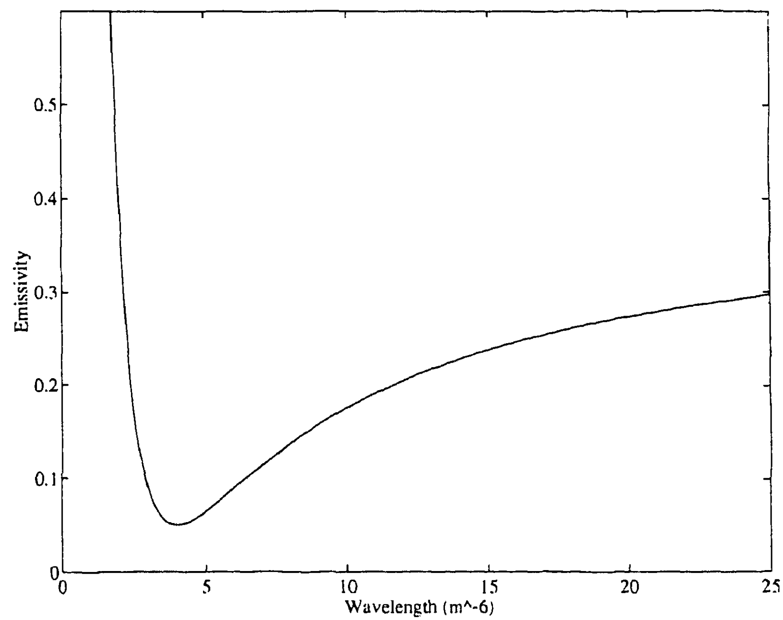

Figure 6.3. Emissivity function of estimated parameters.

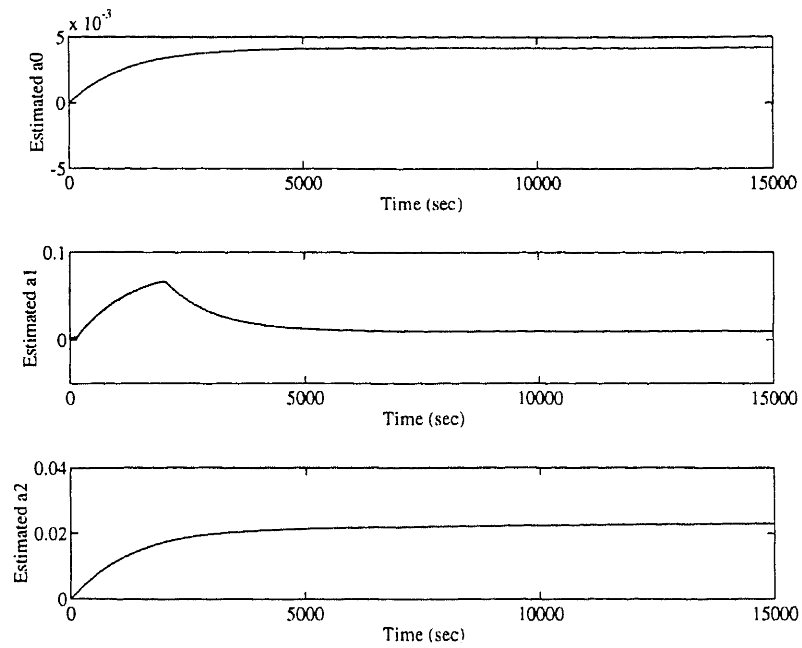

Figure 6.4. Estimated parameters using experimental data.

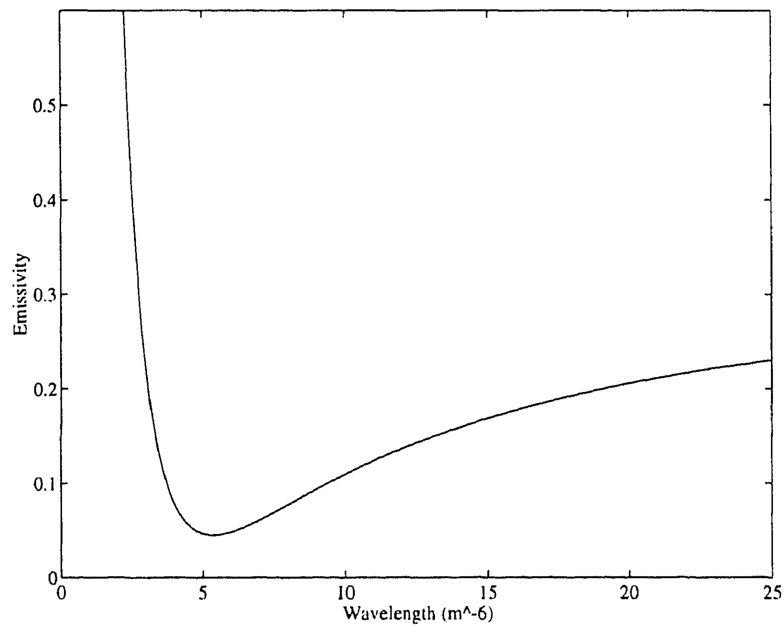

Figure 6.5. Emissivity function of estimated parameters.

CHAPTER 7

CONCLUSION

The initial performance goal of the adaptive control system was to achieve temperature tracking error to within 1K, and to achieve uniformity of the measured temperatures to within 1K. These are the types of performance necessary in order to produce a practical RTP system. The experimental results presented in this thesis fall below these goals. However, results obtained from the simulation and the experimental studies point ways to further improving the performance of the control system so that the initial performance goals can be achieved. The compensation of the lamp ring hysteresis in the system model as well as obtaining suitable wafer with at least three thermocouples, preferably more, should have high priority. As presented in the experimental results, the use of Extended Kalman Filter as the state estimator reduces the temperature measurement noise, thus improving certain aspect of the system performance. But this improvement comes at a cost of increase in steady state error. Also, lacking any theoretical background in the robustness or stability of the system using combined state and parameter estimation, it appears unlikely that EKF will replace the state estimator of the adaptive control algorithm. But using EKF for the filtering purpose, in conjunction with the original state estimator may indeed provide overall improvement in the performance of the control system.

The control system, as implemented, provides flexible and convenient means by which further tuning of the system can be achieved. The attention to modularity in the software design makes it possible to adapt the system to varying experimental needs with relative ease. The availability of real time simulation system also provides means by

which new algorithms and experiments can be tested before they are implemented in the real system. The graphical user interface provides to the experimenter the ease of operation and the real-time plotting capability allows the experimenter to analyze the system as the experiment is being performed. The experimental data are also recorded in binary data format thus allowing the experimenter to analyze the data on another platform using various analysis software packages. The implementation of the system using powerful parallel processing system means that more complex tasks can be performed without having to overhaul the entire system. The system as implemented therefore provides a powerful platform which will afford flexibility and usefulness as well as portability.

APPENDIX A

Adaptive Controller Host Program Listing

```
//
// prt.h
//

#define SkipSize 0
#define DataSize 9
#define DataLength 500

static int skip=0;
static int perm=0;
double pData[DataLength][DataSize];
static int pCnt=0;

// end prt.h
```

```
/* ================================================================== */
/* LabWindows User Interface Resource (UIR) Include File              */
/* Copyright (c) National Instruments 1993. All Rights Reserved.      */
/*                                                                    */
/* WARNING: Do not add to, delete from, or otherwise modify the contents */
/*          of this include file.                                     */
/* ================================================================== */
/*
** rtpl.h
*/

#define  mPanel  0
#define  mPanel_plotU  0
#define  mPanel_plotT  1
#define  mPanel_plotS  2
#define  mPanel_plotL  3
#define  mPanel_S1  4
#define  mPanel_S2  5
#define  mPanel_S3  6
#define  mPanel_U1  7
#define  mPanel_U2  8
#define  mPanel_U3  9
#define  mPanel_T1  10
#define  mPanel_T2  11
#define  mPanel_T3  12
#define  mPanel_text  13
#define  mPanel_text2  14
#define  mPanel_text3  15
#define  mPanel_text4  16
#define  mPanel_temp1  17
#define  mPanel_temp2  18
#define  mPanel_temp3  19

#define  sPanel  1
#define  sPanel_text1  0
#define  sPanel_num1  1
#define  sPanel_text2  2
#define  sPanel_num2  3
#define  sPanel_text3  4
#define  sPanel_num3  5
#define  sPanel_text4  6
#define  sPanel_num4  7
#define  sPanel_file  8
#define  sPanel_print  9
#define  sPanel_start  10
#define  sPanel_reset  11
#define  sPanel_pause  12
#define  sPanel_exit  13

// end rtpl.h
```

```
//
// COMMON.HPP
//     Written : Feb. 15 1994
//     Revised : Jun. 20 1995
//

#if !defined(__COMMON_HPP)
#define __COMMON_HPP

#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <ctype.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#define nameLength 80    // ".btl" file name length
#define TRUE 1
#define NONE 0
#define FALSE 0
#define RESET 0
#define ZERO 2048
#define CHAN_1 0
#define CHAN_2 1
#define CHAN_3 2
#define numProc 3
#define numChan 3

// definition of error codes ...
#define bootError 1
#define linkOpenError 2
#define driverModeError 3
#define mallocErrorRead 4
#define readNotReady 5
#define invalidSize 6
#define bigSize 7
#define readCount 8
#define mallocErrorWrite 9
#define writeError 10
#define writeNotReady 11
#define tmbErrorSet 12

typedef struct
{
  public:
      // LW event control
      int done; // loop done flag
      int go;   // loop start flag
      int reset;// loop reset flag
      int ID;   // not used

      // Data exchange protocol
      int error; // tmb error code
      int count; // tmb.read data count

      // Process information
      int adtIn [numChan];
      int datOut[numChan];
      int status[numProc];
      int SolExist;
      long loopCount;
      double lambda;
      double dt;
      double t ;
      double T [numChan];
      double U [numChan];
      double S [numChan];
}EVENT;

void initialize (EVENT &);
void errorHandler(EVENT &);
#endif

// end common.h
```

```
//
// rtpmain.hpp
//

#include "xputer.hpp"
#include "common.hpp"
#include "lw.hpp"

#define TIMER 0

extern "C"{
int  SetIntLevel     (int);
int  AnalogRead      (int);
void AnalogWrite     (int, int);
int  BinaryRead      (void);
void BinaryWrite     (int);
void EnableISR       (void (*)(void), int, double);
void DisableISR      (void);
}

// end rtpmain.h
```

```
//
//   LW.hpp   Version 1.0
//         Written : Feb. 15 1994
//         Revised : Jun. 20 1995
//

#if !defined(__LW_HPP)
#define __LW_HPP

#include "d:\lw\include\userint.h"
#include "d:\lw\include\lwsystem.h"
#include "d:\lw\include\ui_attr.h"
#include "rtp.h" // LabWindow generated include file
#include "common.hpp"

#define doubleType 4
#define traceNumber 3

class LabWindow
{
    public:
       LabWindow (char *);
       ~LabWindow(void);

       void Event(EVENT &);
       void Plot (EVENT &);

    private :
       void fStart (EVENT &);
       void fPause (EVENT &);
       void fFile  (EVENT &);
       void fExit  (EVENT &);
       void fPrint (void);
       void fReset (EVENT &);

       int panelHandle [2];
       int eventPanelID[1];
       int eventControl[1];
};

#endif
// end lw.hpp
```

```
//
// Xputer.hpp
//

#include "common.hpp"

#define LINK int
#define BOOLEAN int
#define readSizeLimit 512

class Transputer
{
   public:
       Transputer(EVENT &, char *);
       ~Transputer();
       void Read(EVENT &);
       void Write(EVENT &);
       void Error(EVENT &);
   private:
       LINK devHandle;
       BOOLEAN testError(LINK);
       LINK linkOpen(char*);
       int readLink (LINK, long*);
       int writeLink(LINK, long*);
       int setDriverMode(LINK, unsigned*);
       int bootTMB(char*);
};

typedef struct
{
      int dataCount;
      int loopDoneFlag;
      int resetFlag;
}ToTMB;

typedef struct
{
      int lead;
      int adtIn [numChan];
      int datOut[numChan];
      int status[numProc];
      int SolExist;
      long loopCount;
      double lambda;
      double t ;
      double T [numChan];
      double U [numChan];
      double S [numChan];
}FromTMB;

// end xputer.hpp
```

```
//
// rtpmain.cpp: Description of main function
//
#include "rtpmain.hpp"
#include <fcntl.h>
#include <sys\stat.h>
#include "prt.h"

double tt=0.2;
double dtt=0.2;

EVENT event;
LabWindow  LW ("rtp.uir");
Transputer TMB(event, "bootFile.btl");

// timer interrupt service routine
void isr()
{
     if (event.go)
     {
         tt += dtt;
         TMB.Read (event);
         TMB.Write(event);
         TMB.Error(event);

         // save data
         if (!perm)
         {
            if (pCnt<DataLength)
            {
               pData[pCnt][0] =event.T[0];
               pData[pCnt][1] =event.T[1];
               pData[pCnt][2] =event.T[2];
               pData[pCnt][3] =event.U[0];
               pData[pCnt][4] =event.U[1];
               pData[pCnt][5] =event.U[2];
               pData[pCnt][6] =event.S[0];
               pData[pCnt][7] =event.S[1];
               pData[pCnt][8] =event.S[2];
            }
            else
               perm=1;
         }
         pCnt++;

     }
     event.SolExist=pCnt;
     event.dt = dtt;
}


void main(void)
{
    int i,chan,fp;

 // Begin initialization ...
    initialize (event);
    if (event.error != NONE)
        errorHandler(event);

    SetIntLevel(7);
    EnableISR(isr, TIMER, dtt);
 // End of initialization ...

    while (!event.done)
    {
     // get next event ...
        LW.Event(event);

     // event handler ...
        if (event.go)
        {
            if (event.error != NONE)
            {
                event.done = TRUE;
                break;
```

```
            }
            LW.Plot (event);
        }
    }

 // Save plot data to file ...
    _fmode=O_BINARY;
    fp=creat("matfile.dat",S_IREAD|S_IWRITE);
    write (fp, pData, sizeof(pData));
    close(fp);

 // Disable interrupt ...
    DisableISR();

 // Transputer terminating sequence ...
    TMB.Read (event);
    TMB.Write(event);

 // is termination due to error?
    if (event.error != NONE)
        errorHandler(event);

 // for normal termination ...
    MessagePopup("Transputer link terminated ...");
    exit (0);
}

// end rtpmain.cpp
```

```
//
//   LWFUNC.CPP: LabWindow class member function descriptions
//               Written : Mar. 08 1994
//

#include "lw.hpp"

static double prev[3];
static double max[3];
static double min[3];

void LabWindow :: fStart(EVENT &event)
{
   event.go    = TRUE;
   event.reset = FALSE;
   event.done  = FALSE;
}

void LabWindow :: fPause(EVENT &event)
{
   event.go    = FALSE;
   event.reset = FALSE;
   event.done  = FALSE;
}

void LabWindow :: fReset(EVENT &event)
{
   int i;

   for (i=0; i<numChan; i++)
   {
        DefaultCtrl (panelHandle[mPanel], mPanel_T1+i);
        DefaultCtrl (panelHandle[mPanel], mPanel_U1+i);
        DefaultCtrl (panelHandle[mPanel], mPanel_S1+i);
        DefaultCtrl (panelHandle[mPanel], mPanel_temp1+i);
   }
   DefaultCtrl (panelHandle[sPanel], sPanel_num1);
   DefaultCtrl (panelHandle[sPanel], sPanel_num2);
   DefaultCtrl (panelHandle[sPanel], sPanel_num3);
   DefaultCtrl (panelHandle[sPanel], sPanel_num4);
   ClearStripChart (panelHandle[mPanel], mPanel_plotL);
   ClearStripChart (panelHandle[mPanel], mPanel_plotU);
   ClearStripChart (panelHandle[mPanel], mPanel_plotT);
   ClearStripChart (panelHandle[mPanel], mPanel_plotS);
}


void LabWindow :: fExit(EVENT &event)
{
   if (ConfirmPopup ("Are you sure you want to quit?"))
   {
       event.done = TRUE;
       event.go   = FALSE;
   }
}


void LabWindow :: fPrint(void)
{
     int stat=0;

     stat = OutputScreen (-1,"screen.eps");
     if (stat < 0)
       MessagePopup("Error in Printing screen to default device");
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)stat);

     stat = OutputGraph (-1,"plotT.eps",0,panelHandle[mPanel],mPanel_plotT);
     if (stat < 0)
       MessagePopup("Error in Printing plot T to default device");
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)stat);

     stat = OutputGraph (-1,"plotU.eps",0,panelHandle[mPanel],mPanel_plotU);
     if (stat < 0)
       MessagePopup("Error in Printing plot U to default device");
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)stat);
```

```
     stat = OutputGraph (-1,"plotS.eps",0,panelHandle[mPanel],mPanel_plotS);
     if (stat < 0)
       MessagePopup("Error in Printing plot S to default device");
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)stat);

     stat = OutputGraph (-1,"plotL.eps",0,panelHandle[mPanel],mPanel_plotL);
     if (stat < 0)
       MessagePopup("Error in Printing plot L to default device");
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)stat);

}


void LabWindow :: fFile(EVENT &event)
{
    // not implemented yet
}


void LabWindow :: Plot(EVENT &event)
{
     int chan;
     double temp[3];

     for (chan=0; chan<numChan; chan++)
     {
          temp[chan] = event.T[chan]-event.S[chan];
     }
     temp[0]=event.lambda;
     // plot strip chart ...
     PlotStripChart (panelHandle[mPanel], mPanel_plotT, event.T ,3, 0, 0, doubleType);
     PlotStripChart (panelHandle[mPanel], mPanel_plotU, event.U ,3, 0, 0, doubleType);
     PlotStripChart (panelHandle[mPanel], mPanel_plotS, event.S ,3, 0, 0, doubleType);
     PlotStripChart (panelHandle[mPanel], mPanel_plotL, temp    ,3, 0, 0, doubleType);

     for (chan=0; chan<numChan; chan++)
     {
          SetCtrlVal (panelHandle[mPanel], mPanel_T1+chan, event.T[chan]);
          SetCtrlVal (panelHandle[mPanel], mPanel_U1+chan, event.U[chan]);
          SetCtrlVal (panelHandle[mPanel], mPanel_S1+chan, event.S[chan]);
          if (event.loopCount > 25)
          {
               if (temp[chan] > max[chan])
                  max[chan] = temp[chan];
               if (temp[chan] < min[chan])
                  min[chan] = temp[chan];
          }
          SetCtrlVal (panelHandle[sPanel], sPanel_num1+(chan*2), min[chan]);
          SetCtrlVal (panelHandle[mPanel], mPanel_temp1+chan   , max[chan]);
     }

     //SetCtrlVal (panelHandle[mPanel], mPanel_lambda, event.lambda);
     //SetCtrlVal (panelHandle[sPanel], sPanel_num1, (double)event.loopCount);
     //SetCtrlVal (panelHandle[sPanel], sPanel_num2, event.t);
     //SetCtrlVal (panelHandle[sPanel], sPanel_num3, event.dt);
     SetCtrlVal (panelHandle[sPanel], sPanel_num4, (double)event.SolExist);
     for (chan=0; chan<numChan; chan++)
     {
          prev[chan] = temp[chan];
     }

}


void LabWindow :: Event(EVENT &event)
{
   GetUserEvent (0, &eventPanelID[0], &eventControl[0]);
   if (eventPanelID[0] == panelHandle[sPanel])
   {
       switch (eventControl[0])
       {
            case sPanel_start:
                 fStart(event);
                 break;

            case sPanel_pause:
```

```
                fPause(event);
                break;

            case sPanel_reset:
                fReset(event);
                break;

            case sPanel_exit:
                fExit(event);
                break;

            case sPanel_print:
                fPrint();
                break;

            default:
                MessagePopup("Function called is not yet implemented");
                break;
        }
    }
}

LabWindow :: LabWindow (char *uirFile)
{
    int i;

   // load and display panel(s)
    panelHandle[mPanel] = LoadPanel (uirFile, mPanel);
    panelHandle[sPanel] = LoadPanel (uirFile, sPanel);
    DisplayPanel (panelHandle[mPanel]);
    DisplayPanel (panelHandle[sPanel]);

   // set text box labels
    SetCtrlVal (panelHandle[mPanel], mPanel_text, "Temperature");
    SetCtrlVal (panelHandle[mPanel], mPanel_text2,"Kalman Gains");
    SetCtrlVal (panelHandle[mPanel], mPanel_text3,"Estimates");
    SetCtrlVal (panelHandle[mPanel], mPanel_text4,"T - Test");
    SetCtrlVal (panelHandle[sPanel], sPanel_text1,"Loop Counter");
    SetCtrlVal (panelHandle[sPanel], sPanel_text2,"Total Time");
    SetCtrlVal (panelHandle[sPanel], sPanel_text3,"Interrupt");
    SetCtrlVal (panelHandle[sPanel], sPanel_text4,"Sol Exist");
    for (i=0; i<3; i++)
    {
        max[i] = 0.;
        min[i] = 0.;
    }
}

LabWindow :: ~LabWindow(void)
{
     // null function
}

// end lwfunc.cpp
```

```
//
// Xputer.cpp: Transputer interface object member function description
//

#include "xputer.hpp"

LINK Transputer::linkOpen(char* name)
{
     int tmp;

     tmp = open(name,O_BINARY|O_RDWR);
     if(ioctl(tmp,0,0,0)&0x80)
        return(tmp);
     else
     {
        close (tmp);
        return(-1);
     }
}

BOOLEAN Transputer::testError(LINK handle)
{
     unsigned long flag[1];

     ioctl(handle,2,flag,4);
     if (flag[0]&0x01)
        return TRUE;
     else
        return FALSE;
}

void Transputer::Error (EVENT &event)
{
      int err;

      if (testError(devHandle))
          event.error = tmbErrorSet;
      else
          event.error = NONE;
}

int Transputer::bootTMB(char* command)
{
     int stat;

     stat = system("iserver /sr /sc prog4.btl");
     if (stat < 0)
         return -1;
     else
         return 1;
}

int Transputer::setDriverMode(LINK handle, unsigned* stat)
{
     ioctl (handle, 1, stat, 0);
     if (testError(handle))
     {
         close(handle);
         return -1;
      }
      return 1;
}


void Transputer :: Read (EVENT &event)
{
     int i,j,cnt;
     char size[2];
     int *ptr;
     char *in;
     unsigned long stat[1];
     FromTMB *message;
     FromTMB msg;

     in = (char *)malloc(sizeof(msg));
     if (in < 0)
```

```
{
    event.error = mallocErrorRead;
    return;
}
size[0] = 0;
size[1] = 0;

ioctl (devHandle, 2, stat, 4);
if (stat[0]&0x08)
{
    read (devHandle,&size[0],1);
}
else
{
    event.error = readNotReady;
    free(in);
    return;
}

if (size[0] < 0)
{
    event.error = invalidSize;
    free(in);
    return;
}
ioctl (devHandle, 2, stat, 4);
if (stat[0]&0x08)
{
    read (devHandle,&size[1],1);
}
else
{
    event.error = readNotReady;
    free(in);
    return;
}
cnt = (int)(size[0]+size[1]*256);
if (cnt > readSizeLimit)
{
    event.error = bigSize;
    free(in);
    return;
}
ioctl (devHandle, 2, stat, 4);
if (stat[0]&0x08)
{
    j = read (devHandle, in, cnt);
}
else
{
    event.error = readNotReady;
    free(in);
    return;
}
if (j != cnt)
{
    event.error = readCount;
    event.count = j;
    free(in);
    return;
}
event.error = NONE;
event.count = (int)(j);

message = (FromTMB *)in;
for (i=0; i<numChan; i++)
{
     event.adtIn [i] = message->adtIn [i];
     event.datOut[i] = message->datOut[i];
     event.T[i]    = message->T[i];
     event.U[i]    = message->U[i];
     event.S[i]    = message->S[i];
}
for (i=0; i<numProc; i++)
{
     event.status[i] = message->status[i];
```

```
    }
    event.t = message->t;
    event.loopCount= message->loopCount;
    event.SolExist = message->SolExist;
    event.lambda   = message->lambda;

    free(in);
}


void Transputer :: Write (EVENT &event)
{
    int i, cnt;
    int dataSize, messageSize;
    int *message;
    unsigned char out[22];
    unsigned char *ptr;
    unsigned long stat[1];

    dataSize = 4;
    messageSize = dataSize + 2;

    message = (int *)malloc(sizeof(i)*3);
    if (message < 0)
    {
        event.error = mallocErrorWrite;
        return;
    }
    message[0] = dataSize;
    message[1] = event.done;
    message[2] = event.reset;
    ioctl (devHandle, 2, stat, 4);
    if (stat[0]&0x04)
    {
        ptr = (char *)message;
        cnt = write(devHandle, message, messageSize);
        if (cnt != messageSize)
        {
            event.error = writeError;
        }
        else
        {
            event.error = NONE;
        }
        free(message);
        return;
    }
    else
    {
        event.error = writeNotReady;
        free(message);
        return;
    }
}


Transputer :: Transputer(EVENT &event, char *fileName)
{
    unsigned long info1[1];
    unsigned info[1];
    char* command;

// Boot transputer ...
   if (!bootTMB(command))
    {
        event.error = bootError;
    }

// Open handle to the device driver ...
    if ((devHandle = linkOpen("LINK1")) < 0)
    {
        event.error = linkOpenError;
    }

// Set device driver mode to binary/raw mode ...
```

```
    info[0] = 0x20;
    if (setDriverMode(devHandle, &info[0]) < 0)
    {
         event.error = driverModeError;
    }
}


Transputer :: ~Transputer()
{
     close(devHandle);
}


// end xputer.cpp
```

```
//
// common.cpp: Common public function
//
#include "common.hpp"

void initialize(EVENT &event)
{
    int i;
    char tmp;

    event.go = FALSE;
    event.reset = FALSE;
    event.done = FALSE;
    event.error = NONE;
    event.count = RESET;
}

// end common.cpp
```

```
//
// error.cpp: Public error handler
//

#include "lw.hpp"
#include "common.hpp"

void errorHandler(EVENT &event)
{
     switch (event.error)
     {
        case 1:
             MessagePopup ("tmb initialize: boot failure ... ");
             break;
        case 2:
             MessagePopup ("tmb initialize: unable to open link... ");
             break;
        case 3:
             MessagePopup ("tmb initialize: unable to set driver mode ... ");
             break;
        case 4:
             MessagePopup ("tmb read: unable to allocate buffer space ...");
             break;
        case 5:
             MessagePopup ("tmb read: Transputer is not ready to send ... ");
             break;
        case 6:
             MessagePopup ("tmb read: data packet size is invalid... ");
             break;
        case 7:
             MessagePopup ("tmb read: data packet size is too big ... ");
             break;
        case 8:
             MessagePopup ("tmb read: received packet size incorrect ...");
             break;
        case 9:
             MessagePopup ("tmb write: unable to allocate buffer space ...");
             break;
        case 10:
             MessagePopup ("tmb write: unable to write ...");
             break;
        case 11:
             MessagePopup ("tmb write: Transputer is not ready to receive ...");
             break;
        case 12:
             MessagePopup ("tmb write: Transputer has raised error flag ...");
             break;

        default:
             MessagePopup ("General protection failure... aka. unknown error?");
             break;
     }
     exit(-1);
}

// end error.cpp
```

APPENDIX B

Adaptive Controller Transputer Program Listing

```
/*
** prog.cfs : Transputer hardware and software configuration description
*/


/* hardware config. */
T805 (memory = 4M) p0;
T805 (memory = 4M) p1;
T805 (memory = 4M) p2;
T805 (memory = 4M) p3;
edge   dat208;
edge   adt108;

connect p0.link[0] to host;
connect p0.link[2] to p1.link[1];
connect p0.link[3] to p3.link[3];
connect p1.link[0] to adt108;
connect p1.link[3] to p2.link[0];
connect p2.link[2] to p3.link[1];
connect p2.link[3] to p3.link[0];
connect p3.link[2] to dat208;

/* software config */
process (stacksize = 8k, heapsize = 50k);

process (interface( input in, output out,
                    input from_p1, output to_p1,
                    input from_p3, output to_p3)) master;
process (interface( input from_master, output to_master,
                    input from_p2, output to_p2,
                    input fromADT, output toADT)) w1;
process (interface( input from_p1, output to_p1,
                    input from_p3a, output to_p3a,
                    input from_p3b, output to_p3b)) w2;
process (interface( input from_p2a, output to_p2a,
                    input from_p2b, output to_p2b,
                    input fromDAT, output toDAT,
                    input from_master, output to_master)) w3;

input  from_host;
output to_host;
input fromDAT208;
output toDAT208;
input fromADT108;
output toADT108;

connect master.in to from_host;
connect master.out to  to_host;
connect master.from_p1 to w1.to_master;
connect master.to_p1 to w1.from_master;
connect master.from_p3 to w3.to_master;
connect master.to_p3 to w3.from_master;

connect fromADT108 to w1.fromADT;
connect toADT108 to w1.toADT;
connect w1.from_p2 to w2.to_p1;
connect w1.to_p2 to w2.from_p1;

connect w2.from_p3a to w3.to_p2a;
connect w2.to_p3a to w3.from_p2a;
connect w2.from_p3b to w3.to_p2b;
connect w2.to_p3b to w3.from_p2b;

connect fromDAT208 to w3.fromDAT;
connect toDAT208 to w3.toDAT;


/* network mapping */
use "master.lku" for master;
use "proc1.lku" for w1;
use "proc2.lku" for w2;
use "proc3.lku" for w3;
place from_host on host;
place to_host on host;
place fromDAT208 on dat208;
```

```
place toDAT208 on dat208;
place fromADT108 on adt108;
place toADT108 on adt108;
place master on p0;
place w1 on p1;
place w2 on p2;
place w3 on p3;

/* end prog.cfs */
```

```
/*
**      global.h
*/

#include <stdlib.h>
#include <misc.h>
#include <math.h>
#include <channel.h>
#include <process.h>

#define TRUE  1
#define FALSE 0

/* zero output on DAT */
#define ZERO 2048

/* define upper, lower limit for adt,dat */
#define maxLimit 4095
#define minLimit 0
#define scale 2048.

/* number of channels */
#define numChan 3

/* number of processors */
#define numProc 3

/* message from host elements count */
#define msgFromHostCount 3

/* definition of message from host array index */
#define loopDoneFlag 1
#define resetFlag 2

/* message to host data count offset */
#define dataOffset 2

/* type definition for interprocessor messages */
typedef struct
{
    short temp;  /* not used                     */
    short reset; /* reset command to processors */
    short loop;  /* loop done flag               */
    short adtIn [numChan];  /* ditto here                 */
    short datOut[numChan];
    short status[numProc];  /* process status             */
    int SolExist;
    double t;
    double lambda;
    double T[numChan];
    double U[numChan];
    double S[numChan];
    double V[numChan];
    double Th[numChan];
}MESSAGE;


typedef struct
{
      short dataCount;
      short lead;
      short adtIn [numChan];
      short datOut[numChan];
      short status [numProc];
      short SolExist;
      int loopCount;
      double lambda;
      double t;
      double T [numChan];
      double U [numChan];
      /*
      double S [numChan];
      */
      double Th[numChan];
}toHOSTmsg;
/* end global.h */
```

```
/*
** control.h
*/

/* constants */
const double BSC_Sigma=3.67e-8;         /* Stefan-Boltzman, W/(m^2 K^4) */
const double BSC_h=6.35e-4;             /* m                            */
const double BSC_RO=2330.;              /* kg/m^3                       */
const double BSC_JBess[M+1][M+1]={      /* Bessel Matrix                */
            {111., 111.,      111.,      111.},
            {111.,   1., 1.         , 1.      },
            {111.,   1., 0.692536 , 0.156641},
            {111.,   1., 0.0484812,-0.371476}
     };
const double BSC_JBessm1[M+1][M+1]={    /* JBess^-1                     */
            {111.,     111.,     111.,      111.},
            {111., 0.695535,-1.10285, 1.40731 },
            {111.,-1.38689 , 3.60163,-2.21475 },
            {111., 1.69135 ,-2.49878, 0.807431}
     };
const double BSC_A[M+1]=                /* Bessel roots                 */
            {111., 0., 2529., 8478.};
const double BSC_G[M+1][M+1]={    /* lamp ring rad. func. */
            {111., 111.     , 111.      , 111.      },
            {111., 1.3710938, 8.3217773, 10.1225586},
            {111.,-0.0005957, 1.2109375, -0.3002930},
            {111., 1.9082031,-1.5888672, -1.0898438}
       };
const double BSC_Gm1[M+1][M+1]={  /* G^-1 */
            {111.,        111.,        111.,        111.},
            {111., 0.0586856 , 0.2290770 , 0.4819576 },
            {111., 0.0187361 , 0.6796630 ,-0.0132502 },
            {111., 0.0754373 ,-0.5897808 ,-0.0543878 }
       };

const double BSC_ga[M+1]=               /* exponents for lamp rings     */
             {111., 1.,1.,1.}; /*{111.,1., 0.38, 1.4};*/
const double BSC_lammax=2048.;              /* constants for feedback control*/
const double BSC_al[M+1]=               /* constants for observer       */
             {111.,1.e4,2.5e4,1.33e4};/* {111.,2.33e6,2.33e6,2.33e6}; */
const double BSC_Umax[M+1]=             /* max current of the lamps     */
             {111.,2048.,2048.,2048.};
const double BSC_dt=0.2;                /* time interval ,was 0.001     */

/* variables */
double BSC_t;
double BSC_lambda=1.;
double BSC_T[M+1]=/*{111.,290.,310.,280.};*/ {111.,373.,373.,373.};
double BSC_Th[M+1];
double BSC_Tbar[3]=                     /* Tbar[1]+Tbar[2]*t            */
             {111., 375., 10.};/*{111., 1000., 0.};*/
double BSC_u[M+1]=/*{111.,0.27,0.,0.23};*/ {111.,0.,0.,0.};
double BSC_Y[M+1];                      /* bessel coefficients          */
double BSC_Ym1[M+1]=                    /* previous bessel coefficients */
             {111.,0., 0., 0.};
double BSC_P[M+1]=                      /* estimating parameters        */
             {111.,0., 0., 0.};
double BSC_Pm1[M+1]=                    /* estimating parameters        */
             {111.,0., 0., 0.};
double BSC_S[M+1]=                      /* estimating parameters of     */
             {111.,0., 0., 0.};         /* the environment              */
int    BSC_SolExist=1;
int cnt;
double tem[M+1];
double ufc[M+1];

/* function prototypes */
double c(double);
double k(double);
double E(double);
void   equforz(double t, double *z, double *dzdt);
void   rk4m(double *z0, int n, double t, double dt,
       double *z, void (*derivs)(double, double *, double *));

/* end control.h */
```

```
/*
** ekf.h
*/

extern double w[3];
extern double ak[3];
extern double ph[7];
extern double phm[7];
extern double xh[4];
extern double xhm[4];
extern double a[5];
extern double kg[3];
extern double Eo;
extern double al;

void TimeUpdate(double, double *, double *);
void MeasUpdate(double, double *, double *);
void estimateT(void);
void estimateP(void);
void ekf(void);

/* end ekf.h */
```

```
/*
**     param.h
*/

#define M 3 /* number of thermo couples */

        extern double BSC_t;
        extern double BSC_T[M+1];
        extern double BSC_Th[M+1];
        extern double BSC_u[M+1];
        extern double BSC_lambda;   /* adaptive variable for FB control */

        extern double BSC_Sigma;         /* Stefan-Boltzman */
        extern double BSC_h;     /* m */
        extern double BSC_RO;      /*kg/m^3 */


        extern double BSC_JBess[M+1][M+1];    /*Bessel Matrix */
        extern double BSC_JBessm1[M+1][M+1];  /*JBess^-1*/
        extern double BSC_A[M+1];              /* Bessel roots */


        extern double BSC_G[M+1][M+1];    /* lamp ring rad. func. */
        extern double BSC_Gm1[M+1][M+1];  /* G^-1 */
        extern double BSC_ga[M+1];        /*exponents for lamp rings */


        extern double BSC_Tbar[3];        /*Tbar[1]+Tbar[2]*t   */

        extern double BSC_lammax;       /*constants for feedback control */
        extern double BSC_al[M+1];        /*constants for observer */
        extern double BSC_Umax[M+1];     /* max current for the lamps */

        extern double BSC_dt;             /* time interval */


        extern double BSC_Y[M+1];         /*bessel coefficients */
        extern double BSC_Ym1[M+1];       /*previous bessel coefficients */


        extern double BSC_P[M+1];         /*estemating parameters */
        extern double BSC_Pm1[M+1];         /*estemating parameters */
        extern double BSC_S[M+1];         /*estimating parameters of the env. */
        extern int    BSC_SolExist;       /*control exist,=1;
                                    ubar>=0, but direct FB failed,=0
                                    ubar>=0, but corrected FB failed,=-1
                                    ubar<0, =-2*/

        extern double tem[M+1];
        extern double ufc[M+1];
        extern int cnt;


double volt1(double);
double volt2(double);
double volt3(double);
double c(double);
double k(double);
double E(double);
void   equforz(double t, double *z, double *dzdt);
double *rk4m(double *z0, int n, double t, double dt,
       double *z, void (*derivs)(double, double *, double *));
double *rk6m(double *z0, int n, double t, double dt,
       double *z, void (*derivs)(double, double *, double *));

/* end param.h */
```

```
/*
**      EQUFORZ.H
*/

extern const double BSC_A[];
extern const double BSC_al[];
extern const double BSC_dt;
extern const double BSC_RO;
extern const double BSC_h;
extern const double BSC_Sigma;
extern const double BSC_G[M+1][M+1];
extern const double BSC_ga[M+1];

extern double BSC_u[];
extern double BSC_Yml[];
extern double BSC_Y[];

/* end equforz.h */
```

```
/*
**      MISC.H
*/

#include <math.h>

#define M 3

/* function prototypes */
double c(double);
double k(double);
double E(double);
void   equforz(double t, double *z, double *dzdt);
void   rk4m(double *z0, int n, double t, double dt,
       double *z, void (*derivs)(double, double *, double *));

/* end misc.h */
```

```
/*
**      master.c : Root transputer program
**      6/19/95
*/

#include "global.h"

int main()
{
    MESSAGE message;
    MESSAGE reply;
    toHOSTmsg  msgToHost;

    Channel *fromHOST;
    Channel *toHOST;
    Channel *fromP1;
    Channel *toP1;
    Channel *fromP3;
    Channel *toP3;

    int chan;
    int counter;
    int msgFromHostSize;
    int msgToHostSize;
    int messageSize;
    int replySize;

    short msgFromHost[msgFromHostCount];

    /* assign channels */
    fromHOST = (Channel*) get_param(1);
    toHOST   = (Channel*) get_param(2);
    fromP1   = (Channel*) get_param(3);
    toP1     = (Channel*) get_param(4);
    fromP3   = (Channel*) get_param(5);
    toP3     = (Channel*) get_param(6);

    /* initialize ... */
    message.loop = TRUE;
    counter = 0;

    /* define message sizes */
    msgFromHostSize = sizeof(msgFromHost[0])*msgFromHostCount;

    msgToHostSize   = sizeof(msgToHost);
    messageSize = sizeof(message);
    replySize   = sizeof(reply  );

    /* main loop */
    while (TRUE)
    {
          /* read from host ...                 */
          ChanIn (fromHOST, msgFromHost, msgFromHostSize);
          if (msgFromHost[loopDoneFlag]) break;

          /* send message to processes ...    */
          ChanOut(toP1   , &message, messageSize);
          ChanIn (fromP3, &reply   , replySize);

          /* send reply to host ...           */
          for (chan=0; chan<numChan; chan++)
          {
               msgToHost.Th[chan]= reply.Th[chan];
               msgToHost.U[chan] = reply.U[chan];
               msgToHost.T[chan] = reply.T[chan];
               /*
               msgToHost.S[chan] = reply.S[chan];
               */
               msgToHost.adtIn [chan] = reply.adtIn [chan];
               msgToHost.datOut[chan] = reply.datOut[chan];
               msgToHost.status[chan] = reply.status[chan];
          }
          msgToHost.dataCount = (short)(msgToHostSize - dataOffset);
          msgToHost.loopCount = counter;
          msgToHost.lambda = reply.lambda;
          msgToHost.SolExist = (short)reply.SolExist;
```

```
        ChanOut (toHOST, &msgToHost, msgToHostSize);

        /* update loop counter              */
        counter++;
    }
    ChanIn  (fromP3  , &reply  , sizeof(reply  ));
    msgToHost.dataCount = (short)(msgToHostSize - 2);
    msgToHost.loopCount = counter;
    ChanOut  (toHOST, &msgToHost, msgToHostSize);

    exit_terminate(0);
}

/* end master.h */
```

```
/* process 1:  */

#include "param.h"
#include "adt108.h"
#include "global.h"
#define procID 0

extern void init_adt108(Channel *, Channel *);
extern int  convert(BYTE, BYTE);
extern void rtpinit(void);
extern void rtpsim(void);

int main()
{
     Channel *fromP2;
     Channel *toP2;
     Channel *fromM;
     Channel *toM;
     Channel *fromADC;
     Channel *toADC;

     MESSAGE message;
     MESSAGE reply;
     int messageSize;
     int replySize;
     int loop;
     int chan;
     int i,j;
     int sample[numChan];
     int firstTime;
     double temp;

     fromM = (Channel*)get_param(1);
     toM   = (Channel*)get_param(2);
     fromP2= (Channel*)get_param(3);
     toP2  = (Channel*)get_param(4);
     fromADC=(Channel*)get_param(5);
     toADC  =(Channel*)get_param(6);

     messageSize = sizeof(message);
     replySize   = sizeof(reply  );

     init_adt108(fromADC, toADC);
     /* rtpinit();*/

     loop = TRUE;
     firstTime = TRUE;

     while (loop)
     {
         ChanIn  (fromM, &message, messageSize  );
         loop =   message.loop;
         for (chan=0; chan<numChan; chan++)
         {
              sample[chan]=0;
              for (i=0; i<10; i++)
              {
                   sample[chan]+=convert(chan,GAIN_1);
                   for (j=0; j<100; j++);
              }
              message.adtIn[chan] = (short)(sample[chan]/10);
              for (j=0; j<100; j++);
         }
         for (chan=0; chan<numChan; chan++)
         {
              temp=0.;
              temp = 200.*((double)(message.adtIn[chan]-2048)/409.4);
              message.T[chan] = 273.15 + temp;
         }
         message.T[2]=message.T[1];
         ChanOut (toP2, &message, messageSize);

     }/* end while */

     exit_terminate(0);
} /* end process1.c */
```

```
/* process 2 : */

#include "global.h"
#include "param.h"
#include "ekf.h"

#define procID 1
#define T_ref 800.

extern void controlInit(void);
extern void control(void);
extern void ekf(void);

int main()
{
     Channel *fromP3a;
     Channel *toP3a;
     Channel *fromP3b;
     Channel *toP3b;
     Channel *fromP1;
     Channel *toP1;

     MESSAGE message;
     MESSAGE reply;
     int messageSize;
     int replySize;
     int chan,loop,firstTime,x;
     double counter,time;
     double t_ref;
     double Vo[M+1];
     int i,m=M;

     fromP1  = (Channel*)get_param(1);
     toP1    = (Channel*)get_param(2);
     fromP3a = (Channel*)get_param(3);
     toP3a   = (Channel*)get_param(4);
     fromP3b = (Channel*)get_param(5);
     toP3b   = (Channel*)get_param(6);

     messageSize = sizeof(message);
     replySize   = sizeof(reply  );
     loop = TRUE;
     x=2;
     firstTime=TRUE;
     counter = BSC_t;
     t_ref=100.;
     cnt=2;

     /* initial sequence */
     controlInit();
     while (loop)
     {
         /* temperature trajectory */
         if(BSC_t>=t_ref)
         {
            BSC_Tbar[1]=T_ref;
            BSC_Tbar[2]=0.;
         }

         ChanIn  (fromP1, &message, messageSize);
         loop =    message.loop;

         if (firstTime)
         {
             BSC_Tbar[1]=message.T[1];
             t_ref=(T_ref-BSC_Tbar[1])/BSC_Tbar[2];

             /* for ekf ... */
             xhm[1]=message.T[0];
             BSC_Ym1[1]=xhm[1];
             /* ...         */

             firstTime=FALSE;
         }

         message.T[2]=(double)(rand()/RAND_MAX)+(message.T[0]+message.T[1])/2.;
```

```
        for (chan=0; chan<numChan; chan++)
        {
             BSC_T[chan+1]= message.T[chan];
        }

        BSC_SolExist = 1;

        ekf();

        control();
        Vo[1] = volt1(BSC_u[1]/2048.);
        Vo[2] = volt2(BSC_u[2]/2048.);
        Vo[3] = volt3(BSC_u[3]/2048.);

        for (chan=0; chan<numChan; chan++)
        {

             message.U[chan] = kg[chan];
             /*
             message.U[chan] = BSC_u[chan+1];
             */
             message.S[chan] = BSC_S[chan+1];
             message.V[chan] = Vo[chan+1];
             message.Th[chan]= BSC_Th[chan+1];
        }
        message.T[2]=BSC_Tbar[1]+BSC_Tbar[2]*BSC_t;
        message.lambda    = BSC_lambda;
        message.SolExist = BSC_SolExist;
        ChanOut (toP3a  , &message  , messageSize  );

        BSC_t += BSC_dt;
        counter++;
        for(i=1;i<=m;i++)
            BSC_Ym1[i]=BSC_Y[i];
    }

    exit_terminate(0);
}

/* end process2.c */
```

```
/* process 3 : */
#include "dat208.h"
#include "global.h"
#define procID 2
extern void init_dat208(Channel *, Channel *);
extern void update_dat208(void);
extern void write_dat208(int, int);
extern char dat208_id(void);

int main()
{
    Channel *fromP2a;
    Channel *toP2a;
    Channel *fromP2b;
    Channel *toP2b;
    Channel *fromDAT;
    Channel *toDAT;
    Channel *fromM;
    Channel *toM;

    MESSAGE message;
    MESSAGE reply;
    int messageSize;
    int replySize;
    int chan, loop, dat_id, dat_out;

    fromP2a  = (Channel*)get_param(1);
    toP2a    = (Channel*)get_param(2);
    fromP2b  = (Channel*)get_param(3);
    toP2b    = (Channel*)get_param(4);
    fromDAT  = (Channel*)get_param(5);
    toDAT    = (Channel*)get_param(6);
    fromM    = (Channel*)get_param(7);
    toM      = (Channel*)get_param(8);

    messageSize = sizeof(message);
    replySize   = sizeof(reply  );

    /* initialize ... */
    init_dat208 (fromDAT, toDAT);
    dat_id = dat208_id ();
    loop = TRUE;

    for (chan=0; chan<numChan; chan++)
    {
         dat_out = 2048;
         write_dat208 (chan, dat_out);
    }
    update_dat208();

    while (loop)
    {
          ChanIn (fromP2a, &message  , messageSize);
          loop = message.loop;
          /* write to dat208 ...               */
          for (chan=0; chan<numChan; chan++)
          {
               dat_out = (int)(2048. + 2048.*message.V[chan]/5.);
               if (dat_out > maxLimit)
                    dat_out = maxLimit;
               if (dat_out < minLimit)
                    dat_out = minLimit;
               write_dat208 (chan, dat_out);
               message.datOut[chan] = (short)dat_out;
          }
          update_dat208();
          ChanOut (toM, &message, messageSize);
    }/* end while */

    /* ... reset ADT                           */
    for (chan=0; chan<numChan; chan++)
         write_dat208 (chan, ZERO);
    update_dat208();
    exit_terminate(0);
} /* end process3.c */
```

```
/*
**      CONTROL.C
**      written by: Sergey Belikov
**      revised by: David Hur
*/

#include "misc.h"
#include "control.h"
#include "global.h"

void controlInit(void)
{
      int i,j;
      int m=M;

      for (i=1; i<=m; i++)
      {
           BSC_Ym1[i]=0.;

           for (j=1; j<=m; j++)
                BSC_Ym1[i]+=(BSC_JBessm1[i][j]*BSC_T[j]);

      }

      BSC_t=BSC_dt;
}

void control(void)
{
     int m=M;                   /* number of thermocouples */
     int i,j;                   /* for cycles */
     double z[M+1], z0[M+1];    /* for observer */
     double ubarsq[M+1];        /* control voltage in corr.power */
     double abar[M+1];          /* for adaptive     */
     double bbar[M+1];          /* choise of lambda */
     double Umaxga[M+1];        /* Umax^ga */
     double temp,temp4;         /* temporal var. */
     double lamupp;
     double lamlow;
     double temvec[M+1];        /* temporal vector */


    /************* Y->Pm1 **********(11.13)**/
     for (i=1; i<=m; i++)
     {
          BSC_Pm1[i]=BSC_S[i]*(2./BSC_h);
          z0[i]=BSC_Pm1[i]-BSC_al[i]*BSC_Ym1[i];
     }

     rk4m (z0, m, 0., BSC_dt, z, equforz);

     for (i=1; i<=m; i++)
     {
          BSC_Pm1[i]=BSC_al[i]*BSC_Y[i]+z[i];
     }

    /************ Pm1->S ************(11.12)*****/
     for (i=1; i<=m; i++)
     {
          BSC_S[i]=(BSC_h/2.)*BSC_Pm1[i];
     }

    /********** Open Loop: S,Tbar->ubar *******(11.16)******/
     temp=BSC_Tbar[1]+BSC_Tbar[2]*BSC_t;
     temp4=temp*temp*temp*temp;
     BSC_P[1]=BSC_RO*c(temp)*BSC_Tbar[2]+E(temp)*temp4;
     BSC_P[2]=0.;
     BSC_P[3]=0.;

     temvec[1]=-BSC_S[1]+0.5*BSC_h*BSC_P[1];
     temvec[2]=-BSC_S[2];
     temvec[3]=-BSC_S[3];
     for (i=1; i<=m; i++)
     {
          ubarsq[i]=0.;
          for (j=1; j<=m; j++)
```

```
      {
           ubarsq[i]+=(BSC_Gm1[i][j]*temvec[j]);
           if (ubarsq[i]<0.)
               BSC_SolExist=-2;
      }

 }

/******* abar and bbar for u=-lambda*abar +bbar**********/
/*  abar  */
 for (i=1; i<=m; i++)
 {
      temvec[i]=BSC_Y[i];
 }
 temvec[1]-=temp;
 for (i=1; i<=m; i++)
 {
      abar[i]=0.;
      for (j=1; j<=m; j++)
      {
           abar[i]+=(BSC_Gm1[i][j]*temvec[j]);
      }
      abar[i]*=(0.5*BSC_h*BSC_RO*c(BSC_Y[1]));
 }
/*  bbar  */
 temvec[1]=BSC_Tbar[2]*BSC_RO*c(BSC_Y[1])
           +E(BSC_Y[1])*BSC_Y[1]*BSC_Y[1]*BSC_Y[1]*BSC_Y[1]-BSC_P[1];
 for (i=2; i<=m; i++)
 {
      temvec[i]=(BSC_A[i]*k(BSC_Y[1])
                +4.*E(BSC_Y[1])*BSC_Y[1]*BSC_Y[1]*BSC_Y[1])*BSC_Y[i]
                -BSC_P[i];
 }
 for (i=1; i<=m; i++)
 {
      bbar[i]=0.;
      for (j=1; j<=m; j++)
      {
           bbar[i]+=(BSC_Gm1[i][j]*temvec[j]);
      }
      bbar[i]=0.5*BSC_h*bbar[i]+ubarsq[i];
 }

/****************Calculation of lambda***************************/
 lamlow=0.;
 lamupp=BSC_lammax;
 for (i=1; i<=m; i++)
 {
      Umaxga[i]=pow(BSC_Umax[i],BSC_ga[i]);
 }

 for (i=1; i<=m; i++)
 {
      if (abar[i]==0.&&(bbar[i]<0.||bbar[i]>Umaxga[i])
            &&BSC_SolExist>-1)
          BSC_SolExist=-1;
      temp=bbar[i]/abar[i];
      temp4=(bbar[i]-Umaxga[i])/abar[i];
      if (abar[i]>0.)
      {
          if (lamlow<temp4) lamlow=temp4;
          if (lamupp>temp)  lamupp=temp;
      }
      if (abar[i]<0.)
      {
          if(lamlow<temp)  lamlow=temp;
          if(lamupp>temp4) lamupp=temp4;
      }
 }
 if (lamlow>lamupp && BSC_SolExist>-1)
     BSC_SolExist=-1;
 BSC_lambda=lamupp;

 if (BSC_lambda>1.) BSC_lambda=1.;
 if (BSC_lambda<0.) BSC_lambda=0.;
```

```
    /******************** u ************************************/
     for (i=1; i<=m; i++)
     {
          BSC_u[i]=-BSC_lambda*abar[i]+bbar[i];
     }
     if (BSC_SolExist<0)
     {
         for (j=1; j<=m; j++)
              BSC_u[j]=0.;
     }

     for (j=1; j<=m; j++)
     {
          BSC_u[j]=pow(BSC_u[j],1./BSC_ga[j]);
     }
}
/* end of control */
```

```
/*
** Extended Kalman Filter
*/

#include <math.h>
#include "global.h"
#include "param.h"
#include "ekf.h"

double w[3]={25.,25.,25.};
double ak[3]={0.,226775.43,760222.26};
double ph[7]={0.,.25,0.,0.,.25,0.,.25};
double phm[7]={0.,.25,0.,0.,.25,0.,.25};
double xh[4]={0.,0.,0.,0.};
double xhm[4]={0.,0.,0.,0.};
double a[5]={0.,0.,0.,0.,0.};
double kg[3]={0.,0.,0.};
double Eo=.46;
double al=.19e-7;
double dts=0.05;
int m=M;

void TimeUpdate(double t, double *x, double *dxdt)
{
   double p;
   double p1,p2,p3;


   /* Error Covariance Matrix */
   p1=x[1]+x[2]+x[3];
   p2=x[2]+x[4]+x[5];
   p3=x[3]+x[5]+x[6];
   p=p1+p2+p3;

   dxdt[1]=2.*a[0]*x[1]+w[0]-p*x[1]/25.;
   dxdt[2]=a[1]*x[1]+(a[0]+a[2])*x[2]-p*x[2]/25.;
   dxdt[3]=a[3]*x[1]+(a[0]+a[4])*x[3]-p*x[3]/25.;
   dxdt[4]=2.*a[1]*x[2]+2.*a[2]*x[4]+w[1]-p*x[4]/25.;
   dxdt[5]=a[3]*x[2]+a[1]*x[3]+(a[2]+a[4])*x[5]-p*x[5]/25.;
   dxdt[6]=2.*a[3]*x[3]+2.*a[4]*x[6]+w[2]-p*x[6]/25.;

}


void MeasUpdate(double t, double *x, double *dxdt)
{

   /* Estimate Update:X=X+K(Y-h) */
   dxdt[1]=(1./(BSC_RO*c(x[1])))*
           (-E(x[1])*x[1]*x[1]*x[1]*x[1]+(2./BSC_h)
            *(BSC_G[1][1]*BSC_u[1]+BSC_G[1][2]*BSC_u[2]+BSC_G[1][3]*BSC_u[3]+BSC_S[1])
           )+kg[0]*(BSC_T[1]-(x[1]+x[2]+x[3]));

   dxdt[2]=(1./(BSC_RO*c(x[1])))*
           (-(BSC_A[2]*k(x[1])+4*E(x[1])*x[1]*x[1]*x[1])*x[2]+(2./BSC_h)
            *(BSC_G[2][1]*BSC_u[1]+BSC_G[2][2]*BSC_u[2]+BSC_G[2][3]*BSC_u[3]+BSC_S[2])
           )+kg[1]*(BSC_T[1]-(x[1]+x[2]+x[3]));

   dxdt[3]=(1./(BSC_RO*c(x[1])))*
           (-(BSC_A[3]*k(x[1])+4*E(x[1])*x[1]*x[1]*x[1])*x[3]+(2./BSC_h)
            *(BSC_G[3][1]*BSC_u[1]+BSC_G[3][2]*BSC_u[2]+BSC_G[3][3]*BSC_u[3]+BSC_S[3])
           )+kg[2]*(BSC_T[1]-(x[1]+x[2]+x[3]));
}


void estimateT()
{
    int i,j;
    int m=M;

    for (i=1;i<=m;i++)
    {
         BSC_Y[i]=xh[i];
         BSC_Th[i]=0.;
         for (j=1;j<=m; j++)
              BSC_Th[i]+=BSC_JBess[i][j]*BSC_Y[j];
```

```
    }

}

void ekf(void)
{
   int i,j;
   double tt;

   /* Jacobian Components */
   a[0]=-al*4.*Eo*xhm[1]*xhm[1]*xhm[1];
   a[1]=-12.*al*Eo*xhm[1]*xhm[1]*xhm[2];
   a[2]=-al*(ak[1]+4.*Eo*xhm[1]*xhm[1]*xhm[1]);
   a[3]=-12.*al*Eo*xhm[1]*xhm[1]*xhm[3];
   a[4]=-al*(ak[2]+4.*Eo*xhm[1]*xhm[1]*xhm[1]);

   /* time update */
   tt=BSC_t;
   for(i=0; i<4; i++)
   {
       rk6m(phm,6,tt,dts,ph,TimeUpdate);

       for (j=1; j<=6; j++)
            phm[j]=ph[j];

       tt+=dts;
   }

   /* kalman Gain: K=PH'inv(HPH'+R) */
   kg[0]=(ph[1]+ph[2]+ph[3])/25.;
   kg[1]=(ph[2]+ph[4]+ph[5])/25.;
   kg[2]=(ph[3]+ph[5]+ph[6])/25.;

   /* measurement update */
   tt=BSC_t;
   for(i=0; i<4; i++)
   {
       rk4m(xhm,m,tt,dts,xh,MeasUpdate);

       for (j=1; j<=3; j++)
            xhm[j]=xh[j];

       tt+=dts;

   }
       for (j=1; j<=3; j++)
            xhm[j]=xh[j];


   estimateT();
       for (j=1; j<=6; j++)
            phm[j]=ph[j];

 }

 /* end ekf.c */
```

```
/*
**      rk4m.c- lightly modified Runge-Kutta from NRC
**      written by: Sergey Belikov
**      revised by: David Hur
*/

#include "misc.h"
#include "global.h"

void rk4m(double y[], int n, double x, double h, double yout[],
          void (*derivs)(double, double [], double []))
{
        int i;
        double xh,hh,h6;
        double dydx[M+1],dym[M+1],dyt[M+1],yt[M+1];

        (*derivs)(x,y,dydx);

        hh=h*0.5;
        h6=h/6.0;
        xh=x+hh;

        for (i=1; i<=n; i++)
             yt[i] = y[i]+hh*dydx[i];

        (*derivs)(xh,yt,dyt);
        for (i=1; i<=n; i++)
             yt[i] = y[i]+hh*dyt[i];

        (*derivs)(xh,yt,dym);
        for (i=1; i<=n; i++) {
             yt[i] = y[i]+h*dym[i];
             dym[i] += dyt[i];
        }

        (*derivs)(x+h,yt,dyt);
        for (i=1; i<=n; i++)
             yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
}

void rk6m(double y[], int n, double x, double h, double yout[],
          void (*derivs)(double, double [], double []))
{
        int i;
        double xh,hh,h6;
        double dydx[6+1],dym[6+1],dyt[6+1],yt[6+1];

        (*derivs)(x,y,dydx);

        hh=h*0.5;
        h6=h/6.0;
        xh=x+hh;

        for (i=1; i<=n; i++)
             yt[i] = y[i]+hh*dydx[i];

        (*derivs)(xh,yt,dyt);
        for (i=1; i<=n; i++)
             yt[i] = y[i]+hh*dyt[i];

        (*derivs)(xh,yt,dym);
        for (i=1; i<=n; i++) {
             yt[i] = y[i]+h*dym[i];
             dym[i] += dyt[i];
        }

        (*derivs)(x+h,yt,dyt);
        for (i=1; i<=n; i++)
             yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
}

/* (C) Copr. 1986-92 Numerical Recipes Software n2'%9A,)+16. */
```

```
/*
**   volt.c -- voltages for the lamp rings
**   written by: Sergey Belikov
*/

#include "param.h"

        double volt1(double u) {
                double utab[6]={
                0.,0.19,0.31,0.39,0.73,1.
                };
                double vtab[6]={
                0.,1.,2.,3.,4.,5.
                };
                int i;
                double ret;

                for (i=1;u>utab[i];i++);
                ret=vtab[i-1]+(vtab[i]-vtab[i-1])
                   *(u-utab[i-1])/(utab[i]-utab[i-1]);
                return(ret);
        }


        double volt2(double u) {
                double utab[2]={
                0.,1.
                };
                double vtab[2]={
                0.,1.
                };
                int i;
                double ret;

                for (i=1;u>utab[i];i++);
                ret=vtab[i-1]+(vtab[i]-vtab[i-1])
                   *(u-utab[i-1])/(utab[i]-utab[i-1]);
                return(ret);
        }


        double volt3(double u) {
                double utab[6]={
                0.,0.046,0.29,0.53,0.80,1.
                };
                double vtab[6]={
                0.,1.,2.,3.,4.,5.
                };
                int i;
                double ret;

                for (i=1;u>utab[i];i++);
                ret=vtab[i-1]+(vtab[i]-vtab[i-1])
                   *(u-utab[i-1])/(utab[i]-utab[i-1]);
                return(ret);
        }

/* end volt.c */
```

```
/*
**    equforz.c -right part of ODE for Runge-Kutta
**    written by: Sergey Belikov
*/

#include "misc.h"
#include "equforz.h"
#include "global.h"

void equforz(double t, double *z, double *dzdt)
{
     double y[M+1];
     double Gu[M+1];
     int i,j;
     int m=M;

     for(i=1; i<=m; i++){
        y[i] = BSC_Ym1[i]+(BSC_Y[i]-BSC_Ym1[i])*t/BSC_dt;
        Gu[i]=0.;
        for (j=1; j<=m; j++)
        {
             Gu[i]+=(BSC_G[i][j]*pow(BSC_u[j],BSC_ga[j]));
        }
        Gu[i]*=(2./BSC_h);
      }

     dzdt[1] = -(BSC_al[1]/(BSC_RO*c(y[1])))
               *(-E(y[1])*y[1]*y[1]*y[1]*y[1]
               +BSC_al[1]*y[1]+z[1]+Gu[1]);

     dzdt[2] = -(BSC_al[2]/(BSC_RO*c(y[1])))
               *(-(BSC_A[2]*k(y[1])+4.*E(y[1])*y[1]*y[1]*y[1])*y[2]
               +BSC_al[2]*y[2]+z[2]+Gu[2]);

     dzdt[3] = -(BSC_al[3]/(BSC_RO*c(y[1])))
               *(-(BSC_A[3]*k(y[1])+4.*E(y[1])*y[1]*y[1]*y[1])*y[3]
               +BSC_al[3]*y[3]+z[3]+Gu[3]);
}


double c(double T)
{
      double Ttab[9]={
            0.,100.,200.,400.,600.,800.,1000.,1200.,3000.
            };
      double ctab[9]={
            256.,256.,549.,780.,856.,900.,934.,955.,955.
            };
      int i;
      double ret;

      for (i=1; T>Ttab[i]; i++);
      ret = ctab[i-1]+(ctab[i]-ctab[i-1])
         * (T-Ttab[i-1])/(Ttab[i]-Ttab[i-1]);
      return(ret);
}

double k(double T)
{
      double Ttab[9]={
            0.,100.,200.,400.,600.,800.,1000.,1200.,3000.
            };
      double ktab[9]={
            884.,884.,264.,98.9,61.3,42.2,31.2,25.7,25.7
            };
      int i;
      double ret;

      for (i=1; T>Ttab[i]; i++);
      ret = ktab[i-1]+(ktab[i]-ktab[i-1])
         * (T-Ttab[i-1])/(Ttab[i]-Ttab[i-1]);
      return(ret);
}


double E(double T)
```

```
{
        double Ttab[5]={
               0.,550.,700.,740.,3000.
               };
        double etab[5]={
               0.2,0.2,0.5,0.7,0.7
               };
        int i;
        double e;

        for (i=1; T>Ttab[i]; i++);
        e = etab[i-1]+(etab[i]-etab[i-1])
          * (T-Ttab[i-1])/(Ttab[i]-Ttab[i-1]);
        return(2.*e*BSC_Sigma/BSC_h);
}

/* end equforz.c */
```

APPENDIX C

Operational Procedure for RTP System

Initial Operation:

1. Turn on power for RTP system.
2. Make sure RTP (gas) controller switches are in OFF position.
3. Turn on Nitrogen:
   a. Turn on main tank valve.
   b. Turn on main valve to RTP.
   c. Adjust control valve to about 40~60 psi reading on the tank.

To Access Wafer:

Need: Cotton swab, Acetone, and Rubber glove

1. Make sure the side wafer access door is in UNLOCKED position.
2. On the RTP gas controller panel, turn on ***purge*** switch.
3. When the side access door pops open, chamber can be opened.

To change or shift the wafer, lower inside chamber using the crank at side of the RTP.
Before Closing the chamber:

a. Clean wafer and susceptor with acetone to prevent oxidation.
b. Apply vacuum grease around the rubber ring of chamber and side access door.
c. Close the chamber door and raise the chamber.
d. Make sure the thermo-couples are located within the notch of the O-ring of the chamber.

To Turn On RTP System:

1. Enable computer read-out of the RTP chamber temperature.
   a. Change to directory *notebook.*
   b. Run program by typing *nb.*
   c. Choose *setup* from menu.
   d. Select *save/recall* then select *recall.*
   e. Load either *tong* or *johnz1*.
   f. Press *esc*, then *iconview*, the *run*.
2. Make sure power connector for each lamp zone is plugged in.
3. **Turn on water**.
4. Turn Compressor on:
   a. Turn main switch on the compressor power panel.
   b. Turn on secondary switch.
   c. Let compressor run for few seconds.
   d. Open all the way the valve to RTP.
5. On vacuum controller panel position vacuum control switch to on (upper) position. (Max. pumping)
6. On gas controller panel, turn on *roughing valve.*
7. When the read-out is about 20, switch vacuum control switch to middle positon and set the dial to *auto*.

   *** During the experiment, keep read-out to about 70 psi.**
8. Set *set point* to about 70.

9. On gas controller panel, turn on *N2* switch. Adjust set point to keep the read-out near 70.
10. On gas controller panel, channel 1 corresponds to Nitrogen gas. Use this dial to set the gas control read-out to around 1.2.
11. When steady state of gas and vacuum is reached, RTP lamps are ready to turn on.

To Turn On the Adaptive Controller System:

1. Make sure the voltage supply for the switch box is plugged in.
2. Make sure the ribbon cable from the controller computer is plugged into the thermocouple signal conditioner terminal. Black is pin 1.
3. **Make sure all switches on the switch box is in OFF position.**
4. Turn on controller computer.
5. Change to appropriate directory where the control program resides.
   d:\david\cloop4 - original adaptive control.
   d:\david\cloop6 - emissivity experiment
   d:\david\cloop7 - adaptive control EKF
6. Run the control program. (type *test*)
7. When all LEDs on the switch box is OFF, turn all switches on the switch box to ON position.
8. Start control loop by clicking on the START button.

To Turn Off Adaptive Controller System:

1. **Turn all switches on the switch box to OFF position**
2. Clock EXIT button on user interface.

To Shutdown RTP System:

1. Turn off *N2* and *roughing valve* switches on the gas controller panel.
2. Close the valve on the compressor.
3. Turn off secondary switch, then main switch on the compressor power panel.
4. Close the main valve to RTP for nitrogen, then close main tank valve.
5. When thermocouple temperature reading reaches below 100C, turn off water.
6. Remove switch box power supply from the wall socket.
7. Turn off computer.
8. Turn off power to RTP.

BIBLIOGRAPHY

[1] Stephen A. Norman, "Wafer Temperature Control in Rapid Thermal Processing," Ph.D. Thesis, Stanford Univ., 1992.

[2] Y. M. Cho, A. Paulraj, T. Kailath, G. Xu, "A Contribution to Optimal Lamp Design in Rapid Thermal Processing," IEEE Trans. Semicond. Manufact., Vol 7, No 1, pp 34-41, Feb. 1994.

[3] S. Belikov, M. Kaplinsky, N. Ravindra, F. Tong, W. Kosonocky, "Wafer Temperature Measurement Correction for Multi-Wavelength Imaging Pyrometer Using Kalman Filtering," Proceedings, The 3rd International Rapid Thermal Processing Conference, Amsterdam, Netherlands, August 30-September 1 1995.

[4] S. Belikov, D. Hur, B. Friedland, "Real Time Estimation and Adaptive Control for RTP System", Proceedings, 3rd International Rapid Thermal Processing Conference, Amsterdam, Netherlands, August 30-September 1 1995.

[5] S. Belikov, M. Kaplinsky, B. Friedland, "Parameter Estimation for Evaluating Ability of a RTP System to Maintain Uniform Temperature", Proceedings, 4th IEEE Conference on Control Applications, Albany, NY, September 28-29 1995.

[6] S. Belikov, B. Friedland, "Closed-loop adaptive control for Rapid Thermal Processing", Proceedings, 34th IEEE Conference on Decision and Control, New Orleans, LA, December 1995.

[7] S. Belikov, "M-WIP Based RTP Controller," 1994. N.P, n.p.

[8] S. Belikov, H. Martynov, M. Kaplinsky, C. Manikopoulos, N. Ravindra, W. Kosonocky, "A Design Methodology for Configuration of Lamps in an RTP System", Proceedings, The 2nd International Rapis Thermal Processing Conference, Monterey, CA, August 31- September 2 1994.

[9] S. Belikov, H. Martynov, M. Kaplinsky, C. Manikopoulos, "On Using Wavelength Dependent Emissivity of Semiconductor to Model Heat Transfer in Rapid Thermal Processing Station", Proceedings, IEEE Transactions on Semiconductor Manufacturing, 1995.

[10] B. Friedland, "A Simple Non-Linear Observer for Estimating Parameters in Dynamic Systems", Proceedings, The 12th IFAC Congress, Sydney, Australia, July 18-23 1993.

[11] B. Friedland, *Control Systems Design, An Introduction to State Space Methods* McGraw Hill, New York, 1986

[12] L. C. Thomas, *Heat transfer-Professional Version,* PTR Prentice Hall, Englewood Cliffs, NJ, 1993.

[13] A. Ting, "Influence of Wafer-Dependent Radiation in Simulation of Lamp Heated RTP System", Proceedings, The 2nd International Rapid Thermal Processing Conference, Monterey, CA, August 31-September 2 1994.

[14] *Transputer Development and iq System Handbook,* Inmos, 1991

[15] *Transputer ANSI C toolset Manual, User manual,* Inmos, 1991

[16] *TMB16 Hardware Manual, User Manual,* Transtech , 1991

[17] *ADT108 A/D TRAM User's Manual,* Sunnyside, 1991

[18] *DAT208 D/A TRAM User's Manual,* Sunnyside, 1991

[19] S. Belikov, D. Hur. B. Friedland, N.M. Ravindra, "Estimation of emissivity of a wafer in an RTP chamber by a dynamic observer", Proceedings, in Rapid Thermal and Integrated Processing V (Mater. Res. Soc. Proc. 342, San Francisco, CA 1996)

[20] Frank L. Lewis, *Optimal Estimation with an Introduction to Stochastic Control Theory*, Wiley-Interscience, New York, 1986