1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**DESIGN OF PARALLEL SOFTWARE
FOR
ADAPTIVE MANIPULATOR CONTROL**

by

KENNETH EDWARD GRIGG, B.Sc.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of**

Master of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

January 4, 1991

© copyright 1991,

Kenneth Edward Grigg

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-68884-X

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**"DESIGN OF PARALLEL SOFTWARE
FOR
ADAPTIVE MANIPULATOR CONTROL"**
submitted by Kenneth Edward Grigg, B.Sc.
in partial fulfilment of the requirements for
the degree of Master of Engineering

Dr. C.M. Woodside, Thesis Supervisor

Dr. H.M. Schwartz, Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

January 4, 1991

Abstract

This research illustrates the use of a methodology and a toolset for the analysis and design of parallel robotic control software. The methodology includes decomposition of the control and estimation algorithms, analysis of the dataflow constraints, selection of the hardware architecture, and design of the software based upon iterative computation and communication performance predictions. The toolset includes the use of the Activity Sequence Network notation, dataflow graphs, timed Petri Nets, and Gantt Charts. The validity of the resulting design relies upon accurate estimation of sequential computation complexity and upon adequate modelling of the hardware constraints.

A linear second-order self-tuning regulator with indirect recursive least squares estimation was implemented on a VMEBus multiprocessor to provide adaptive control of the SCARA manipulator available at Carleton. The parallel implementation provided speedups close to those predicted using the toolset. Although a simulation of the manipulator was used, the resulting design is independent of this fact, and therefore forms the basis of a fast adaptive control testbed for future research. The implementation provides a modular structure allowing for simple module swapping to experiment with other indirect self-tuning regulator algorithms. With more substantial modifications, the testbed can support experimentation with other forms of adaptive control algorithms.

Acknowledgments

The compilation of this research has been achieved by the continued support and encouragement given to me from many sources. Most of all I have to acknowledge my wife and best friend, Suhayya Abu-Hakima, whose persistent assurances of my predestined success kept me on the right side of the fine line. Encouragement was also a staple diet at Carleton University, where my supervisors Murray Woodside and Howard Schwartz always ended with "keep up the good work" after setting me straight on yet another issue. I have to acknowledge Yves Robert and Serge Miguet for showing me that yes, I could actually do some publishable research (avec un accent américain), and Frank Dehne and Afonso Ferreira for leading me there. Finally, I must thank all of my previous managers and Jack Dymont at Bell-Northern Research both for providing such an excellent environment for my development as an engineer and for the encouragement and support to continue my education.

Others lending support or encouragement included the Systems Engineering support staff and other students at Carleton (Vivienne, Don, Cheryl, Erik, Mark, Anne-Marie, among others), and of course my family in Vancouver, Ottawa, and Montreal. To all of you I am very grateful!!

Table of Contents

Abstract.....	iii
Acknowledgments.....	iv
Table of Contents.....	v
List of Tables	viii
List of Figures	ix
1.0 Introduction.....	1
1.1 Motivations	1
1.2 Thesis Objectives	3
1.3 Thesis Contributions	5
1.4 Thesis Overview	6
2.0 Robot Adaptive Control.....	9
2.1 Characterization of the Carleton SCARA Robotic Manipulator	9
2.2 Control Algorithms Considered.....	14
2.2.1 Proportional Error Control	16
2.2.2 Self-Tuning Regulators.	18
2.2.3 Model-Referenced Adaptive Controllers.....	20
2.3 Testbed Controller Selection.....	22
2.3.1 PID Controller Details	26
2.3.2 Feedforward Control Detai	27
2.3.3 Parameter Estimation Details.....	29
2.3.4 Controller Parameter Selection.....	33
2.4 Manipulator Simulation	38
2.5 Trajectory Selection and Generation.....	39
3.0 Parallel Processing for Real-Time Control.....	44
3.1 Hardware Alternatives.....	45
3.2 Parallel Research on Manipulator Control Algorithms	50
3.2.1 The Inverse Dynamics Problem	51
3.2.2 Parallelism in Adaptive Control.....	51

3.3	Parallel and Distributed Software Design Tools.....	53
3.3.1	Software Decomposition Techniques	54
3.3.2	Activity Sequence Network Notation	55
3.3.3	Task-to-Processor Allocation and Performance Considerations... ..	58
3.3.4	Synchronization Techniques	61
3.4	Description of the Development and Target Environments	63
3.4.1	The VMEBus Hardware	63
3.4.2	The Unison Operating System.....	64
3.4.3	The Event Monitor	65
4.0	Design of Parallel Controller Software.....	68
4.1	Algorithm Decomposition.....	69
4.1.1	The Feedback and Feedforward Control Laws	69
4.1.2	The A/D Interface.....	71
4.1.3	The Estimator	72
4.2	Analysis of Dataflow Constraints.....	75
4.2.1	Dataflow Constraints.....	75
4.2.2	Sequential Software Design.....	79
4.2.3	Parallel Activity Sequence Network Design.....	82
4.3	Design by Performance Analysis	83
4.3.1	Architectural Constraints	84
4.3.2	Sequential Implementation and Analysis	86
4.3.3	Allocation of Activities to Multiple Processors	92
4.3.4	Exploitation of Inherent Parallelism.....	104
4.3.5	Predicted Speedup and Efficiency	112
4.4	Implementation Details	120
4.4.1	Design for Flexibility	120
4.4.2	Synchronization	121
4.4.3	Initialization	123
4.5	Summary of Parallel Design Process.....	126
5.0	Results of Parallel Implementation.....	131
5.1	Real-Time Performance	131
5.1.1	Comparison Between Predicted and Actual Results.....	132

5.1.2 Measured Speedup and Efficiency	137
5.2 Control Results.....	149
5.2.1 Parameter Convergence Observations	150
5.2.2 Trajectory Tracking Observations	154
5.3 Further Improvements.....	159
6.0 Conclusions	178
6.1 Summary of Contributions	179
6.2 Areas for Further Research	181
Glossary.....	184
Bibliography	186

List of Tables

Table 2.1 - Approximate Physical Characteristics of the Carleton Manipulator.....	36
Table 3.1 - Parallel Architecture Trade-Offs	47
Table 3.2 - Survey of Parallel Architectures for Robotic Control	49
Table 3.3 - Activity Sequence Network Notations	57
Table 3.4 - Petri Net Notation	60
Table 4.1 - Shortened Activity Names.....	80
Table 4.2 - VMEBus Access Delays	85
Table 4.3 - Slave Processor Memory Access Delays	85
Table 4.4a - Performance Predictions (in microseconds).....	90
Table 4.4b - Performance Predictions (in microseconds)	91
Table 4.5 - Matrix-Vector Product & Covariance Update Performance Predictions	91
Table 4.6 - Three Processor Activity Allocation.....	94
Table 4.7 - Extra VMEBus Activity for 3 Processor Allocation	95
Table 4.8 - Overall Predicted VMEBus Usage Time for 3 Processor Version.....	98
Table 4.9 - Six Processor Activity Allocation.....	109
Table 4.10 - Expected VMEBus Activity for 6 Processor Allocation	110
Table 5.1 - Bus Contention for Varying Utilizations [Mar83]	135
Table 5.2 - Estimated Memory Contention from VMEBus Utilization.....	135
Table 5.3a - Actual Performance (in microseconds)	141
Table 5.3b - Actual Performance (in microseconds)	142
Table 5.4 - Six Processor MVP & Cov Activity Actual Performance.....	142
Table 5.5 - Seconds to Parameter Convergence Error $< 1.0e-4$	154
Table 5.6 - Positional Tracking Accuracy Over Final 5 Seconds.....	158

List of Figures

Figure 2.1 - Open Kinematic Chain Manipulator.....	10
Figure 2.2 - Carleton Closed Kinematic Chain Manipulator	11
Figure 2.3 - PID Controller Structure.....	17
Figure 2.4 - Self-Tuning Regulator Structure.....	19
Figure 2.5 - Model-Referenced Adaptive Controller Structure	21
Figure 2.6 - Normalized STR Structure	23
Figure 2.7 - Normalized MRAC Structure	23
Figure 2.8 - Selected Testbed Controller Structure.....	25
Figure 2.9 - Trajectory Segment Timeline.....	41
Figure 2.10 - Link 1 Slow Trajectory.....	42
Figure 2.11 - Link 2 Slow Trajectory.....	42
Figure 2.12 - Link 1 Fast Trajectory.....	43
Figure 2.13 - Link 2 Fast Trajectory.....	43
Figure 4.1 - Dataflow Precedence Graph of Calculations	76
Figure 4.2 - Improved Dataflow Precedence Graph of Equations.....	78
Figure 4.3 - Sequential Activity Sequence Network	80
Figure 4.4 - Sequential Estimator Software Activity Sequence Network.....	81
Figure 4.5 - Parallel Activity Sequence Network.....	82
Figure 4.6 - Parallel Estimator ASN	83
Figure 4.7 - Sample Code Taken From Matrix-Vector Product Activity	88
Figure 4.8 - Three Processor Petri Net Model.....	100
Figure 4.9 - Three Processor Gantt Chart - Detailed.....	101
Figure 4.10 - Three Processor Gantt Chart - Detailed (continued).....	102
Figure 4.11 - Three Processor Gantt Chart - Overview	103
Figure 4.12 - Diagrammatic View of MVP Row Operations.....	106
Figure 4.13 - Diagrammatic View of Covariance Update Row Operations	107
Figure 4.14 - Synchronization of Inherent Parallelism	108
Figure 4.15 - Predicted Speedup with Upper Bounds.....	115

Figure 4.16 - Predicted Efficiency	115
Figure 4.17 - Six Processor Petri Net Model.....	116
Figure 4.18 - Six Processor Gantt Chart - Detailed	117
Figure 4.19 - Six Processor Gantt Chart - Detailed (continued).....	118
Figure 4.20 - Six Processor Gantt Chart - Overview	119
Figure 5.1 - Predicted Six Processor Activity Cycle.....	137
Figure 5.2 - Actual Six Processor Activity Cycle	137
Figure 5.3 - Measured Speedup with Upper Bounds.....	139
Figure 5.4 - Measured Efficiency	139
Figure 5.5 - Three Processor Gantt Chart (Detailed).....	143
Figure 5.6 - Three Processor Gantt Chart (Detailed) - continued.....	144
Figure 5.7 - Three Processor Gantt Chart (Overview).....	145
Figure 5.8 - Six Processor Gantt Chart (Detailed)	146
Figure 5.9 - Six Processor Gantt Chart (Detailed) - continued	147
Figure 5.10 - Six Processor Gantt Chart (Overview).....	148
Figure 5.11 - Link 1 Parameter a1 Convergence (Slow Trajectory)	162
Figure 5.12 - Link 1 Parameter a2 Convergence (Slow Trajectory)	162
Figure 5.13 - Link 1 Parameter b1 Convergence (Slow Trajectory)	163
Figure 5.14 - Link 1 Parameter b2 Convergence (Slow Trajectory)	163
Figure 5.15 - Link 2 Parameter a1 Convergence (Slow Trajectory)	164
Figure 5.16 - Link 2 Parameter a2 Convergence (Slow Trajectory)	164
Figure 5.17 - Link 2 Parameter b1 Convergence (Slow Trajectory)	165
Figure 5.18 - Link 2 Parameter b2 Convergence (Slow Trajectory)	165
Figure 5.19 - Link 1 Parameter a1 Convergence (Fast Trajectory).....	166
Figure 5.20 - Link 1 Parameter a2 Convergence (Fast Trajectory).....	166
Figure 5.21 - Link 1 Parameter b1 Convergence (Fast Trajectory)	167
Figure 5.22 - Link 1 Parameter b2 Convergence (Fast Trajectory)	167
Figure 5.23 - Link 2 Parameter a1 Convergence (Fast Trajectory).....	168
Figure 5.24 - Link 2 Parameter a2 Convergence (Fast Trajectory).....	168
Figure 5.25 - Link 2 Parameter b1 Convergence (Fast Trajectory)	169
Figure 5.26 - Link 2 Parameter b2 Convergence (Fast Trajectory)	169
Figure 5.27 - Link 1 Slow Trajectory and Single Processor Tracking.....	170

Figure 5.28 - Link 2 Slow Trajectory and Single Processor Tracking.....	170
Figure 5.29 - Link 1 Positional Tracking Error (Slow Trajectory).....	171
Figure 5.30 - Link 2 Positional Tracking Error (Slow Trajectory).....	171
Figure 5.31 - Link 1 Fast Trajectory and Single Processor Tracking.....	172
Figure 5.32 - Link 2 Fast Trajectory and Single Processor Tracking.....	172
Figure 5.33 - Link 1 Positional Tracking Error (Fast Trajectory)	173
Figure 5.34 - Link 2 Positional Tracking Error (Fast Trajectory)	173
Figure 5.35 - Link 1 Covariance Matrix Trace (Fast Trajectory)	174
Figure 5.36 - Link 2 Covariance Matrix Trace (Fast Trajectory)	174
Figure 5.37 - Link 1 Detailed Covariance Matrix Trace (Fast Trajectory).....	175
Figure 5.38 - Link 2 Detailed Covariance Matrix Trace (Fast Trajectory).....	175
Figure 5.39 - Link 1 Covariance Matrix Trace (Slow Trajectory).....	176
Figure 5.40 - Link 2 Covariance Matrix Trace (Slow Trajectory).....	176
Figure 5.41 - Link 1 Detailed Covariance Matrix Trace (Slow Trajectory).....	177

1.0 Introduction

The complex area of robotic control relies upon theory developed in the fields of mechanics, control systems, software engineering, and VLSI. Robotics became recognized as an area of study just after World War II, when the first robot was developed for industrial use by English inventor Cyril Walter Kenward. Since that time, much effort has brought our current technological abilities to the point where we have sent robotic tools to the surface of other worlds.

However, much has yet to be done. Current technology, although far advanced, still lacks the knowledge required to build robots that can emulate the dexterity of the human body. There are many reasons for this shortcoming, not the least of which is our inability to implement adequate low level control systems that can operate in anywhere near the real-time required to follow fast, intricate trajectories accurately. Two areas of research are providing techniques to help overcome this problem - adaptive control and parallel processing.

1.1 Motivations

Adaptive control is one area of research that may prove successful in solving the accuracy problems that arise due to the uncertainties present in measuring the physical characteristics of mechanical devices. Adaptive control is also used in systems where the plant is experiencing time-varying dynamic parameters. In some cases these methods can also be

used to simplify a nonlinear dynamic plant model into a linear system experiencing time-varying parameters. This area of control theory provides algorithms that use the measured dynamics and/or tracking errors of the system in real-time to continually improve on estimations of dynamic system parameters. These improved estimates are then continuously applied to improve the accuracy and stability of the control.

As might be expected, the algorithms involved in adaptive control may be computationally intensive, thus reducing the sampling rate achievable with any given computer technology over simpler control schemes. A lower sampling rate demands a lower control bandwidth which in turn results in poorer trajectory following. However, the field of parallel processing, which looks at converting sequential algorithms to operate on multiple cooperating processors, is providing methods whereby the extreme computational requirements can be met in real-time.

Distributed processing methods, which deal with the ability to break up system tasks among multiple networked processors, may also be applied to the robotics problem. Complex control schemes that involve several different sequential or parallel activities can make use of distributed processing techniques to share the workload amongst several processors, provided the interconnecting network provides adequate bandwidth for the interprocessor communication.

The work presented within this thesis synthesizes these fields to provide an example of how to utilize the new computing techniques and technology in the area of low level adaptive control of a robotic manipulator.

1.2 Thesis Objectives

Although there has been a significant amount of research into the application of parallel processing to some areas of robotic control, very little effort has been spent specifically on the parallelization of adaptive control. This work breaches the subject, with a specific goal of providing a parallel testbed for the simplification of future prototyping and testing of adaptive control algorithms for the robotic manipulator available at Carleton University. Previous experimentation with the manipulator has relied upon an IBM-PC/AT to run the controlling algorithms. The insufficient computing performance of the personal computer has provided further motivation to parallelize the software.

Throughput is important in control systems development since required sampling rates may be very high to achieve a desired control quality. For example, large disturbances will have a shorter effect on the process given a shorter sampling period [Ast89]. Ideally, controllers would run fast enough to allow direct implementation of the continuous time equations of dynamics and control. When this is not the case, the equations are discretized for a specific sampling rate, essentially approximating the continuous system using discrete step functions. The example developed within this thesis illustrates the effect of sampling rate on a discrete model controller, but the techniques are equally applicable to a continuous model control system.

Note that the opportunity for higher control bandwidth under higher sampling rates is not exploited in this analysis. Although this is obviously a key advantage in improving control quality, this analysis illustrates only the improvement due solely to increased sample rates, not system parameter changes. Furthermore, this problem was approached from the viewpoint of providing identical functionality in minimum time. Thus pipelined and other multiple rate implementations, although discussed, were not implemented.

Designing parallel software for a real-time system is a complex activity, and there are many methodologies that one can follow (e.g. Communicating State Machine or Finite State Machine analysis). Control systems form a specific subset of real-time systems, requiring very hard deadlines that may prove catastrophic if missed. There are several methods that have been developed to deal with these problems, such as Real-Time Temporal Logic [Ost87, Fau88]. This thesis experiments with the design of such software using the Activity Sequence Network notation [Woo88a] combined with the software and performance analysis methods provided by Generalized Stochastic Petri Nets [Pet77, Chi87].

Note that this thesis approaches software development from an application design viewpoint, with a goal of illustrating software design using performance predictions, rather than exploring a specific software methodology. However, the objective of a flexible controller testbed is in line with the philosophy purported by the Rapid Prototyping methodology [Bru89].

Beyond the questions of control system quality and software design methods, one can question whether or not there is an intrinsic structure of low-level robotic control software and an ideal hardware platform for implementation. This work investigates adaptive control software for parallelism, and discusses the use of the structure in choosing a suitable hardware architecture. A VMEbus multiprocessor based on Motorola 68020 processors with floating point coprocessors forms the platform for the testbed implementation, and provides an example of a bus-based platform.

Finally, note that this research involves the lowest level of manipulator control, and tends to avoid reference to higher level mechanisms such as path planning, end-effector positioning, collision avoidance, etc. However, there has been a lot of work done in considering how an overall manipulator controller should function. As an example, one can refer to [Elg88], where they discuss a high-level robot controller implemented on the Harmony [Gen89] real-time operating system which interfaces to a PUMA manipulator at the servo-control level.

1.3 Thesis Contributions

The major contributions of this thesis include the following:

- illustrates the use of Activity Sequence Network notation in the design of real-time parallel software;

- illustrates the use of Petri Net analysis techniques for predicting performance of hard deadline, deterministic real-time robotic control software;
- analyses the differences between the predicted and the actual performance of a multiprocessor implementation of an adaptive control algorithms; and
- provides a flexible, high performance testbed for evaluation and testing of fast adaptive manipulator control algorithms.

1.4 Thesis Overview

This thesis describes an experiment in the use of parallel and distributed processing techniques for implementing high speed and high accuracy low-level adaptive control algorithms for robot manipulators. The basic issues involved in developing adaptive controller software are discussed and an understanding of the problem is developed.

The goals of the thesis were met by judicious use of three engineering disciplines: control theory, software performance engineering, and parallel/distributed processing methodologies. Initially, an understanding of the background control theory and statistical approximation theory was developed. This included research into the effects of trajectory on parameter adaptation and subsequent implementation of a trajectory generation tool. Software engineering techniques were used to ensure modularity and flexibility of the resulting code, and to ensure that the prototype developed would be evolvable beyond the simulation phase into actual experimentation with the Carleton robot. Furthermore, performance engineering techniques were employed to aid in optimization of the software

design using predicted performance measures. Parallel and distributed processing techniques were investigated to provide an insight into the communication and synchronization characteristics of parallel hardware.

After laying out the goals, development of the thesis proceeded by analysis of the algorithm and subsequent decomposition into activities. A dataflow analysis was carried out to provide a precedence graph used to ensure correctness of a sequential implementation and to explore the potential parallelism available in the algorithm through development of an Activity Sequence Network. The multiprocessor system selected for implementation was analysed for expected interprocessor communication penalties, and this information was combined with the sequence network and the sequential simulation performance to develop an implementable three processor restricted parallelism version based on functional parallelism (i.e. parallelism at the activity, or coarse-grained level). This restricted parallelism model was then simulated using a deterministic Petri Net tool, and another tool was developed to convert the Petri Net results into Gantt Chart format for easier analysis. The resulting analysis provided a clear view of the critical path within the parallel set of sequential processes. The activities within the critical path were then analysed for inherent parallelism, resulting in a six processor parallel version. Both versions were analysed for expected performance on the target system by taking into account bus communication penalties and contention.

The uniprocessor, three processor, and six processor versions were then implemented and tested on the target hardware. The software was modified to utilize an event monitor to provide real-time performance figures and to trigger a logic analyser for obtaining bus

utilization measurements. These were then compared against the predicted values, and discrepancies were discussed. The control characteristics were then captured and plotted, showing the resulting control improvement attained through the use of multiple processors.

Included as supporting material are a short survey on current research into parallelism in robotic control algorithms and a comparison of the performance improvements obtained by different researchers using multiprocessor techniques for robotic control problems.

Chapter 2 provides a background in the area of adaptive robot control, discusses the mechanics of the Carleton manipulator, develops the adaptive control algorithm that is used within the thesis, and presents the two example trajectories. Chapter 3 discusses the previous work in the application of parallel/distributed processing techniques in the area of robotic control systems, and introduces the parallel hardware and supporting software used for this research. Chapter 4 introduces the design tools and methods used and details the development of the multiprocessor implementation. The experimental results are discussed in chapter 5, investigating the actual speedup obtained and the controller performance. Finally, chapter 6 provides a summary of the thesis and conclusions.

2.0 Robot Adaptive Control

When designing a controller for a robotic manipulator, one must consider the following:

- the mechanical design;
- how this design is to be modelled;
- the ability to determine the model parameters;
- the type of trajectory to be followed;
- the type of control algorithm to implement; and
- the quality of control desired.

This chapter discusses these considerations for the Carleton manipulator in light of the well-established field of control theory. The manipulator is first characterized dynamically and then the control algorithms under consideration are described. This is followed by a detailed description of the control algorithm selected for implementation, along with the justification for the choice. The algorithm used was proposed by Professor Schwartz as a general vehicle for exploratory research.

2.1 Characterization of the Carleton SCARA Robotic Manipulator

Traditionally, robotic manipulators have been designed in a serial link fashion, such as the PUMA models offered by Unimation Inc [Uni]. The links of a serial manipulator like the PUMA arm form an open kinematic chain (see Figure 2.1). The joints are driven by

motors whose torques are transmitted and amplified by gear trains. These articulated arms are dependent upon the quality of the gears for their mechanical stiffness, and thus are limited in the speed and accuracy with which they can follow a given trajectory.

Figure 2.1 - Open Kinematic Chain Manipulator

Direct drive manipulators are designed to provide high accuracy and speed by coupling the motors directly to the links without the use of gears [Asa87]. They eliminate the mechanical inaccuracies common to serial arms, providing improved dynamic response which in turn allows for accurate movement and pure torque control. The direct drive mechanics also allow for large accelerations - over 5G acceleration with velocities over 10 m/sec. The Carleton manipulator is an example of a two degree-of-freedom direct drive manipulator, and uses a SCARA (Selective Compliance Assembly Robotic Arm) architecture [Raj89] (see Figure 2.2). The robot has a horizontal parallelogram link structure where the link motion is constrained to a horizontal plane. The motors are built

on the central vertical axis and are not subject to gravitational load. Furthermore, the SCARA design has the potential to offer another very attractive control characteristic - dynamically decoupled links.

Figure 2.2 - Carleton Closed Kinematic Chain Manipulator

All robotic manipulators can be modelled with the following matrix equations of dynamics [Sch90]:

$$U = M(y)\ddot{y} + C(y,\dot{y})\dot{y} + G(y) \quad (2.1)$$

where,

U is an $n \times 1$ vector of joint torques

$M(y)$ is the $n \times n$ mass matrix representing the manipulator inertia

y is the $n \times 1$ vector of joint positions

$C(y,\dot{y})$ is the $n \times n$ matrix of Coriolis, centrifugal, and friction terms

$G(y)$ is the $n \times 1$ vector of gravity terms

The equations of motion of the Carleton arm have been derived (in [Raj89₁]) using Lagrangian dynamics. These were adapted from the equations of motion for a five-bar structure given in [Riv88], from which the gravitational forces have been removed since the Carleton arm operates solely in the horizontal plane. The friction effects have been assumed negligible. Dropping the gravity term from (2.1), we obtain:

$$U = M(y)\ddot{y} + C(y, \dot{y})\dot{y} \quad (2.2)$$

The matrix C can also be simplified with respect to an open-loop (serial) manipulator due to the particular closed-loop structure of the Carleton manipulator. Specifically, the Coriolis terms (functions of $\dot{y}_1\dot{y}_2$) are absent and the number of nonlinear terms is reduced. Neglecting friction effects as negligible, equation (2.2) can be written as [Raj89]:

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix} \begin{bmatrix} \ddot{y}_1 \\ \ddot{y}_2 \end{bmatrix} + \begin{bmatrix} 0 & H_{122} \\ H_{211} & 0 \end{bmatrix} \begin{bmatrix} \dot{y}_1^2 \\ \dot{y}_2^2 \end{bmatrix} \quad (2.3)$$

where,

$$H_{11} = m_1 l_{1c}^2 + I_1 + m_3 l_{3c}^2 + I_3 + m_4 l_1^2 + m_0 l_1^2$$

$$H_{22} = m_2 l_{2c}^2 + I_2 + m_3 l_2^2 + m_4 l_{4c}^2 + I_4 + m_0 l_{4c}^2 + I_0$$

$$H_{12} = H_{21} = m_3 l_2 l_{3c} - m_4 l_1 l_{4c} - m_0 l_1 l_{4c} \cos(y_1 - y_2)$$

$$H_{122} = H_{211} = m_3 l_2 l_{3c} - m_4 l_1 l_{4c} - m_0 l_1 l_{4c} \sin(y_1 - y_2)$$

and,

u_i is the corresponding torque for the i^{th} link

m_i mass of the i^{th} link (link 0 represents the payload)

l_i length of the i^{th} link

I_i moment of inertia of the i^{th} link

Furthermore, the links may be modelled as decoupled links (i.e. $H_{12} = H_{21} = 0$) if there is no payload ($m_0 = 0$) and the following condition holds:

$$m_3 l_2 l_{3c} - m_4 l_1 l_{4c} = 0$$

This condition also removes the centrifugal forces (i.e. $H_{122} = H_{211} = 0$) and the equations can then be written:

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} H_{11} & 0 \\ 0 & H_{22} \end{bmatrix} \begin{bmatrix} \ddot{y}_1 \\ \ddot{y}_2 \end{bmatrix} \quad (2.4)$$

This of course greatly reduces the computation required to calculate the inverse dynamics of the manipulator, allowing for very fast control. This simplified model was used to develop the simulations discussed in this thesis. Thus the algorithm development which follows depends specifically on a decoupled link model. If the links were not to be modelled as decoupled due to the presence of a payload or to non-ideal link lengths or masses, equation (2.3) would be used, requiring a nonlinear control algorithm involving substantially higher computation complexity.

As for output, the Carleton manipulator is equipped with an angular position transducer (or resolver) coupled directly to each of the manipulator's servomotor shafts. These resolvers provide digital outputs indicating the angular positions of their associated links. The manipulator is also equipped with angular velocity transducers (or tachometers) that provide output voltages proportional to the angular velocity of the associated links. Both of the transducers are connected to an analog-to-digital (A/D) converter on the controlling computer system. The digital position signals consist of 12-bits of resolution and are directly transmitted to the controlling computer. The analog velocity signals are converted to 12-bit digital signals before transmission to the controller.

2.2 Control Algorithms Considered

The objective of a control algorithm is to ensure that the manipulator follows a specified, "reasonable" trajectory within a small but acceptable error boundary. In this context, "reasonable" implies that the trajectory must be smooth enough to be continuously differentiable up to the second order with respect to time. This permits dynamic models such as that in equation (2.1) to be solved for the torque required at any instant along the trajectory, given the desired position, velocity, and acceleration.

Use of a dynamic model alone constitutes feedforward control, and the control algorithm is then essentially just an inverse dynamics computation. This form of control is not satisfactory under most practical circumstances, since it requires exact knowledge of the dynamic properties of the manipulator and does not account for disturbances, such as

noise, numerical roundoff errors, and approximations inherent in the discretization of a continuous system.

Feedback control methods are used to overcome the problems associated with feedforward control. Research in the area of feedback control is extremely broad and active, judging from the literature available. The main areas of research are robust control, optimal control, intelligent control, compliant motion control, and adaptive control. Robust controllers use constant feedback gains chosen to maintain "good" control characteristics despite changes in the system parameters due to inherent nonlinearities [Cra86, Ast89]. Optimal control researchers attempt to develop stable controllers that minimize some cost function, such as the control effort and/or the tracking error [Kai80]. Intelligent control refers to the methods developed from artificial intelligence research. Such systems typically use the theory of neural networks to create control systems that learn over time. Currently these methods are very immature and do not yet provide acceptable manipulator control [Mey87]. Compliant motion control methods depend upon the environment of the manipulator, where the controller receives feedback regarding the force, or virtual force, it is experiencing. Force control methods solve the problem of inaccurate modelling of the manipulator environment, and is usually applied to the tracking of surfaces and to tool insertion problems [Asa86, Nec89].

The field of adaptive feedback control as applied to robotic manipulators is relatively new. The main purpose of adaptive controllers is to attempt to provide continuously improving stable control. These controllers use system input and tracking errors to adjust the control parameters to improve the control quality. There are two main classes of adaptive

controller: the self-tuning regulator which uses estimation techniques to identify the actual plant dynamics which are then used to improve the regulator; and the model-referenced adaptive controller which uses tracking errors to adjust the regulator in an attempt to make the plant conform to a given model.

The following subsections discuss both self-tuning control and model-referenced adaptive control. However, a short treatment of the simplest form of feedback control, proportional error control, is given next since it provides an ideal mechanism for providing control before the adaptive algorithms can stabilize, and is a simple way to adjust for the errors inherent in feedforward control.

2.2.1 Proportional Error Control

Proportional controllers simply apply control in proportion to the tracking error, in effect ignoring the plant dynamics in calculating the applied control. In the case of manipulators, tracking errors are usually measured by sampling the position and velocity and comparing these to the reference trajectory. Figure 2.3 shows a block diagram of a PID controller (proportional, integral, derivative) for a robotic manipulator.

Figure 2.3 - PID Controller Structure

The controller equations are as follows (see [Ast89], [Fre71]):

$$u(t) = K_p \left\{ e(t) + \frac{1}{T_i} \int_0^t e(s) ds + T_d \dot{e}(t) \right\} \quad (2.5)$$

where,

$u(t)$ is the torque applied at time t

K_p is the proportional feedback gain on the position error

$e(t)$ is the position error at time t , so $e(t) = y_d(t) - y(t)$

T_i is the integration time constant

T_d is the anticipation period of the controller

Proportional control alone provides the main correcting factor of the controller, attempting to drive the position tracking error to zero. By itself however, it leads to oscillations about the desired trajectory, so the derivative control provides a dampening effect. The integration term takes care of any steady state errors.

The main design problem when using such a controller is selecting the gain and time constants. Selection of the gains completely determines how the controller will function, since this involves placement of the transfer function poles in the s-plane. There are many methods for selecting gains (including adaptive methods, see [Veg89]), but normally the model of the manipulator dynamics is combined with the PID equations to obtain an error equation which can then be analysed for gains providing critical damping.

In many cases, particularly when extreme accuracy and speed is not required, PID controllers are desirable since the computation required at each cycle is very simple. They provide robust stability [Asa86] but tend to overshoot if underdamped and are sluggish if overdamped or critically damped. Stability of proportional controllers is well known [Asa86], and as such they are a popular choice in control system design.

2.2.2 Self-Tuning Regulators

Self-tuning regulators (STRs) are useful when the manipulator can be modelled as a linear discrete-time system whose parameters are unknown, inaccurately known, or are slowly changing. The scheme works by continuously updating estimates of the process, or plant, parameters which are in turn used to update the regulator parameters through a solution of a design problem [Ast89]. The basic block diagram is shown in Figure 2.4.

Figure 2.4 - Self-Tuning Regulator Structure

When designing a STR, a control law is first designed under the assumption that the linear model of the manipulator is known (this is the *certainty equivalence principle*), then the parameters of the model are estimated on-line [Hsi86]. There are many possibilities in the choice of the underlying design problem. Koivo [Koi83], for example, chooses an optimal controller design that minimizes the sum of squares of the weighted tracking error and the applied control torque. Åström and Wittenmark [Ast89], for another example, show the use of pole placement in the design problem. Hsia [Hsi86] discusses pole placement design and velocity control design.

There are also many choices of the parameter estimation algorithm. Stochastic approximation, various forms of least squares estimation, instrumental variables, and maximum likelihood are usual choices (see [Ast71]). Convergence of the parameters is an

important issue in these controllers since the stability analysis of the control law is based on the *certainty equivalence principle*. Convergence relies basically on two points: the structure of the estimated model must be appropriate; and the input signal must sufficiently excite the plant [Ast89].

2.2.3 Model-Referenced Adaptive Controllers

A model-referenced adaptive controller provides a designer-specified model that the process is to follow. The controller parameters are then adjusted by a mechanism depending upon the error between the model output and the process output as both follow a specified trajectory. The basic structure of the controller is provided in Figure 2.5.

Figure 2.5 - Model-Referenced Adaptive Controller Structure

Normally, MRAC regulators are designed by equating the model transfer function with the closed loop regulator-process transfer function and solving the resulting Bezout (or Diophantine) equation under the assumption that certain associated compatibility conditions hold (see [Ast89]). Under the assumption that the plant parameters are known perfectly, this method can provide perfect model following. This method requires that the process to be controlled can be modelled by a linear time-invariant system, making it unsuitable for the manipulator control problem. However, some research has been done into the design of MRAC regulators for manipulators (see [Cra88] and [Hsi86]), under the assumption that the non-linear dynamics can be modelled as a slowly time-varying second order linear system, or assuming decoupled linear link models.

The second part of the design problem is to provide an adaptation mechanism for changing the regulator gains to accommodate unknowns in the plant parameters. Essentially the gain adaptation must adjust the regulator to keep the tracking error between the model and the process output close to zero. One much published mechanism is called the MIT Rule, or the Gradient Approach [Ast89], where a cost function based on the square of the error between the model and the plant output is minimized. This adapts the parameters in the direction that will decrease these errors. This method is not always stable, however, since the stability actually depends upon the magnitude of the desired trajectory. Preferred methods are based on Lyapunov stability analysis, where the resulting adaptive laws provide uniform stability (see [Ast89], [Nar89a]). As in the case of self-tuning regulators, convergence of the parameters depends upon the persistent excitation of the adaptive algorithm by the input trajectory.

2.3 Testbed Controller Selection

The STR and the MRAC approaches are actually closely related as can be seen from the Figures 2.4 and 2.5. The structures are similar, where both include a feedback loop from process to regulator and both include some adaptation mechanism. The essential difference between the STR and MRAC methods is that the STR method adapts the regulator parameters based on the prediction errors, whereas the MRAC uses the tracking errors directly to accomplish the same thing [Ast89]. Figures 2.6 and 2.7 are re-

arrangements of Figures 2.4 and 2.5, respectively, and show the similarity in the structures of the STR and the MRAC-type controllers.

Figure 2.6 - Normalized STR Structure

Figure 2.7 - Normalized MRAC Structure

The STR approach is used for this research because it allows for a straightforward derivation of a simple control law, and the estimation mechanism displays an interesting

parallelism which is exploited in following sections. The structural similarity, however, illustrates that the example STR chosen does not limit the flexibility of this research and its future application to other control algorithms.

Before discussing the STR algorithm, the nature of adaptive control and its use in embedded systems must be further explored. The problem with purely adaptive models is that the parameter estimates are inaccurate for some periods of time due to initial convergence delay. This may be compensated for by "training" the controller, for example by running the manipulator through the trajectory several times under some type of proportional control. Samples taken from these training runs can be used to estimate an initial parameter set for the adaptive controller. Estimates are then repeatedly updated with the new information and the adaptive controller is only used after parameter convergence. Obviously this method is useful only if the trajectory is repeatable and if the parameters do not vary over the trajectory, which can only be assumed in relatively closed circumstances, such as an assembly-line setting.

An alternative would be to provide a proportional controller that runs in parallel with the adaptive controller. This approach is taken in this research, where a PID controller works alone until the parameter estimates have converged to reasonable values. At that point the adaptive control output is added to the PID output to provide the controlling torque. The parameter convergence is monitored continuously and the STR is removed upon any sign of the parameter values diverging. This method may be used with the model of the manipulator discussed above since the parameters are constant. This method would be equally applicable in a system where the parameters are slowly time-varying.

The control algorithm used as an example in this research is therefore based on the combination of a PID controller and a self-tuning feedforward controller. The PID controller defines the system error dynamics, while the feedforward control provides the torque to track the desired trajectory. The torque output of both controllers is summed, and if the feedforward controller is providing accurate tracking, then the effect of the PID controller should be minimal. Figure 2.8 illustrates the continuous domain form of the controller structure.

Figure 2.8 - Selected Testbed Controller Structure

Note that since each link is assumed to follow the uncoupled linear dynamics shown in equation (2.4), the control for the two links is completely separated, so this controller can

be viewed as a single-input, single output (SISO) system which is implemented once for each link.

2.3.1 PID Controller Details

The PID controller is implemented using the discrete form of the continuous equation provided in equation (2.5), which can be rewritten in the form:

$$u(t) = K_p \left\{ e_p(t) + \frac{1}{T_i} \int_0^t e_p(s) ds + T_d \dot{e}_v(t) \right\} \quad (2.6)$$

where,

$e_p(t)$ is the position tracking error at time t

$\dot{e}_v(t)$ is the velocity tracking error at time t

There are many methods available to derive the discrete form of equation (2.6), of which sampled data theory Laplace transform methods are the most accurate. However, since the PID regulator is so simple, difference equation approximations such as discussed in [Mon85] prove adequate. Following those methods, $u(t)$ above is differentiated to obtain the velocity form,

$$\dot{u}(t) = K_p \dot{e}_p(t) + \frac{K_p}{T_i} e_p(t) + K_p T_d \ddot{e}_v(t) \quad (2.7)$$

When sampling with period T and by ignoring second order and higher terms in the Taylor series expansions of the time derivatives, equation (2.7) can be approximated by the following difference equation:

$$\frac{u(k) - u(k-1)}{T} = K_p * \frac{e_p(k) - e_p(k-1)}{T} + \frac{K_p}{T_i} * e_p(k) + K_p * T_d * \frac{e_v(k) - e_v(k-1)}{T} \quad (2.8)$$

where,

k is the time step

T is the sampling period

Equation (2.8) may then be reduced to the following:

$$u(k) = u(k-1) + K_p * \left\{ \left(1 + \frac{T}{T_i} \right) * e_p(k) - e_p(k-1) + T_d * [e_v(k) - e_v(k-1)] \right\} \quad (2.9)$$

This is the discrete form of the PID controller that was implemented, using sensor output data from the manipulator providing $e_p(k)$ and $e_v(k)$ for each time step k .

2.3.2 Feedforward Control Details

Starting from the individual link form of the decoupled equations (2.4) for the Carleton manipulator,

$$u_i = H_{ii} * \ddot{y}_i, \quad \text{or} \quad \ddot{y}_i = \frac{u_i}{H_{ii}} \quad (2.10)$$

and choosing per-link state variables as y_i and $\dot{y}_i$, we can write this second order system in state-space form:

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{H} \end{bmatrix} * u \quad (2.11)$$

where the subscripted 'i's are dropped for clarity.

Using standard z-transform methods with a sampling period of T , this system can be discretized into the following discrete linear difference equation:

$$y(k) = 2.0y(k-1) - y(k-2) + \frac{T^2}{2H}u(k-1) + \frac{T^2}{2H}u(k-2) \quad (2.12)$$

This discrete system can be used to derive a minimum-variance-type control scheme, where the controller attempts to drive the output to the desired output in one time step. If equation (2.12) is solved for $u(k-1)$ we obtain:

$$u(k-1) = \frac{2H}{T^2} * \left[y(k) - 2.0*y(k-1) + y(k-2) - \frac{T^2}{2H} * u(k-2) \right] \quad (2.13)$$

We then shift the time variable one step in the future and substitute the desired trajectory for the position values to obtain the control law:

$$u(k) = \frac{2H}{T^2} * \left[y_d(k+1) - 2.0*y_d(k) + y_d(k-1) - \frac{T^2}{2H} * u(k-1) \right] \quad (2.14)$$

where $y_d(k)$ is the desired joint angular position at time k .

This is the feedforward control law that will be used on a per link basis in the example control system, except for the fact that the position and torque coefficients will be estimated. Stability of such minimum-variance-type controllers is well-known [Ast73], thus under the *certainty equivalence principle* assumption for convergence of the parameters, stability of this controller is ensured.

2.3.3 Parameter Estimation Details

The method of recursive least squares estimation (see [Ast89], [Ast71], [Ise82], [Koi83], etc.) was chosen for the estimator since it is a well-known and robust method. This method minimizes the cost function:

$$J_N(\theta) = \frac{1}{2N} \sum_{k=1}^N \left\{ \lambda^{N-k} [y(k) - \varphi^T(k)\theta]^2 \right\} \quad (2.15)$$

where,

- N is the number of samples;
- J_N is the cost function;
- θ is the parameter vector;
- λ is the forgetting factor; and
- φ is the regression vector.

This cost function represents the mean square of the parameter tracking error, as is described below. The forgetting factor, λ ($0 < \lambda \leq 1$), is used to reduce the effects of older data if the parameters are slowly time-varying. The forgetting factor adjusts the rate of parameter convergence since lower values discard older data faster. There are many other estimation schemes based on the recursive least squares mechanism, modified for several specific conditions. For a survey see [Sha88].

Using the recursive least squares estimation, we replace the model parameters on a per link basis with their estimates as follows:

$$y(k) = \hat{a}_1 y(k-1) + \hat{a}_2 y(k-2) + \hat{b}_1 u(k-1) + \hat{b}_2 u(k-2) + e(k)$$

$$\text{or } y(k) = \hat{\theta}^T \varphi(k-1) + e(k) \quad (2.16)$$

where,

$$\hat{\theta}^T = [\hat{a}_1 \quad \hat{a}_2 \quad \hat{b}_1 \quad \hat{b}_2] \text{ is the vector of parameter estimates, and:}$$

$\hat{a}_1$ should converge to 2.0;

$\hat{a}_2$ should converge to -1.0;

$\hat{b}_1$ should converge to $\frac{T^2}{2H}$;

$\hat{b}_2$ should converge to $\frac{T^2}{2H}$.

$$\varphi^T(k) = [y(k-1) \quad y(k-2) \quad u(k-1) \quad u(k-2)] \text{ is the regression vector}$$

$e(k)$ is the zero-mean white noise tracking error due to parameter inaccuracies.

Note that the actual parameters a_1 and a_2 could be used instead of their respective estimates since these parameters are known and do not depend upon the physical parameters of the manipulator. However, the estimates were used in the experiments to generalize the results.

Each link's parameters are then estimated using the following recursive equations:

$$\text{Update of the covariance matrix: } P(k) = \frac{1}{\lambda} \left[P(k-1) - \frac{P(k-1)\varphi(k)\varphi^T(k)P(k-1)}{\lambda + \varphi^T(k)P(k-1)\varphi(k)} \right] \quad (2.17)$$

$$\text{Update of the parameters: } \hat{\theta}(k) = \hat{\theta}(k-1) + P(k)\varphi(k)[y(k) - \hat{\theta}^T(k-1)\varphi(k)] \quad (2.18)$$

For implementation, equation (2.17) can be rewritten into the following form:

$$P(k) = \frac{1}{\lambda} \left[P(k-1) - \frac{K(k)K^T(k)}{\lambda + \alpha(k)} \right] \quad (2.19)$$

where,

$$K(k) = P(k-1) * \varphi(k)$$

$$\alpha(k) = K^T * \varphi(k)$$

Then equation (2.18) can then be rewritten into the form:

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{K(k)}{\lambda + \alpha(k)} [y(k) - \hat{\theta}^T(k-1)\varphi(k)] \quad (2.20)$$

These forms reduce the complexity of calculating the new controller parameters at each cycle.

It should be noted that the method of recursive least squares estimation may become numerically unstable in some circumstances. The variation in magnitude of the covariance matrix $P(k)$ may cause numerical instability due to digital roundoff errors in the repeated subtraction of equation (2.19) since this matrix must remain positive definite and symmetric. When necessary, alternative algorithms known as the square root, or Cholesky factorization, methods can be implemented to avoid this problem (see [Ast89], [Bie76], [Bie77], [Gen73]). Although still of $O(n^2)$ time complexity as the equations above, these square root methods are substantially more computationally intensive as well as being less intuitive. They separate the covariance matrix into an upper/lower triangular matrix and a vector of the diagonal elements. The update of equation (2.19) is then applied using numerically stable matrix transformations. Besides providing numerical stability, the use of a triangular matrix also precludes the chance of loss of symmetry. The use of these methods was not considered worthwhile for this study since double precision, floating point number representation was available and the trajectory were of short durations.

Another issue that must be answered is that of determining when the parameters have converged. This issue has apparently not formally been addressed in the literature, except for being discussed as a transient condition. Unfortunately, this transient condition can actually cause instability in the system. Two methods are used in this research to determine when the parameters have converged: limiting the cycle-to-cycle change in the parameter

magnitudes and checking the "whiteness" of the prediction error (i.e. of $y(k) - \hat{\theta}^T(k-1)\phi(k)$). The magnitude of the covariance, or "information", matrix may also be used to detect convergence by setting expected convergence bounds on the matrix trace.

A final issue to study is the convergence of the parameters. It was noted in Section 2.3.2 that under the condition that the parameter estimates are exact (the *certainty equivalence principle*), the control law is stable. Convergence of the parameters, required to satisfy the principle, is ensured by choosing persistently exciting trajectories (see [Ast89], [Cra88], [Sch89], [Sch90]). Persistent excitation essentially requires that the input signal, or trajectory, provide enough information to excite the dynamic modes of the system such that the parameters can be identified. Trajectories can be chosen to be persistently exciting, in fact the research reported by Armstrong [Arm89] illustrates how to optimize trajectory selection for persistency of excitation. The use of non-persistently-exciting trajectories may also result in parameter convergence due to the information provided by the manipulator tracking error, but convergence is much slower and not guaranteed in the presence of noise [Sch90].

2.3.4 Controller Parameter Selection

Having defined the controller equations, this section discusses the selection of the gains of the PID controller and of the parameters associated with the estimator.

When selecting the gains of the PID part of the controller, the entire system must be considered, including the feedforward controller and the manipulator. From equations

(2.6) and (2.10), we can write the complete per-link feedback plus feedforward control system in continuous form as:

$$u(t) = K_p * \left\{ e_p(t) + \frac{1}{T_i} * \int_0^t e_p(s) ds + T_d * \dot{e}_v(t) \right\} + H * \ddot{y}_d(t) \quad (2.21)$$

To simplify this analysis, and since there are no gravitational terms required in the Carleton manipulator and no disturbances, we can make the integration time constant, T_i , very large, essentially neglecting the integration. Under the assumption that T_i is infinite, the system becomes second order. Taking this assumption and including the manipulator dynamics $u = H\ddot{y}$, we have:

$$H * \ddot{y}(t) = K_p * e_p(t) + K_p * T_d * \dot{e}_v(t) + H * \ddot{y}_d(t) \quad (2.22)$$

This can be rewritten in the error dynamics form:

$$\ddot{e}_p(t) + \frac{K_p T_d}{H} \dot{e}_p(t) + \frac{K_p}{H} e_p(t) = 0 \quad (2.23)$$

which has the characteristic equation:

$$s^2 + \frac{K_p T_d}{H} s + \frac{K_p}{H} = 0 \quad (2.24)$$

with roots

$$s_{1,2} = -\frac{K_p T_d}{2H} \pm \frac{1}{2} \sqrt{\frac{(K_p T_d)^2}{H^2} - \frac{4K_p}{H}}$$

(2.25)

This system can then be analysed for optimal pole placement. The technique which results in minimum response time with no oscillation and thus no overshoot is critical damping.

Critical damping can be applied by setting:

$$(K_p * T_d)^2 - 4K_p H = 0 \quad (2.26)$$

This method of using the system error dynamics for selecting gains is described in [Cra86] and [Cra87].

The physical parameters of the system available from [Raj89] include the link masses and lengths. The moments of inertia have been approximated for this research by representing the links as thin rods instead of the 'H' cross-section actually used for construction. Table 2.1 gives the physical parameters used, which results in the following effective inertia values:

$$H_{11} = 0.3221 \text{ [kg-m}^2\text{]}$$

$$H_{22} = 0.2123 \text{ [kg-m}^2\text{]}$$

Link	Mass (kg)	Length (m)	Moment of Inertia I_j (kg-m ²)
1	2.17	0.3048	0.0672
2	1.37	0.2540	0.0295
3	1.70	0.3048	0.0526
4	1.21	0.3556	0.0128

Table 2.1 - Approximate Physical Characteristics of the Carleton Manipulator

To specify the feedback gains, values for K_p and T_d must be selected. The choice here effectively specifies the control bandwidth, which in turn dictates how the controller handles uncertainties in the parameters. The larger the gains (and thus the control bandwidth), the better the tracking performance. Countering the desire for a large control bandwidth is that the control bandwidth is limited by three factors [Asa86]. The bandwidth must be:

- less than the arm's structural resonance modes so that the controller does not excite unmodelled high frequency dynamics;
- less than any effect from unmodelled time delays, for example in the actuators; and
- somewhat less than the sampling frequency (usually less than 10%).

Furthermore, the actuator electronics should not be saturated by the controller. After some consideration and experimentation, $K_p = 18.0$ and $T_i = 50$ seconds (the integration time

constant) were selected for both links, and the anticipation periods were specified by solving equation (2.26) as follows:

$$T_{d1} = 0.26754 \text{ seconds (link 1 anticipation period)}$$

$$T_{d2} = 0.21720 \text{ seconds (link 2 anticipation period)}$$

Note that these parameters, and thus the control bandwidth, were maintained constant throughout the experimentation presented in this thesis.

Selection of the various parameters to use in the estimation routine is the subject of substantial research (e.g. [For81], [Ast89]). The covariance matrix P is usually initialized to a large constant (e.g. 10000 times the dimension 4 identity matrix). The constant selected may be arbitrarily large [Ast89], but a limit of 10000 appears to maintain numerical integrity while providing for fast parameter convergence. The regression vector $\phi(t)$ was initialized to a vector of zeroes, denoting a starting state at the origin in all frames. The estimates $\hat{a}_1$, $\hat{a}_2$, $\hat{b}_1$, and $\hat{b}_2$ are usually initialized to educated guesses of the actual parameter values, although since the controller in question allows initial convergence before use, they were initialized to zeroes. The forgetting factor can roughly be viewed as determining how long sampled values are significant to the calculation of subsequent parameter estimates. A value of 0.99 indicates on the order of 100 samples are used in the estimation, whereas 0.999 indicates roughly 1000 samples are used. Typically values closer to 1.0 are used when the parameters are not varying or are very slowly changing, although faster and less accurate convergence is usually obtained with lower values. In the

included examples, a value of 0.999 was used since the parameters of the simulation are fixed.

2.4 Manipulator Simulation

The simulation of the manipulator utilizes the dynamic equations of motion (equation (2.4)) to achieve a close approximation to the real manipulator. A 4th order Runge-Kutta integration routine is used to perform the simulation given the torque/voltage from the controller. The interface to the manipulator simulator matches the interface to the D/A, A/D, and D/D drivers of the real manipulator. The A/D converter modelled is the Acromag Series 9510 VME Multi-function card. The velocity voltage is converted from analog to digital one link at a time with a delay of approximately 32 μ s between channels. The digital position information is available simultaneously for both channels. The A/D conversion is driven by a periodic interrupt, which is triggered when the analog velocity reading has settled for the first link, at which time link 1 velocity and the positions of link 1 and 2 can be read. After the 32 μ s delay, link 2 velocity is available. Two ports are available simultaneously for link 1 and link 2 control output.

Modelling in this detail allows the simulator and manipulator to be interchanged as required while testing new algorithms, and ensures that the manipulator simulation runs at a close approximation to the rate of the real converter and manipulator system.

2.5 Trajectory Selection and Generation

The control experiments performed in this research are directed towards achieving very fast sampling rates and as such, other than ensuring persistent excitation, the trajectory selection is not a main focus of the work. Due to this, the separate trajectory for each joint is assumed to be available at the start of any motion. This implies that an activity at a higher level of the control hierarchy is assumed to have previously determined the joint trajectories required to achieve a specific end-effector goal. This information would then have been passed as a set of angular positions and velocities down to the control software. Such trajectory planning is also a subject of research as evidenced by the work in [Bor89], [Lis89], [Nec89], and [Dom87].

To ensure the persistent excitation condition is met and to avoid the manipulator's singular points, the main trajectory consists of the repeating circular motion described in [War89]:

$$\begin{aligned} y_1 &= 1.5 + \cos(2\omega t) - \cos(4\omega t) \\ \text{and } y_2 &= \frac{\pi}{2} - 1.5 + \sin(\omega t) + \sin(2\omega t) \end{aligned} \tag{2.26}$$

These position equations were differentiated to find the corresponding velocities, and the trajectory is followed for 20 seconds in the experimentation.

The trajectory generation used for startup from the origin to the main trajectory sequence and for slowdown and stopping at the origin is based on method specified in [Lee86a],

where the trajectory is broken up into three segments. The first segment is derived from a fourth-degree polynomial which provides a continuous trajectory satisfying the desired boundary value conditions for the initial and first intermediate point (or lift-off position). The boundary value conditions are the desired angular positions, velocities, and accelerations of the joint.

The second trajectory segment is derived from a third-degree polynomial satisfying the boundary value conditions set by the lift-off position and the second intermediate point (or set-down position). The third segment then completes the trajectory from another fourth-degree polynomial satisfying the boundary value conditions set by the set-down position and the final state.

This method was implemented for both the startup and slowdown parts of the overall trajectory by simultaneously solving for the coefficients of the three polynomials given the segment timing information and the eight boundary value conditions:

- 1) initial angular position
- 2) lift-off angular position
- 3) set-down angular position
- 4) final angular position
- 5) initial angular velocity
- 6) final angular velocity
- 7) initial angular acceleration
- 8) final angular acceleration

The intermediate boundary value velocities and accelerations were determined by equating those values at the segment interfaces. After determination of the coefficients, the actual startup and slowdown trajectory points were generated to match the sampling rate of the controller, using the appropriate segment polynomial at each time step.

The starting and stopping trajectories cover 1 second each, and attempt to bring the manipulator $\frac{1}{10}$ th of the way towards the startup or stop point in 0.2 seconds, $\frac{8}{10}$ ths of the way in the next 0.6 seconds, and the final $\frac{1}{10}$ th of the way in the final 0.2 seconds. Figure 2.9 gives the corresponding time line showing the use of the different polynomials and trajectories. Figures 2.10 and 2.11 show the desired link 1 and 2 slow trajectories, and Figures 2.12 and 2.13 show the desired link 1 and 2 fast trajectories.

Figure 2.9 - Trajectory Segment Timeline

Figure 2.10 - Link 1 Slow Trajectory

Figure 2.11 - Link 2 Slow Trajectory

Figure 2.12 - Link 1 Fast Trajectory

Figure 2.13 - Link 2 Fast Trajectory

3.0 Parallel Processing for Real-Time Control

This chapter discusses the background research in parallel techniques carried out in support of this thesis. An overview of parallel architectures is provided along with a survey of various hardware implementations for manipulator control. Following this is a review of the research involving parallelism in robot control algorithms, with a section covering the work specifically related to adaptive control. Several tools used to aid the design of parallel software are then introduced, and lastly the decisions leading to the choice of the VMEbus shared-memory multiprocessor are given.

It should be noted that there are several areas of robotics where research into the application of parallel processing techniques is being studied. A major area is at the planning level, where researchers are looking at the integrated systems approach to path planning, collision avoidance, vision sensing, etc. An example of this research is in the work by Elgazzar, Green, and Gentleman [Elg88] and Green and Liscano [Gre89], where they are using a multiprocessor architecture with the Harmony operating system [Gen89] to develop an integrated, intelligent robot system providing a flexible, scalable base for applications such as autonomous robot control, telerobotics, and dual manipulator control.

Another example is provided in the work by Kossman and Malowany [Kos87], where they discuss a multiprocessor implementation of the Robot Control C Library (RCCL) with its associated underlying Real-Time Control system (RTC). RCCL provides the equivalent of a manipulator control language implemented as C functions and macros. Programmers use

the RCCL functions to specify the motion of the manipulator, including both the trajectory and the end-effector orientation. RCCL communicates with RTC which then implements the commands.

The research in this thesis concentrates on the areas associated with implementation of the joint-level control algorithms. As such, most of the background research carried out covered the areas of hardware selection, control algorithm decomposition, and parallel design tools. There is considerable literature in each of these areas, and the following sections provide an introduction to this body of work.

3.1 Hardware Alternatives

When building a parallel control system, selection of the hardware is the most crucial process in determining the throughput potential of the implementation. This section provides some examples and guidance in making this selection. If possible, the final solution should be left until the parallelism inherent in the control algorithm is well understood.

The selection of the parallel architecture on which to base a controller implementation depends mainly on the granularity of the parallelism exploitable within the control algorithm structure. The granularity has a major effect on the amount of communications, since fine granularity implies many communicating processes. In turn, the computation to communication ratio for each process then provides a view on which type of processor

interconnection network would be acceptable. Furthermore, the granularity has a bearing on the required computation and memory capabilities of the processors themselves.

Table 3.1 summarizes some of the tradeoffs in the selection of an architecture. The first entry is the uniprocessor (single instruction single data class, or SISD), which is the simplest to program and to maintain. The steadily improving capabilities of new microprocessors makes this choice sometimes preferable over parallel architectures. The next entry represents the single instruction multiple data (SIMD) class of parallel machines, such as the Connection Machine. This architecture is valuable when algorithms exhibit very high amounts of symmetry, such as large digital signal processing systems. The hypercube interconnection network shown is only one example.

The final four entries in the table are of the multiple instruction multiple data (MIMD) class, which covers both multiprocessor and multicomputer systems. The distinction between multiprocessors and multicomputers is the memory model used, and is sometimes blurred. A multiprocessor is a collection of processor modules accessing a separate shared memory module (or modules). A multicomputer is a distributed memory machine, where there are several separate computing modules, each with a processor and local memory. The blurring occurs in those 'multicomputers' where each processor's local memory is accessible from other computers.

The interconnection networks for MIMD class machines may be of any architecture. Apparently, the global-bus (e.g. [DY-88], [Phi90]) and hypercube models (e.g. [Kim90]) are the most common, although dual-ported memory pipelined models (e.g. [Tak89]),

serial link pipelined models (e.g [Chi90], [Ish89]), and switch-based networks, among others, are the topics of continuing research. An excellent survey is provided by [Hay82].

Class	Mem. Model	Mem. Size	Interconn'n Network	Comm. Time	Typical No. of Proc's	Typical Parallelism
SISD	-	Large	-	-	1	none
SIMD	Dist.	Small	hypercube	Medium	1K-64K	high-very high
MIMD	Shared	Large	global bus	Short	1-16	medium-low
MIMD	Shared	Large	dual-port mem.	Short	1-8	medium-high
MIMD	Dist.	Small	pipeline	Medium	5-20	medium-high
MIMD	Dist.	Large	hypercube	Long	2 ⁿ (2-128)	medium

Table 3.1 - Parallel Architecture Trade-Offs

Table 3.2 provides a survey of recent implementations of parallel robotic controllers. This table updates the information provided by Leung and Shanblatt in [Leu88]. The best-looking performance was a sampling rate of 21 KiloHertz (KHz) by the special purpose, two ALU JIFFE processor [And89] for the sensitivity analysis computation of inertial coupling, Coriolis forces, and centripetal forces (presumably) for a 6 degree-of-freedom (DOF) manipulator. This rate does not include the control algorithm computations, and the author states that implementation of an associated force control algorithm would bring the sampling rate closer to 1 KHz. The most modest achievement was that of Fleming, Jones, and Jones [Fle89], where a Transputer network processor farm using dynamic scheduling obtained a sampling rate of 38 Hz for a 6 DOF Newton-Euler (NE) inverse dynamics computation.

Besides the actual implementations surveyed in the tables, there have been several other architecture proposals put forward without implementation results. For example, Kabuka and Escoto [Kab89] discuss another implementation using four NEC μ PD77230 DSP chips with a host 80386 or 68020 and shared memory to achieve up to 3.3 KHz sampling rate using Newton-Euler inverse dynamics computations. Nigam and Lee [Nig85] proposed a distributed memory pipelined structure with a global bus that would obtain approximately a 1.6 KHz sampling rate using seven 68020 processors to compute the 6 DOF Newton-Euler inverse dynamics. Houstis and Aboelaze [Hou87] investigate a multiple bus-based shared-memory multiprocessor system, achieving 880 Hz for NE computation with an 8 processor, 8 bus system. Their paper details the effect of varying the processing time to communication time ratio under a heuristic scheduling algorithm. They also compare the multiple-bus systems to an 8 processor Banyan network system.

Author	Applic- ation	CPU	No. CPUs	Comm- unication	Scal- ability	Gran- ularity	Alloc- ation	Perfor- mance
Ahm89	6 DOF NE Inv. Dyn. & Kin.	68020 & 68881's	1 CPU & 6 APU's	SM, internal bus	open	fine	static	400 Hz
And89	6 DOF dyn.	VLSI	1 w/2 ALUs	Internal bus	none	very coarse	n/a	1 KHz (21 KHz - dyn. only)
Bro90	4 DOF Decentr. Adp.Ctl.	80286/7 & 80196	Master & Slave	Dual- ported SM, int. bus	none	very coarse	static	1 KHz PD, 250 Hz AC
Buh89a	1 DOF Juggling	T800	3	DM., fibre links	open	coarse	static	1 KHz
Fle89	6 DOF NE Inv. Dyn.	T414-G15	2-6	DM, 20 Mb serial links	open	fine	dyn- amic	38 Hz
Jon89	3 DOF NE Inv. Dyn.	T414-G15	5	DM, 20 Mb serial links	open	fine	dyn- amic	67 Hz
Nar89b	Utah-MIT hand	68020	8	SM, VMEBus	open	coarse	static	400 Hz
Par90	6 DOF NE Inv. Dyn.	T800	12	DM, 20 Mb serial links	open	fine	static	300 Hz
Tak89	6 DOF Active Stiffness	μ PD77230 80286, & 8086	3	SM, internal bus	open	coarse	static	1 KHz

Table 3.2 - Survey of Parallel Architectures for Robotic Control

(SM = shared memory, DM = distributed memory)

Although control theory and signal processing theory are closely related, robotic control algorithms tend to be non-symmetrical, unlike signal processing algorithms. However, parts of a robotic control algorithm may be similar to signal processing, such as the estimation problem. Thus there is a disagreement currently in the literature as to whether or not the use of special-purpose DSP hardware is valuable in robotics work. Certainly some results uphold the usefulness, as in the work by Takanashi, Ikeda, and Tagawa [Tak89], and by Kabuka and Escoto [Kab89]. Others, such as Andersson [And89] debate that more heterogenous multiprocessor systems are required.

3.2 Parallel Research on Manipulator Control Algorithms

All of the architecture experiments described above resulted from preceding research into the control algorithms that were implemented. The majority involve the Newton-Euler (NE) inverse dynamics computation, where the inherent parallelism involved is clearly exploitable by parallel processing. The associated research covers the techniques used to decompose the algorithm and the techniques used to allocate the activities onto parallel architectures.

Research in the parallelization of adaptive control algorithms is not nearly as common. Substantial research involving the actual control algorithms is available, but the majority appears in the form of related work on parallelization of parameter estimation algorithms.

3.2.1 The Inverse Dynamics Problem

There has been significant research into the application of parallelism in computing forward and inverse dynamics. The inverse dynamics calculations use the joint positions, velocities, and accelerations along with the robot's dynamics and with external applied forces/torques to determine the joint torques required at each time step. Forward dynamics calculations use the same techniques to reverse the calculations for simulating the effect of torque input on a robot.

This problem structure has exploitable parallelism in each of its three formulations, the Lagrange-Euler method requiring $O(n^4)$ calculations, the Generalized D'Alembert method requiring $O(n^3)$ calculations, and the Newton-Euler method requiring $O(n)$ calculations. For obvious reasons the Newton-Euler method attracts the most attention, with systolic array (e.g. [Lat85], [Has89]), SIMD class (e.g. [Lee86b]), and multiprocessor implementations (e.g. [Hou87]) appearing in several forms. Multiprocessor scheduling of the Newton-Euler method has been the subject of many of these papers (e.g. [Ahm89], [F'ou87]). A hypercube implementation was studied by this author in [Gri89a], and a comprehensive survey is available in [Gri89b]. Note that the author did not pursue this research due to the inability to apply estimation techniques to the NE formulation.

3.2.2 Parallelism in Adaptive Control

Very little research has been carried out with the specific target of parallelization of adaptive control algorithms. Lainiotis, Likothanasis, and Katsikas [Lai88] compare the optimal

Linear Separation Control algorithm, a sub-optimal adaptive version of this developed by Lainiotis, and a Predictive Model self-tuning regulator. They claim that neither the Linear Separation Control algorithm nor the self-tuning regulator can gain from parallelization, but that the Lainiotis algorithm may be decomposed into a number of simultaneous Kalman filters. A short analysis of computational and memory complexity is provided for comparison. Note that the conclusion they make regarding self-tuning regulators is incorrect, as the rest of this thesis shows.

Nader [Nad79] performed some very interesting research on the parallelization of a model-referenced adaptive control algorithm, showing the use of Petri Nets (see next section) in design and initial performance analysis. He provides two decompositions of the algorithm, one based on functional decomposition and the other an extension using inherent parallelism within a matrix inversion operation. A simple two processor version is further extended to include the effect of communication delays, although resource contention is ignored.

Parallel techniques applied to parameter estimation have enjoyed considerable research. Chisci and Zappa [Chi88], for example, investigated the systolic array implementation of various flavours of the recursive least squares algorithm. They consider several square root formulations of the problem, deriving approximations that are implementable as fast (i.e. $O(n^1)$) systolic arrays with small degradations in estimator convergence or tracking. Kim and Agrawal [Kim90] put forward two algorithms for applying Cholesky factor (square root) multiple updating to recursive least squares filtering on hypercube architecture machines, one using Givens rotations and the other using Householder reflections.

Hashemipour, Roy, and Laub [Has88] developed parallel algorithms for synchronous and sequential data Kalman filters, again using the square root formulation. All of these papers target very large signal processing applications.

Finally, a lot of research has been carried out into the optimization of matrix-vector multiplication, potentially applicable to non-square root implementations of recursive least squares estimation. Zubair and Madan [Zub87], for example, propose a bit-level pipelined systolic array for the matrix-vector product, following on from earlier work by Kung and Leiserson [Kun78]. Grigg, Miguet, and Robert [Gri90] developed an optimal (in time and memory space) algorithm to perform a multiplication between a symmetric matrix and a vector on a ring-shaped multiprocessor, providing experimental results from a Transputer hypercube configured as a ring. Both of these methods are again directed to large-scale applications.

3.3 Parallel and Distributed Software Design Tools

In this section the various tools used in the design and performance analysis of the controller are introduced. These include Activity Sequence Networks developed by Woodside [Woo88a], a Petri Net tool [Chi87], and critical path analysis using Gantt charts. There exists a plethora of alternative design and analysis tools such as Rendezvous models [Nei88] and [Woo85], MachineCharts [Buh89b], Real-Time Temporal Logic [Ost87], Communicating Finite-State Machines [Bra83], Process Activity Language [Kri88], and Communicating Sequential Processes [Bro84]. An excellent survey of older work in this

area of concurrent programming design is found in the work of [And83], and a less extensive survey of newer methods may be found in [Leu90].

3.3.1 Software Decomposition Techniques

There have been several methods put forward on determining how to decompose a sequential algorithm for the purposes of parallelization. Garcia-Nocetti, Thompson, De Oliveira, Jones, and Fleming [Gar90] illustrate two methods, heuristic decomposition and parallel branching. Heuristic decomposition implies breaking down an algorithm in such a way as to maintain functional meaning. The example provided used the control system block diagram blocks as functional entities, so that each block retained its sequential nature while operating in parallel with other blocks. Parallel branching implies the use of parallelism inherent within the algorithm. In the example, transformations were performed on the blocks to obtain sets of parallel Laplace transform functions. This method results in some duplication of functionality, but overall parallelism and potential throughput is enhanced.

Other, graph-based, methods are common. For example, Mok [Mok84] builds a graph-based model providing communication and timing constraints. Each node in the graph is a timing constraint, consisting of a task precedence graph, period, and deadline. The task precedence graph nodes are the functions required by the timing constraint, weighted according to the execution time. This model is then used as a basis for task to processor allocation, as discussed in Section 3.3.3.

Nader [Nad79] illustrates the use of Petri Nets in the decomposition process, aiding in the determination of critical paths. The actual decomposition, however, was apparently made by visual analysis. Petri Nets are discussed in Section 3.3.3, below.

These techniques represent two basic paradigms used for software decomposition: breaking up the existing software by function or finer granularity; and transforming the algorithm into a parallel set of smaller problems. Of these two methods, the former is employed within this research.

3.3.2 Activity Sequence Network Notation

After decomposition of a system specification, modelling tools are helpful in structuring the resulting functions into a meaningful set of communicating activities. Such tools illustrate the precedence between activities, including data and control flow. Furthermore, some method of predicting computational performance is required in order to ensure deadlines are met and to identify critical paths.

The Activity Sequence Network (ASN) notation (see [Woo88b] and [Woo88a]), was developed to combine the data flow and control flow concepts in a single software description tool. ASNs fully support parallelism in the design and provide a hierarchical environment that allows the design team to hide details and thus work with very large systems. Capturing software designs in this form simplifies the task of performance modellers, since all precedence relations and data dependencies are included in one model. Furthermore, the model is closely related to timed Petri Nets and a tool is now available

[Woo90] to convert ASN models to Petri Net models for the express purposes of performance analysis and validation of structural correctness.

The paper describing the ASN to Petri Net conversion tool by Woodside and Romaniuk [Woo90], provides an excellent introduction to ASNs and uses part of the robot control system design described in this thesis as an example. The basics of the ASN notation used in this thesis is provided in Table 3.3.

Notation Name	Graphical Representation	Description
Simple Activity		A single activity or function within the system.
Complex Activity		An activity that contains hidden activities.
Start and End Blocks		A starting point or ending point of the network or an external control flow connection.
Arc		Connection between the activities.
Data Resource		Represents a data resource external to an activity.
External Data Resource		Represents a data resource external to the current level within the hierarchy.
Data Access Arc		Shows which data resources are accessed by an activity.

Table 3.3 - Activity Sequence Network Notations

3.3.3 Task-to-Processor Allocation and Performance Considerations

The process of allocating a set of activities onto tasks and of mapping these tasks onto processors has received perhaps the highest level of interest in real-time software engineering research. Optimal (in time, processor utilization, minimum bandwidth, etc.) approaches in multiprocessing stem from early work by Liu and Layland [Liu73] for single processor systems which resulted in the Rate Monotonic Priority Assignment. Although the Liu and Layland method was very simple, the multiprocessing methods suffer from the fact that optimal scheduling of tasks on multiprocessors is an NP-complete problem [Kim88], so that solutions are limited to very small systems.

To counter the NP-completeness, heuristic algorithms (e.g. [Ahm89]) were developed that take into account the hardware characteristics of the target. These methods have also been simplified to place restrictions on the number of tasks, to ignore dependency relations, to limit the optimization to either communication or computation time, and to only consider balancing of the workload. For example, Monforton [Mon89] proposes a generalized bin-packing algorithm for multiprocessor scheduling based on processor workloads with options that consider the hardware characteristics.

Alternately, for smaller systems, visual allocation methods can be employed. For example, Mok [Mok84] describes and compares three allocation methods based on his graph-based decomposition described in Section 3.3.1. The first method described is decomposition by Critical Timing Constraints (CTC), where tasks are composed of the activities within a single timing constraint. This is useful for designs with constant periods and deadlines.

The second method is decomposition by Centralizing Concurrency Control (CCC), where timing constraints that have evenly divisible periods and identical deadlines are grouped into tasks. These periodic tasks then form equivalence classes which tend to be independent thereby minimizing communications. The third method is decomposition by Distributed Concurrency Control (DCC), where each unique activity in every task precedence graph is made into a separate task with a period equal to the shortest period occurring in any of the precedence graphs. This method maximizes the parallelism in the system. Of the three methods, the first (CTC) yields the most maintainable code, but future modifications may result in missed deadlines requiring a major overhaul of the system. The other methods are much more flexible to modifications, but may underutilize the potential parallelism (CCC) or result in poor performance due to resource contention (DCC).

Performance engineering tools such as the timed Petri Net models can help with such analysis. Petri Nets were developed as an analytical modelling tool for ensuring the correctness of distributed software designs. The analysis covers structural boundedness which validates the control flow, safeness which ensures that no infinite looping can occur, liveness which ensures that all parts of the network are executed at some time, and deadlock avoidance which ensures that the system will not lock itself out over resource contention.

The net models consist of places, directed arcs, and transitions as shown in Table 3.4. The places hold tokens which indicate the state of the system. These tokens circulate around the network over the arcs by the firing of the transitions, and indicate a resource such as a

processor or data. Transitions fire when there are tokens in each of the places providing input to the transition.

Notation Name	Graphical Representation	Description
Place		Represent a state within the model.
Directed Arc		Direction of token flow between places and transitions.
Transitions		Represent a functional activity or synchronization point, respectively.
Tokens		Represent the allocation of a resource such as processing time or data.

Table 3.4 - Petri Net Notation

Later developments in Petri Net models included timing characteristics. Firing durations of either deterministic or stochastic periods were added to the transitions, allowing the performance analysis of both deterministic and non-deterministic systems. The timing addition provides a quick way of locating critical paths within the software structure. With an automated tool such analysis can be used daily in software development. Such a tool was developed by Chiola [Chi87] for building, checking correctness, and simulating the nets for performance analysis. A supplemental tool was developed during the research of this thesis to present the simulation timing results in the form of a Gantt chart for easier analysis. Another tool based on Petri Net models was developed by Bruno and Marchetto

[Bru89], called "process translatable", or PROT nets. Their tool generates ADA code directly from the net description.

Timed Petri Nets have been used for control system analysis by Nader [Nad79] and by Leung and Suri [Leu90], the latter comparing Petri Net models with other techniques such as queuing models, sensitivity analysis, and discrete event simulation.

3.3.4 Synchronization Techniques

Synchronization of activities in parallel and distributed processing systems is necessary to ensure proper data flow since the separate processors are rarely timed from the same clock crystal and therefore have unpredictable time complexity. One exception to this generalization is the systolic array, where all processors proceed synchronously. In multiprocessor systems, however, synchronization must be used when two or more processors must share data, either by shared memory or by messaging [And83].

There are two specific problems that synchronization solves. The first is to ensure that the data that processors access is up to date. The second problem, called mutual exclusion, is to mediate exclusive access between multiple processors when there are multiple writers to a variable or when a single writer may alter data incompletely accessed by readers. Mutual exclusion is necessary, for example, when one processor continuously updates values that are used cyclically by other processors.

Common synchronization techniques for *shared memory* systems include busy waiting, Dijkstra's P (atomic wait until positive then decrement) and V (atomic increment) semaphore operations, conditional critical regions (structured use of semaphores), and monitors (data hiding via the use of objects) (see [And83]). Busy waiting, the most common technique in multiprocessor systems, relies on the writer signalling via a semaphore upon completion of the variable access. The readers polling for the signal can then gain access to the data. In the case of multiple writers, busy waiting is accomplished through an atomic test and set operation.

Mutual exclusion can be implemented by the use of semaphores and atomic test and set. This method works by the processors vying for write access to the data first execute the P operation, access the code, then execute the V operation. It can also be ensured by careful scheduling or synchronization of the associated activities within each processor, where data is always updated only after the readers have completed their access.

In *message passing* systems such as Unison [Mul87], synchronization is accomplished through the use of messages since messages cannot be received until after they have been sent. This implies that readers of data from a single writer are ensured to have the correct data upon reception of the writer's message. Furthermore, mutual exclusion is accomplished through the readers' ability to put off receiving new messages until processing of the last message is completed. Multiple writers to data are similarly protected since the receiver can only receive and process one message at a time. Such message passing systems are thus extremely attractive for parallel problems. Their drawback is the processing overhead required to implement the message passing.

3.4 Description of the Development and Target Environments

This section presents the characteristics of the target system used for experimental verification of the design methods contributed by this thesis. The operating system, the hardware, and the associated monitoring tools are described.

The system used was selected on the basis of availability, but fortunately provides the required parallel capability and processing speed to approach the desired objective. The system is a VMEBus-based shared-memory multiprocessor (MIMD class) running the Unison multitasking multiprocessor operating system. It provides scalability in the lower range, allowing between 1 and 15 processors to be allocated to any problem.

3.4.1 The VMEBus Hardware

The VMEBus multiprocessor available at Carleton University is based on DY-4 DVME-134 processor boards [DY-88] and an IEEE P1014 standard VMEBus supporting 24-bit addressing and 16-bit data transfers. Each processor board employs a 12.5 MHz Motorola MC68020 microprocessor [Mot88], a Motorola MC68881 floating point coprocessor [Mot87], a Motorola 68901 multifunction peripheral for serial input/output, and a dual-ported 32 bit wide memory. Most of the boards consist of one megabyte of memory [Bai89]. Up to 15 processor boards may be supported along with an Ethernet interface board.

The multiprocessor supports a linear shared memory structure, such that each board is able to access the memory resident on other boards via the VMEBus using the second port of the dual-ported memory. On simultaneous memory access by the local processor and another via the VMEBus, the remote access is given priority.

3.4.2 The Unison Operating System

The Unison multi-tasking multicomputer operating system [Mul87] is supported on the VMEBus multiprocessor. Unison is a derivation of Harmony [Gen89], which introduced the send-receive-reply synchronous message passing paradigm. The difference between the two operating systems is that Unison:

- provides timeouts on the blocking *_receive* and *_send* primitives;
- allows processes to have more than one receive port; and
- allows processes to ignore outstanding messages on one or more receive ports, replacing the *_receive_specific* function in Harmony.

Unison supports multiple processes running on multiple processors, and provides the message passing functions for synchronization between the processes. The message passing mechanism is the same whether the processes share the same processor or reside on different processors, simplifying the programming effort.

One performance-affecting problem with the Unison operating system is the multitasking capability. Context switching between processes takes on the order of 60 μ s, potentially affecting the achievable throughput rate. Even if only one user process is allocated to a processor, Unison provides a null task known as the "idle task" which is swapped in whenever the processor is idle. Thus one objective of the software design for higher performance is to reduce or, if possible, eliminate context switching. Reduction can be obtained by allowing only one task per processor, at the expense of message-passing between activities allocated to the same processor.

Measurements have shown that the Unison interprocessor send-receive-reply message passing cycle take approximately 1 millisecond [Fra89], which is unfortunately of the same order as the desired control algorithm sampling period. This appears typical of bus-based multiprocessor systems (e.g. [Nar89b]), and implies that message-passing will severely restrict the ability to use parallelism within the control algorithms. Fortunately, the VMEBus architecture supports shared-memory, so Unison message passing is required only for the initial pointer addressing. Exclusive use of shared memory during the sampling periods also removes the possibility of idle task context switching, since synchronization will be based on shared memory semaphores.

3.4.3 The Event Monitor

In order to obtain performance measurements and sequencing data on the VMEBus multiprocessor, the Unison kernel was modified to include an event monitor [Bai90]. The monitor logs changes in task states to provide tracing of message passing events. Once

logging is activated for a processor, every state change in every task on the processor is logged. Each log entry contains the new state and the timestamp, providing a timing resolution of 50.5 μ s. In addition, user-defined events may be logged with a `_log_event` primitive.

As this is an intrusive monitoring technique, it was designed with minimum intrusion as a major objective. To this end, the bulk of the supporting software runs on a separate processor card. The supporting kernel software on the Unison master and slaves is minimal, simply writing the event code across the VMEBus to the local memory of the monitor card. In the case of user-defined events, the code is linked directly with the application, avoiding the requirement to context-switch. Thus the intrusiveness when monitoring user-defined events is due simply to the local software execution time to generate the write and to the extra bus utilization. As discussed in the previous section, no message-passing will occur after processor initialization, so only user-defined events will be logged to capture the performance of the robot controller.

The action of writing to the monitor memory causes a monitor interrupt to capture the timestamp. The intrusiveness and this interrupt limit the achievable resolution of the monitor, so it becomes less useful as the period between events decreases. This leads to the selection of 50.5 μ s as the timing resolution.

Logged events are stored in the monitor card memory, and thus the number of events that can be captured is limited. Supporting software has been written to provide a method of retrieving the events stored on the monitor card. An uploader accessing the Ethernet allows

capture of the binary event log on the Sun host. A user definable post-processor then converts the binary data into an ASCII result file containing the timestamps and/or the duration between subsequent events.

4.0 Design of Parallel Controller Software

With the control algorithm defined in Chapter 2 and the hardware described in Chapter 3, the stage is set for the design of the parallel implementation. As discussed in Chapter 3, there are many approaches available when designing parallel software. The use of the VMEbus multiprocessor architecture and the Unison operating system, however, imposes several constraints on the design approach that must be taken.

This chapter concentrates on defining and illustrating the design process for the parallel implementations of the software. The design choices are highlighted along with the criteria used for selection between the alternatives. After the individual activities are derived using decomposition techniques, a precedence graph based on dataflow constraints is used to order the activities into an Activity Sequence Network. After timing results have been obtained from a sequential simulation, this network is then converted, under several assumptions, into a timed deterministic Petri Net model. This model is then used to determine the critical path and to ensure safeness of the design. The Petri Net model is also shown to be useful in determining where effort should be spent in further parallelization of the software. This analysis results in a predicted throughput that will be verified by experimentation in Chapter 5. Next, the details of the implementation are discussed. These include the issues of inter-processor synchronization technique and design for flexibility. The chapter concludes with a summary of the methodology used.

4.1 Algorithm Decomposition

This section provides an analysis of the controller system with the goal of decomposing the overall problem into a set of small activities. This analysis is necessary in the design of any software system, ensuring modularity and maintainability. In parallel system design, this step forms an important basis as to how efficient an implementation may be obtained.

Essentially the end objective is to define an intuitive activity set which both simplifies future design changes or additions and maximizes the potential parallelism.

The decomposition method used in this analysis is known as the heuristic, or functional, approach as discussed in Section 3.3.1. This is the most intuitive decomposition approach, and derives a set of independently sequential activities, each of which implements a specific function. Again note that the decomposition objective is to duplicate the sequential functionality of the control and estimation algorithm. This design effort does not investigate multiple cycle rate implementations.

The following subsections define the functional activities.

4.1.1 The Feedback and Feedforward Control Laws

The calculation of the individual feedback and feedforward control laws form two coarse-grained activities. The calculation is not the only activity involved, however. Both control laws require input data and must supply their output to one or more other activities.

Furthermore, the feedforward controller output is not used unless the estimation parameters have converged, so the detection of convergence also decomposes down into two sub-activities.

The *feedback controller* consists of the following sub-activities:

- **read trajectory**: obtains the current desired position and velocity;
- **receive feedback**: receives the sampled robot position and velocity;
- **calculate u_b** : calculates the PID control from the equations:

$$e_p(k) = y_d(k) - y(k) \quad (4.1)$$

$$e_v(k) = \dot{y}_d(k) - \dot{y}(k) \quad (4.2)$$

$$u_b(k) = u_b(k-1) + K_p * \left\{ \left(1 + \frac{T}{T_i} \right) * e_p(k) - e_p(k-1) + T_d * [e_v(k) - e_v(k-1)] \right\}; \quad (4.3)$$

- **send u_b** : transmits the control output.

The *feedforward controller* consists of the following sub-activities:

- **read trajectory**: obtains the previous, current, and next desired positions;
- **receive prediction error**: receives the prediction error for whiteness checking;
- **receive theta**: receives the latest model parameters;
- **check whiteness**: checks that the prediction error is only white noise;
- **check delta**: checks that the parameters change only slightly between samples;
- **calculate u_f** : calculates the feedforward control from the equation:

$$u_f(k) = \frac{1}{\theta_1} * [y_d(k+1) + \hat{a}_1 * y_d(k) + \hat{a}_2 * y_d(k-1) + \hat{\theta}_2 * u_f(k-1)]; \quad (4.4)$$

- **send up:** transmits the feedforward control output.

Each of these activities are required independently for both of the link motors, so a functional breakdown commonly seen in the parallelism of robotic control systems is to use a separate processor to control each link. This separation results in a duplication of these activities, usually enhancing throughput capability. Where this method may fail in improving performance is when the extra communications required and resulting resource contention outweighs the computational gain. This issue is dealt with in Section 4.3 which covers performance analysis.

The 'receive' and 'send' activities for these controllers and for the other activities to be defined below have not yet been specified. Their implementation may range from nothing when the activities are within the same task, to a complex synchronization activity for separate tasks on separate processors. The specification must take place later during allocation of the activities to tasks and processors.

4.1.2 The A/D Interface

The software interface to the VMEBus card supplying the analog-to-digital input from the position and velocity sensors and providing ports to the voltage input for the joint motors is a suitable activity in itself. It must receive the periodic interrupt signal from the device and read the position and velocity. It must also receive the feedforward and feedback control torques, sum them, and apply the sum to the digital-to-analog port on the card. These activities may be decomposed functionally into 'sampling' and 'summing' activities.

The *Sampling* activity make then be decomposed into the following sub-activities:

- **read y** : read the digital input port for the position information (both links available immediately after interrupt);
- **read v_1** : read the analog input port for the link 1 velocity value (available immediately after interrupt);
- **read v_2** : read the analog input port for the link 2 velocity value (poll for availability);
- **send old y** : send the last output torque to the estimator;
- **send y and v** : send the position and velocity values to the *feedback controller* and the position to the estimator.

The *Summing* activity may be decomposed as follows:

- **receive u_b** : receive the feedback control torque;
- **receive u_f** : receive the feedforward control torque;
- **calculate u** : sum the input torques as follows:

$$u(k) = u_b(k) + u_f(k) \quad (4.5)$$

- **write u** : write the control torque to the digital-to-analog output port;
- **send u** : send the control torque to the estimator.

4.1.3 The Estimator

The estimation routines may intuitively be broken down into the individual activities required for the calculation of equations (2.19) and (2.20). These activities include updating the $\phi(k)$ vector, calculating matrix-vector product $P(k-1) * \phi(k)$, calculating the vector product $\phi^T(k) * P(k-1) * \phi(k)$, updating the covariance matrix $P(k)$, calculating the prediction error $y(k) - \theta^T(k-1)\phi(k)$, and updating the parameter estimates.

Phi Update involves, for each link, adding the corresponding values to the vector:

$$\phi^T(k) = [y(k-1) \quad y(k-2) \quad u(k-1) \quad u(k-2) \quad]$$

and requires the following sub-activities:

- **receive u:** receive the previous applied control torque;
- **receive y:** receive the previous sampled position values;
- **update phi:** shift old values and apply the new values to the vector;
- **send phi:** send the resulting vector to the *prediction error* and the *matrix-vector product* activities.

The *Matrix-Vector Product* per-link activity involves the 4x4 matrix-vector product and vector product calculations of the following two equations:

$$K(k) = P(k-1) * \phi(k) \tag{4.6}$$

$$d(k) = \lambda + K^T(k) * \phi(k) \tag{4.7}$$

and can be decomposed into the following sub-activities:

- **receive P:** receive the previous covariance matrix;
- **receive phi:** receive the regression vector;

- **product:** calculate equations (4.6) and (4.7);
- **send $\mathbf{K}$ and $\mathbf{d}$:** send the results to the *covariance* and *parameter update* activities.

Similarly, the *Covariance Update* activity involves the per-link matrix operations given in the following equation:

$$\mathbf{P}(k) = \frac{1}{\lambda} \left[\mathbf{P}(k-1) - \frac{\mathbf{K}(k)\mathbf{K}^T(k)}{d(k)} \right] \quad (4.8)$$

This activity is decomposed into the following sub-activities:

- **receive $\mathbf{K}$ and $\mathbf{d}$:** receive the $\mathbf{P}$ - ϕ product results;
- **update:** update the covariance matrix according to equation (4.8);
- **send $\mathbf{P}$:** send the updated matrix back for the next matrix-vector product.

Calculation of the *Prediction Error* requires the evaluation of the following equation:

$$e_{\theta}(k) = y(k) - \hat{\theta}^T(k-1)\phi(k) \quad (4.9)$$

and may be functionally decomposed into the following sub-activities:

- **receive ϕ :** receive the current regression vector $\phi(k)$;
- **receive y :** receive the current sampled position values;
- **receive θ :** receive the latest parameter estimates;
- **calculate e :** solve equation (4.9) for $e_{\theta}(k)$;
- **send e :** send the resulting prediction error to the *Feedforward Controller* and to the *Parameter Update* activities.

Finally, the *Parameter Update* activity involves the following per-link vector update:

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{K(k)}{d(k)} * e_{\theta}(k) \quad (4.10)$$

and is decomposed into the following sub-activities:

- **receive e**: receive the latest prediction error;
- **receive K and d**: receive the P-phi product results;
- **calculate theta**: solve equation (4.10) for the new parameter vectors;
- **send theta**: send the updated parameters to the *Feedforward Controller* and the *Prediction Error* activities.

4.2 Analysis of Dataflow Constraints

The activities defined in the previous section tell only half the story. The algorithm dataflow determines the precedence constraints that link the activities together. These constraints form part of the standard software design process, sequencing the software so that each activity receives its correct operating data at the correct time. Including this information allows formulation of the design as an Activity Sequence Network (as described in Chapter 3).

4.2.1 Dataflow Constraints

On initial inspection of the control and estimation algorithms of equations from equations (4.1) to (4.4) and after analysis to obtain the controlling output $u(k)$ in the smallest time

possible, we obtain the precedence graph of Figure 4.1, which charts the dataflow requirements of the calculations.

Figure 4.1 - Dataflow Precedence Graph of Calculations

This figure, and Figure 4.2, represent the dataflow precedence constraints between the activities. The encircled data represents the activity associated with obtaining or deriving the data. Non-encircled data represents data that has already been obtained or computed by a prior activity, and is thus available for use. The arcs represent the data availability constraints placed on each subsequent activity.

The control is calculated at the fourth and fifth steps of this process after sampling the input, determining the predicted error, and calculating new parameters. The new values of K , d , and P for the next cycle are then calculated after the control has been output.

Although this figure would indicate that there is very little parallelism in the problem, at least the control is output before the time complex update of the covariance matrix P . This conclusion is typical in control algorithm analysis, since designers usually think in a sequential way when optimizing an algorithm for implementation.

The most obvious aspect of Figure 4.1 is its sequential nature. An initial improvement can be made by assuming that a delay of one cycle in applying the latest parameter estimates to the feedforward control law calculation will have no observable effect on system stability. This assumption can be justified since, as discussed in Section 2.3.3, parameters must only change slightly from cycle to cycle, and, by the *certainty equivalence principle*, eventually they will converge. This assumption allows the designer to separate the calculation of the control output from that of the new parameter estimate during a cycle, so that control output calculations may be started immediately upon receiving the sampled robot output, y .

Next, further analysis of the estimator equations yields the fact that two of the largest calculations, $\theta(k)$ and $P(k)$, are independent of each other and require both $K(k)$ and $d(k)$. Furthermore, the $e(k)$ and $K(k)$ calculations are independent of each other and both require $\phi(k)$. This information was used to derive the alternate dataflow of Figure 4.2.

This dataflow constraint analysis is part of the iterative design cycle. In general, this dataflow analysis may be revisited several times as the design evolves.

Figure 4.2 - Improved Dataflow Precedence Graph of Equations

4.2.2 Sequential Software Design

The data flow information can be used with the functional activities to develop the Activity Sequence Networks (ASNs) illustrating the design thus far. Use of the hierarchical constructs of the ASN notation (see Table 3.3) ensures that the design decisions to be taken in this complex system are easily understood to facilitate future modifications or maintenance.

Combining the activities of Section 4.1 with the dataflow constraints shown in Figure 4.1, the sequential software design may be mapped out as an ASN as illustrated in Figure 4.3 and Figure 4.4. Within each activity shown, the sub-activities discussed in Section 4.1 are performed sequentially. Note that the ASNs have been simplified for clarity by removing the startup and data flow arcs. The data flow arcs were removable since the sequential nature of the control flow results in the data flow constraints being redundant.

When referencing this and subsequent diagrams, the following Table 4.1 provides a mapping of the shortened names to their corresponding activities:

Name	Activity Name	Corresponding Eq'ns
Sample	Sampling	-
Back	Feedback Controller	4.1 - 4.2 - 4.3
Forward	Feedforward Controller	4.4
Sum	Summing	4.5
Phi	Phi Update	-
Pred	Prediction Error	4.9
Theta	Parameter Update	4.10
MVP	Matrix-Vector Product	4.6 - 4.7
Cov	Covariance Update	4.8

Table 4.1 - Shortened Activity Names

Figure 4.3 - Sequential Activity Sequence Network

Figure 4.4 - Sequential Estimator Software Activity Sequence Network

These nets were used as the basis for a sequential implementation of the system. This implementation was developed under the Copycat Unison emulator running on a Sun workstation to obtain estimates for prediction of the achievable system throughput. This work is covered in Sections 4.3 and 4.4.

Note that the sequential system shown here may be parallelized by the simple and direct method of separating the calculations for each joint onto different processors. This way the complexity for each processor is only $\frac{1}{2}$ of the total time complexity. This parallelism is not exploited in this research for several reasons which will be discussed in detail later, including excessively small granularity for the size of processors, resulting bus contention, and limitation in the number of available processors.

4.2.3 Parallel Activity Sequence Network Design

In this section we discuss the parallelism illustrated in Figure 4.2. The interrelationship of the data allows sequencing of the various activity streams in the problem. The corresponding ASN can be developed directly from the precedence graph as shown in Figures 4.5 and 4.6. Note again that the ASNs have been simplified for clarity by removing the startup arcs. The data flow arcs remain since they affect the parallelism attainable by the multiple control flows.

Figure 4.5 - Parallel Activity Sequence Network

Figure 4.6 - Parallel Estimator ASN

Within each of the activities, the corresponding subactivities proceed in a sequential manner. The entire system is driven by the Timer which provides an interrupt at the sampling period rate.

4.3 Design by Performance Analysis

The next step in the software design is to determine how to allocate the activities to processors in the multiprocessor. The "full parallelism" ASN described above might provide the ideal real-time solution, where each activity is assigned to one processor. Several design issues ensure rejection of this implementation, however. A requirement for

efficient processor utilization, for example, would certainly rule out full parallelism since some activities are of such short duration that the processor would be much less than 1% utilized. Communication times would also reject the fully parallel solution unless it could be shown that the communication penalty (including resource contention) was insignificant.

4.3.1 Architectural Constraints

The selected VMEBus multiprocessor architecture incurs some constraints on efficient use of the parallelism in this problem. The number of processors and support cards on the testbed VMEBus is limited by electrical considerations to 15 cards. The use of shared memory results both in a significant penalty for remote versus local memory access and in a chance of memory access contention. Furthermore, the use of a single bus implies that contention for access to the bus may be a problem under conditions of excessive bus utilization.

Tables 4.2 and 4.3, borrowed from [Fra89] and [DY-88], show the access times for the VMEBus and a slave processor's remote memory. Table 4.2 provides the delays inherent in requesting and gaining access to the bus, resulting in a total per-access delay of $1.19\ \mu\text{s}$. Table 4.3 shows the delays in accessing slave memory under various states of the slave's internal state. These delays can vary from $0.96\ \mu\text{s}$ to $3.6\ \mu\text{s}$. Thus the minimum remote per-access memory delay is $1.19+0.96=2.15\ \mu\text{s}$.

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Delaying Factor	Maximum Period
Request Speed	210 ns
Bus Mastership Delay	350 ns
Intercycle Delay	250 ns
DTACK* Latency	50 ns
Bus Release Delay	290 ns
Slave Release Delay	40 ns
Total	1190 ns

Table 4.2 - VMEBus Access Delays

Slave Bus Activity	Maximum Access Period
Local Bus Idle	960 ns
During Refresh	1250 ns
During Local DRAM Access	1280 ns
During Local I/O Access by CPU	1280 ns
During Local RMW Cycle Access	3600 ns
During Local MFP Access	2000 ns
During Local EPROM Access	1440 ns

Table 4.3 - Slave Processor Memory Access Delays

This delay is incurred whenever a remote byte or word is accessed. When a doubleword is accessed remotely, two VMEBus accesses are required since the testbed has only a 16-bit bus. Thus remote memory access of a doubleword incurs about a 4.3 μ s delay while a remote access of a short integer incurs about a 2.15 μ s delay.

These delays are incurred regardless of the data transfer method used, so block transfer of data between processors is no better than individual doubleword access for this architecture. The latter method may actually be superior since the bus will not be fully occupied for the time taken to transfer the blocks of data, thus reducing the chance of resource contention between processors. Individual access can be worse if some of the data must be accessed multiple times by the individual processors during a cycle, but the incidence of this is expected to be small.

It is important to realize that contention for shared resources may have a significant effect on system throughput. Obtaining accurate estimates of resource contention requires significant modelling effort, and is typically performed through the analysis of stochastic Petri Net models, Markov chains, or queuing network models. For the purposes of this research it will initially be assumed that the utilization of the bus is uniformly distributed across the processing time, and that the bus and memory contention between processors is negligible.

4.3.2 Sequential Implementation and Analysis

At this stage of design, it is necessary to come up with an estimate of the time complexity of each of the activities. This may be achieved by estimating the type and number of all the operations within the activities, determining the instructions that would implement the operations, and estimate the performance by counting the clock cycles. A simpler method is to begin implementation at this stage following the Rapid Prototyping thesis, and directly measure the performance. With a modular implementation using dynamically allocated

memory and explicit pointers, code developed at this stage will be easily adaptable for later use in the target implementation. Ease of adaptation is aided by the use of the CopyCat emulator on the Sun workstation, since code developed under CopyCat may be almost directly portable to the target. Furthermore, the turn-around time for debugging is much more rapid than direct implementation on the target.

The sequential implementation based on the preceding design was implemented and tested with the simulated robot. The Sun profiler *gprof* was used to obtain performance data on the code, and this information is captured in column one of Tables 4.4a and 4.4b (CopyCat Results).

Since the sequential implementation timing information was obtained through simulation on a Sun-3/140, there are a couple of considerations to be addressed when estimating the sequential throughput achievable on the target system. Although the processor on the Sun is a 68020 as is the target, the Sun's clock rate is 16 MHz whereas the target's clock rate is only 12.5 MHz. Thus the target achieves a clock period of 0.08 μ s versus the Sun's period of 0.0625 μ s, meaning the target will run about 1.28 times slower.

Furthermore, the target system asserts a wait state on each memory access bus cycle, whereas the Sun runs with 0 wait states. This results in a further degradation of performance of one third extra clock cycles during memory access. In this numerically intensive control system problem, it appears reasonable to assume that approximately 50% of the instructions involve memory access for data (the prefetch can normally be ignored due to on-chip instruction caching). These memory accessing instructions include **fmove**

and **fmuld** which account for about 60% of the total processing time and **mov.w** making up about 15% of the processing time (based on worst-case instruction times from [Mot88] applied to the sample code segment from the matrix-vector product activity, Figure 4.7). This means that about 75% of the total processing time is spent executing instructions that access memory.

```

moveq    #0, d0
movw     a6@(-0x6), d0
moveq    #0, d1
movw     _PROC, d1
addl     d1, d0
asll     #0x2, d0
movl     _P, a0
movl     a0@(0, d0:1), a0
moveq    #0, d0
movw     a6@(-0x8), d0
asll     #0x3, d0
addl     d0, a0
fmoved   a0@, fp0
moveq    #0, d0
movw     a6@(-0x6), d0
moveq    #0, d1
movw     a6@(-0x8), d1
addl     d1, d0
asll     #0x3, d0
movl     _phi, a0
fmuld    a0@(0, d0:1), fp0
moveq    #0, d0
movw     a6@(-0x6), d0
moveq    #0, d1
movw     _PROC, d1
addl     d1, d0
asll     #0x3, d0
movl     _K, a0
addl     d0, a0
fadd     a0@, fp0
fmoved   fp0, a0@

```

Figure 4.7 - Sample Code Taken From Matrix-Vector Product Activity

The additional overhead due to the extra wait state can be estimated by totalling the memory access instruction clock cycles from Figure 4.7 and determining how many extra wait state clock cycles occur during this code sample. These instructions consume a total of 340 clock cycles over 30 memory accesses resulting in an additional 30 clock cycles due to the wait states. Thus this mix of instructions results in approximately $\frac{30}{340} = 9\%$ overhead when one wait state is required in the bus cycle. Therefore the new time of an activity can be predicted using the following formula:

$$t_{\text{new}} = t_{\text{orig}} + 0.09*(0.75*t_{\text{orig}}) = 1.07*t_{\text{orig}} \quad (4.11)$$

Thus to convert between CopyCat timings on the Sun and expected sequential performance on the target system requires only a multiplication by the factor of $1.28*1.07 = 1.37$. These sequential predictions are provided in the second columns of Tables 4.4a and 4.4b (Sequential Timing Predictions).

Activity	CopyCat Timings [μsec]	Sequential Timing Predictions [μsec]	3 Proc. Timing Predictions [μsec]	6 Proc. Timing Predictions [μsec]
Sample	110	151	163	163
- read v_1	20	33	33	33
- read v_2	20	33	33	33
- read y	10	23	23	23
- send old y	20	28	30	30
- send y_v	30	42	44	44
Feedback Law	290	397	397	397
- read trajectory	40	55	55	55
- receive y_v	10	14	14	14
- calculate u_p	210	288	288	288
- send u_p	20	28	28	28
Feedforward Law	860	1178	1295	1295
- receive e	10	14	14	14
- receive θ	10	14	14	14
- read trajectory	40	55	55	55
- calculate u_f	310	425	425	425
- check whiteness	150	206	223	223
- check δ	340	466	536	536
- send u_f	20	28	28	28
Sum	130	178	191	191
- receive u_p	10	14	14	14
- receive u_f	0	14	14	14
- calculate u	90	124	124	124
- write to robot	10	23	23	23
- send u	10	14	16	16
Phi	120	165	198	198
- receive u	30	41	41	41
- receive y	10	14	14	14
- update ϕ	50	69	86	86
- send ϕ	40	55	57	57
Pred. Error	270	370	382	382
- receive ϕ	0	14	14	14
- receive y	10	14	14	14
- receive θ	10	14	14	14
- calculate error	230	315	324	324
- send e	10	14	16	16
Theta	330	452	525	525
- receive e	10	14	14	14
- receive k_d	10	14	14	14
- update parms	260	356	425	425
- send θ	50	69	71	71

Table 4.4a - Performance Predictions (in microseconds)

Activity	CopyCat Timings [μsec]	Sequential Timing Predictions [μsec]	3 Proc. Timing Predictions [μsec]	6 Proc. Timing Predictions [μsec]
Mat-Vec Product	1190	1630	1805	see Table 4.5
- receive P	10	14	14	
- receive phi	0	14	14	
- calculate kd	1110	1521	1693	
- send kd	60	82	84	
Covariance	1670	2288	2288	
- receive kd	10	14	14	
- update covariance	1430	1959	1959	
- send P	220	302	302	

Table 4.4b - Performance Predictions (in microseconds)

Activity	Processor 3 Timings [μs]	Processor 4 and 5 Timings [μs]	Processor 6 Timings [μs]
Mat-Vec Product	1195	973	934
- receive P	14	14	14
- receive phi	14	14	14
- calculate kd	1092	945	906
- send kd	84	0	0
Covariance	746	866	1089
- receive kd	14	16	16
- update covariance	732	850	771
- send P	0	0	302

Table 4.5 - Matrix-Vector Product & Covariance Update Performance Predictions

(Note: 'Processor N Timings' refers to the activity on processor number N)

The transformation of the CopyCat results to the target hardware leads to the expectation of a sequential implementation cycle period of approximately 6.85 ms, or a rate of 146 Hz.

4.3.3 Allocation of Activities to Multiple Processors

With the activity timings now available along with the potential parallelism illustrated in Section 4.2.3, the allocation of the activities amongst multiple processors may now be discussed.

There is much in the literature on allocation methods for hard real-time systems, as discussed in Chapter 3. These methods provide allocations that minimize the number of processors or that minimize the processing time. Determining the optimal (in time) allocation is considered an NP-complete problem and is not the subject of this thesis. Since the problem is presented in a deterministic fashion and there are only a small number of activities and processors, near-optimal allocations can be determined by inspection. These conditions are not always met in robotic control problems, as evidenced by the research into scheduling by Ahmed and Li [Ahm89] and by Houstis and Aboelaze [Hou87]. However, the adaptive control algorithms under consideration at Carleton seem to meet the conditions.

To determine the allocation, the information provided by the dataflow diagrams and the ASNs is combined with the performance information of the associated activities. In addition, communication time must be factored in. The objective is to divide the computation as evenly as possible amongst the processors while minimizing the amount of communication between them. Note that this analysis does not consider the multiple rate system possibilities, the objective is to provide functionality identical to the sequential implementation.

From the ASN of Figures 4.5 and 4.6, several sequential paths can be isolated. The following list gives some examples:

- Sample -> Phi Update -> Prediction Error -> Parameter Update
- Feedforward Controller -> Summing
- Sample -> Feedback Controller -> Summing
- Sample -> Phi Update-> Matrix-Vector Product -> Covariance Update

From the sequential timings in Table 4.4 it is obvious that the *Matrix-Vector Product* and *Covariance Update* activities are the most time-complex activities, at 1.630 ms and 2.288 ms respectively. Unfortunately these activities are sequentially dependent since each needs the result of the other to proceed. This analysis implies that one processor should be allocated to the last sequential path shown above. However, the closer analysis of the ASN in Figure 4.6 shows that no other activity is dependent upon the *Covariance Update* activity, implying that it can proceed during the processing of the *Sample* and *Phi Update* activities. Thus one processor should certainly be allocated solely to the *Matrix-Vector Product* and *Covariance Update* activities, at a cycle rate of approximately 3.92 ms.

This cycle rate can only be attained if there is no delay in supplying the *Phi Update* output to the *Matrix-Vector Product* activity. Thus another activity allocation should put the *Phi Update* activity at top priority, since some of its processing can be done before the *Sample* activity is completed. Beyond this *Sample* and *Phi Update* activity separation, activity allocation is non-critical since the entire processing complexity of the leftover activities is

only 2.89 ms, somewhat less than the 3.92 ms cycle rate. A logical breakdown would be to combine the controller activities together and the estimation activities together, thus minimizing the communications requirements.

Thus the initial selected allocation requires only 3 processors, as shown in Table 4.6 (where the 'expected utilization' values are derived later in this section).

Processor	Allocated Activities	Expected Utilization
1	Sample, Feedback Law, Feedforward Law, Summing	46.5%
2	Phi Update, Prediction Error, Parameter Update	24.1%
3	Matrix-Vector Product, Covariance Update	95.7%

Table 4.6 - Three Processor Activity Allocation

Next, to predict the actual attainable throughput, it is necessary to determine the communication penalties resulting from this allocation. Table 4.7 provides a list of the communications required by each activity, along with the corresponding delay incurred. The delay figures include only the transmission delays, no contention analysis is included.

Accessing Activity	Bus Activity	Location of Data	Data	Data Size	Delay [μ s]	Equation
Sample	P1 Write P2	Phi Update	y_rdy	1 integer	2.15	-
Phi Update	P2 Read P1	Sample	y	2 dwords	8.6	-
Sample	P1 Write P2	Pred. Error	y_rdy	1 integer	2.15	-
Pred.Error	P2 Read P1	Sample	y	2 dwords	8.6	4.9
Sum	P1 Write P2	Phi Update	u_rdy	1 integer	2.15	-
Phi Update	P2 Read P1	Sum	u	2 dwords	8.6	-
Phi Update	P2 Write P3	MVP	phi_rdy	1 integer	2.15	-
MVP	P3 Read P2	Phi Update	ϕ	40 dwords	172.0	4.6,4.7
Pred.Error	P2 Write P1	Forward	e_rdy	1 integer	2.15	-
Chk White	P1 Read P2	Prediction Error	e	4 dwords	17.2	-
Parm.Update	P2 Write P1	Forward	theta_rdy	1 integer	2.15	-
Chk Delta	P1 Read P2	Parameter Update	θ	16 dwords	68.8	-
MVP	P3 Write P2	Parameter Update	Kd_rdy	1 integer	2.15	-
Parm. Update	P2 Read P3	MVP	K,d	16 dwords	68.8	4.10
Total				82 dw,7 int	367.7	

Table 4.7 - Extra VMEBus Activity for 3 Processor Allocation

For example, the sequential timing from Table 4.4 for the *Calculate Theta* sub-activity in the *Parameter Update* activity was 356 μ s, and the communication time from Table 4.7 shows an extra 68.8 μ s for reading the K and d values. Thus the predicted timing for this sub-activity is:

$$t_p = 356 + 68.8 = 425 \mu\text{s}$$

This calculation is carried out for each of the sub-activities, and the results are listed in column 3 of Tables 4.4a and 4.4b.

A timed Petri Net tool (as discussed in Section 3.3.3) can be very useful at this stage to provide confirmation of the critical path and cycle length given the identified processes and estimated timings. Figure 4.8 illustrates the Petri Net model resulting from this three processor allocation. Note that all transitions are deterministic, reflecting the maximum delay expected by the corresponding activity. Some of the transitions lump consecutive two or more sequentially dependent sub-activities together for simplification. Examples are the 'RcvPhiYTh' transition in the *Prediction Error* activity, which adds the times for receiving each of the regression vector, the sampled position, and the parameter vector. Note that some of the arcs in the net correspond to both data passing and processor availability while others correspond to data alone. The latter are used only between cooperating processors or between communicating activities on the same processor that are not executed sequentially. The individual processor execution can be traced between activities by following the arcs terminating at places labelled 'PnFrom---'. Note that the cycle start is driven by an interrupt (denoted INT) from the Cycle transition.

This net was developed using the GreatSPN tool [Chi87] which allowed checking for structural boundedness. This ensured that an implementation based on this net will pass data as expected, with no token buildup at any place (i.e. no overwriting of data before it is used).

The next step to take is the generation of a Gantt Chart based on this Petri Net. The Gantt Chart, Figures 4.9 and 4.10, shows up the critical path of the 3 processor system as being the *matrix-vector product* and the *covariance updating* activities as was expected. In fact, processor 3 never ceases execution as was the design goal. This can be more clearly seen in the multiple cycle overview given by the Gantt Chart in Figure 4.11.

The Petri Net simulation obtained an estimated cycle period of 4.09 ms, an increase of 0.17 ms over the sequential timing-based prediction due to the synchronization overhead incurred.

All of the above calculation was carried out under the assumption of insignificant bus contention. As was mentioned in Section 4.3.1, this could be verified by first determining the bus utilization and then modelling the bus and memory usage. The expected bus utilization is determined by dividing the total time the bus is in use by the total processing time. The total predicted bus usage time is given by Table 4.8, where the bus usage of the *Sample* and *Summing* activities access to the A/D converter are added to the communication delays incurred by the three processor implementation.

Reason for Delay	Data	Delay [μ s]
Polling for A/D	5 integer reads	10.8
Sampling A/D Output	4 doubleword reads	17.2
Applying the D/A Control	2 doubleword writes	8.6
3 Processor Overhead	82 dws, 7 ints	367.7
Total	88 dws, 12 ints	404.3

Table 4.8 - Overall Predicted VMEBus Usage Time for 3 Processor Version

The expected bus utilization is therefore:

$$\rho_3 = \frac{404.3 \mu\text{s}}{4093 \mu\text{s}} = 9.9\%$$

This may or may not lead to significant contention, but for the stated predictions, the contention was assumed to be negligible. As for processor utilization, Table 4.6 includes the expected values derived by dividing the predicted sequential activity execution time for each processor by the cycle time.

Before this section is finalized, there is one obvious functional decomposition that has not yet been discussed. The analysis up to this point has treated the activities as working on both links at once. This double activity within each activity could be functionally separated, possibly to the extent that completely independent multiprocessors could be acquired to handle each link. This may seem desirable, but for the most part the duplication of calculations adds only a small amount of time complexity, and the rapid increase in

processors contending for the bus would likely result in only similar performance. The main reason, though, is that the majority of control algorithms involve interaction between the links, so combining the link calculations may result in more flexibility of the system.

Figure 4.8 - Three Processor Petri Net Model

Figure 4.9 - Three Processor Gantt Chart - Detailed

Figure 4.10 - Three Processor Gantt Chart - Detailed (continued)

Figure 4.11 - Three Processor Gantt Chart - Overview

4.3.4 Exploitation of Inherent Parallelism

The critical path appearing in the Gantt chart of the 3 processor version points out clearly that the throughput bottleneck to this problem is the *Matrix-Vector Product/Covariance Update* activity combination in processor 3. This section investigates the further parallelism that can be exploited to improve this bottleneck.

Decomposing the problem functionally is not the only method available for improving the response time of the system. For example, the decomposition of the Recursive Least Squares algorithm has been the recent subject of much research (e.g. [Chi88], [Kim90], [Gri90]). As a minimum one can at least decompose the most complex parts of the estimation problem, particularly the matrix updating operations.

To decompose such a functionally sequential activity like the *Matrix-Vector Product* or the *Covariance Update* activities, one looks for inherent parallelism within the problem. Inherent decomposition analyses parallel branching and pipelining possibilities within the functional activities, decomposing the sequential activities into finer grain parallel activities.

The problem of multiplying a matrix by a vector is well-known and oft-studied in the field of parallel algorithms. For example, Zubair and Madan [Zub87] provide an efficient method for solving them using a systolic array, and Grigg, Miguet, and Robert [Gri90] provide an optimal method for solving symmetric matrix-vector products on a distributed memory machine. The latter also illustrates the well-known method of distributing the rows or columns of the matrix amongst the processors (distributed or shared memory).

The last technique is an easy way to exploit parallelism in the calculation of $K(k)$ in Equation (4.6).

Furthermore, the calculation of $d(k)$ in Equation (4.7) can be folded into this distribution of the matrix-vector product by introducing a term α which is calculated separately by each of four processors:

$$\alpha(k) = \varphi^T(k) * P(k-1) * \varphi(k) = K^T(k) * \varphi(k) \quad (4.12)$$

so that:

$$d(k) = \lambda + \alpha(k) \quad (4.13)$$

where λ is the least squares forgetting factor.

The $d(k)$ value may then be derived by a single processor adding all of the separate α results as follows:

$$d(k) = \lambda + \alpha_1(k) + \alpha_2(k) + \alpha_3(k) + \alpha_4(k) \quad (4.14)$$

where,

$$\alpha_i(k) = \varphi^T(k) * P_i(k-1) * \varphi(k) = K_i^T(k) * \varphi(k) \quad (4.15)$$

Diagrammatically, processor 'i' computes the parts of K and α as shown in Figure 4.12.

As discussed for the 3 processor version, these computations are duplicated on each processor for the second link. The choice of four processors was natural since each link matrix has dimension 4×4 , so that one row for each link is allocated to each processor.

Fewer processors could be chosen if the communications overhead outweighed the

computation time gain, but the choice would be restricted to 2, 4, or 8 processors to ensure balanced computational complexity.

Figure 4.12 - Diagrammatic View of MVP Row Operations

The next inherent decomposition to investigate concerns the *Covariance Update* activity.

The equation being solved is Equation (4.8):

$$P(k) = \frac{1}{\lambda} \left[P(k-1) - \frac{K(k)K^T(k)}{d(k)} \right]$$

and is diagrammatically illustrated in Figure 4.13. The figure shows how the row operations would be carried out for processor i . Again the choice is clear that 2, 4, or 8 processors are required. Since this computation shares the same processors as the *Matrix-*

Vector Product activity, the matrix P computation was distributed equally to the four processors.

The diagram illustrates the covariance update row operation. It shows a large matrix P on the left, which is equal to $\frac{1}{\lambda}$ times the sum of two terms inside large curly braces. The first term inside the braces is a matrix P multiplied by $\frac{1}{d}$. The second term is a matrix K multiplied by K^T . The matrix K is shown as a small block within a larger matrix structure, and K^T is shown as a shaded rectangular block.

$$\begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}_P = \frac{1}{\lambda} \left\{ \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}_P - \frac{1}{d} \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}_K \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix}_{K^T} \right\}$$

Figure 4.13 - Diagrammatic View of Covariance Update Row Operations

Table 4.9 provides the new activity to processor allocation. Processors 1 and 2 contain the same activities as in the three processor version, and processors 3 to 6 now contain the row operations of the *Matrix-Vector Product* and *Covariance Update* activities.

Synchronization of these separate activities now becomes an issue, since the processors do not execute in exactly the same period. For the *MVP* activity, all processors require access to the regression vector (φ) and the covariance matrix (P). Thus a synchronization point was needed at the start of the *Matrix-Vector Product* activity to ensure the *Covariance Update* is completed. Another synchronization point was required after completion of the α calculations so that a single processor (processor 3) can sum the final $d(k)$ result for each link. The final synchronization point required was at the start of the *Covariance Update* activity to ensure each processor has access to the correct value of $d(k)$. This is shown in

Figure 4.14. Note that the figure has been simplified for clarity by showing data access arcs for only the MVP1 and Cov4 activities. The other activities actually have similar arcs.

Figure 4.14 - Synchronization of Inherent Parallelism

Allocation of the memory for the K and P variables could be either distributed by rows amongst the processors or restricted to single processors for shared remote access. The latter was preferable in this case since the K vector is accessed by processors 2, 3, 4, 5, and 6.

Processor	Allocated Activities	Expected Utilization
1	Sample, Feedback Law, Feedforward Law, Summing	80.7%
2	Phi Update, Prediction Error, Parameter Update	41.8%
3	Matrix-Vector Product (1), Covariance Update (4)	82.2%
4	Matrix-Vector Product (2), Covariance Update (3)	77.9%
5	Matrix-Vector Product (3), Covariance Update (2)	77.9%
6	Matrix-Vector Product (4), Covariance Update (1)	85.7%

Table 4.9 - Six Processor Activity Allocation

Under this allocation, the expected cycle rate was determined by finding out the time complexity of the new matrix operations. The first step was to determine the communication penalties imposed by distributing the calculations of K , d , and P . Table 4.10 provides this information, showing a total communications delay of 1.223 milliseconds. This figure obviously implies significant bus usage, possibly around 50% utilization. To determine the actual cycle rate and bus utilization to be expected, the timings for the matrix operation sub-activities are obtained next.

Accessing Activity	Bus Activity	Location of Data	Data	Data Size	Delay [μ s]	Corresp. Equation
P3 Total				88 dw, 12 ints	404.3	
MVP4	P6 Write P3	MVP1	sync	1 integer	2.15	-
MVP4	P6 Write P4	MVP2	sync	1 integer	2.15	-
MVP4	P6 Write P5	MVP3	sync	1 integer	2.15	-
MVP2	P4 Write P3	MVP1	done	1 integer	2.15	-
MVP1	P3 Read P4	MVP2	α_2	2 dwords	8.6	4.13
MVP3	P5 Write P3	MVP1	done	1 integers	2.15	-
MVP1	P3 Read P5	MVP3	α_3	2 dwords	8.6	4.13
MVP4	P6 Write P3	MVP1	done	1 integers	2.15	-
MVP1	P3 Read P6	MVP4	α_4	2 dwords	8.6	4.13
MVP2	P4 Read P3	MVP1	K	12 dwords	51.6	4.6, 4.12
MVP3	P5 Read P3	MVP1	K	12 dwords	51.6	4.6, 4.12
MVP4	P6 Read P3	MVP1	K	12 dwords	51.6	4.6, 4.12
MVP1	P3 Read P6	MVP4	P_1, P_5	8 dwords	34.4	4.6
MVP2	P4 Read P6	MVP4	P_2, P_6	8 dwords	34.4	4.6
MVP3	P5 Read P6	MVP4	P_3, P_7	8 dwords	34.4	4.6
Cov1	P3 Write P4	Cov2	sync	1 integer	2.15	-
Cov1	P3 Write P5	Cov3	sync	1 integer	2.15	-
Cov1	P3 Write P6	Cov4	sync	1 integer	2.15	-
Cov1	P3 Read P6	Cov4	P_1, P_5	16 dwords	68.8	4.8
Cov2	P4 Read P6	Cov4	P_2, P_6	16 dwords	68.8	4.8
Cov3	P5 Read P6	Cov4	P_3, P_7	16 dwords	68.8	4.8
Cov2	P4 Read P3	Cov1	K, d	24 dwords	103.2	4.8
Cov3	P5 Read P3	Cov1	K, d	24 dwords	103.2	4.8
Cov4	P6 Read P3	Cov1	K, d	24 dwords	103.2	4.8
Total				186 dw, 9 int	1223	

Table 4.10 - Expected VMEBus Activity for 6 Processor Allocation

To determine the *MVP* activity complexity, the work reduction that occurs between the three processor and six processor allocations is analysed. In the three processor allocation, the *MVP* activity involved 42 floating point operations. In the six processor allocation, *MVP1* (the processor 3 implementation) executes 22 floating point operations which is approximately 60% of the computational load experienced by the *MVP* activity in the three processor version. The activities *MVP2*, *MVP3*, and *MVP4* on processors 4, 5, and 6 respectively involve only 18 floating point operations, experiencing about 50% of the load of the three processor version. In the case of the *Covariance Update* activities, *Cov1*, *Cov2*, *Cov3*, and *Cov4* on processors 3, 4, 5, and 6 respectively each involve only 40 floating point operations compared to the previous 160. With the additional overhead of the activity, an expected workload of about 33% of that expected in the three processor implementation was then assumed for each of them.

Therefore, to predict execution timings for the six processor version, the expected timings based on estimated workload were then derived, to which the communications delays from Table 4.10 were added. Furthermore, a rough guess estimation of 5% bus contention was added on to the communications penalties. An example calculation for the *MVP2* activity is:

$$t_p = 50\% * 1693 + 86 + 5\% * (172+86) = 945 \mu s$$

where 1693 μs is the three processor version prediction, 86 μs is the extra communication delay, and 172 μs is the communication required by the three processor version. The resulting predictions are provided in Table 4.5.

Once more the Petri Net model was developed as shown in Figure 4.17 based on the three processor version of Figure 4.8. The difference between the versions is simply the addition of 3 processors in the *Matrix-Vector Product* and *Covariance Update* activities. Again the net was checked for structural boundeness, and the simulation was run to develop the Gantt Charts of Figures 4.18, 4.19, and 4.20. The resulting cycle rate is 2.36 ms, now limited by the processor 1 allocation. The charts now show much better predicted processor utilization as Table 4.9 indicates. The predicted bus utilization is:

$$\rho_6 = \frac{1223 \mu\text{s}}{2360 \mu\text{s}} = 51.8\%$$

The critical path is now shared between the matrix operations on processors 3 to 6 and the *Sample-Back-Forward-Sum* activities on processor 1, since their timings are nearly identical.

4.3.5 Predicted Speedup and Efficiency

The predicted throughput improvement may be formalized in terms of speedup of the two parallel implementations over the single processor version. Figure 4.15 provides the

predicted speedup of the software as a function of the number of processors, where speedup is defined [Eag87] as:

$$\text{speedup} = S(n) = \frac{T_1}{T_n}$$

where,

T_1 = single processor execution time; and

T_n = execution time with n processors.

There are two upper bounds shown on the achievable speedup. The first is due to hardware considerations only, showing a linear speedup proportional to the number of processors available. This gives the maximum attainable speedup for memory resident applications (i.e. no input-output bounding), but is generally never reached since it does not account for communication time and resource contention. The second bound is the software bound (also known as *average parallelism*), which is the ratio of the total computation time required (i.e. the sum of the service demands of the activities) to the computation time of the longest sequential path in the activity precedence graph (see Figure 4.2), given an infinite number of processors. This bound reflects the simple fact that the computation takes at least as long as its longest part. Note that the software bound does not consider hardware restrictions.

In this case the longest path is approximated by the six processor implementation timing of the *Phi Update*, *Matrix-Vector Product*, and *Covariance Update* activities. This totals

approximately 2220 μ s, and the total service demand (i.e. the sum of all the activity demands) was 8950 μ s, yielding a software speedup bound of 4.03. Note that if linear speedup were achievable, the intersection of the two bounds (at approximately 4 processors) would give the maximum number of processors that could achieve 100% utilization since parallelism is limited by the software bound.

Figure 4.15 shows that the predictions imply that the implementation speedups will approach the software bound, but not linearly like the hardware bound. In fact, the implementations can only approach the software bound asymptotically at best due to the communications and contention overhead not considered in the bound. Furthermore, the predicted efficiency of the implementations, shown in Figure 4.16 and given by,

$$E(n) = \frac{S(n)}{n}$$

will quickly decrease as the number of processors increases, resulting in inefficient use of the available processors. Thus the tradeoff in parallel implementation is between decreasing processor utilization and increasing performance. Also note that the speedup can be expected to decrease and fall away from the software bound with large numbers of processors since the contention overheads will increase with the number of processors.

Figure 4.15 - Predicted Speedup with Upper Bounds

Figure 4.16 - Predicted Efficiency

Figure 4.18 - Six Processor Gantt Chart - Detailed

Figure 4.19 - Six Processor Gantt Chart - Detailed (continued)

Figure 4.20 - Six Processor Gantt Chart - Overview

4.4 Implementation Details

Having defined the activity to processor allocation and implemented a sequential version of the software, all that remains is the modification to convert to multiprocessing.

4.4.1 Design for Flexibility

The Unison operating system supports the C language alone at this point, making language choice a moot point. Since one of the goals of this thesis is to provide a flexible testbed for future experimentation, an effort was made to ensure the code is maintainable. The code maintains the modular structure of the ASN from which it was derived, so each activity is coded separately as a module, with variables hidden within procedures where practical. Only variables shared between activities co-resident on a processor are exported.

In an ideal environment, all of the activity implementations would be written with no assumption made about processor co-residency with other activities. This goal of transparency in implementation is part of the Rapid Prototyping methodology, where one tries to provide flexibility in allocation. Unfortunately, Unison does not provide any kernel utilities in support of shared memory model implementations other than messaging tools for passing pointers. It is possible to approach transparency for the integration stage, however. With careful utilization of *define* flags within C, various task allocations can be implemented using a single set of source code. This approach was taken in the implementation of this software, with each of the activities coded for communication with

its precedence graph neighbours using pointers defined either at compilation or by exchanging messages during initialization. The choice is determined by the *define* flags included with the file *globdefs.h* and is implemented in the initialization code for each activity. Furthermore, the *Main* procedure code for each allocation uses the flags for deciding which activities to include in the cyclic scheduling loop. Thus the code as implemented is extremely flexible in terms of reallocating activities as the problem is changed by modification or addition of activities.

4.4.2 Synchronization

As previously discussed, the two things to avoid when designing a hard deadline system on a multitasking multiprocessor are excessive context switching and bus saturation. Synchronization of tasks on different processors is characteristically the area where designers are prone to use context switching, since a synchronization point can be a good place to switch contexts and do other work while waiting for other processors to catch up.

Within this research, the multitasking and messaging capabilities of the Unison operating system have not been employed in order to avoid the penalty due to context switching between tasks on the same processor and because the message passing send-receive-reply duration is of the same order as the desired sampling period. Thus synchronization is based on semaphores.

To avoid excessive contention on the VMEBus, activities busy-waiting on semaphores always do so only on locally defined memory variables. The activities that the busy-

waiting activity is awaiting always write across the bus only once to signal completion. This technique works without causing excessive contention since the writer usually need only access the bus once because remote memory access has priority over local memory access. Also, all such semaphores are updated in a single bus cycle since they are implemented with 16-bit integers (an atomic transaction over the VMEBus).

Synchronization within the implementation always occurs between the *send* sub-activity of one activity and the *receive* sub-activity of one or more other activities. An example of the synchronization method is as follows:

- processor A finishes working and busy-waits on local semaphore *X_rdy* waiting for data *X* from processor B;
- processor B finishes working and copies its working data *X* to a local copy *public_X*;
- processor B increments the processor A semaphore *X_rdy* by reading and writing across the bus;
- processor A reads the new *X_rdy* value and exits the busy-wait;
- processor A continues working and accesses the *public_X* data from processor B across the bus.

Note that this semaphore method does not provide robust mutual exclusion. A concern when implementing this method without mutual exclusion is that one processor may access data from another processor that is still working on the current cycle and therefore presents old data in its local copy. This possibility is avoided in this system through the use of counting semaphores. The spin-waiting processor must see the local semaphore

incremented before accessing the remote data. These semaphores are never decremented. 16-bit wraparound is not a problem either, since the processors must remain in lock-step so that the wraparound will occur in the same sampling cycle on all processors involved.

The opposite concern without mutual exclusion protection is the possibility of a writer overwriting public data before all readers have finished reading it. This problem is avoided since the software is cyclic and each activity is prevented from overwriting its output until it has received the next cycle's input. The close coupling and cyclic nature ensures that activities execute in the proper order.

4.4.3 Initialization

The initialization process for each activity consists of three parts:

- 1) instantiation and initialization of the local variables and pointers;
- 2) messaging to set up remote variable pointers (if necessary);
- 3) sending any initial data required for the first cycle by other processors.

The instantiation is accomplished by using the Unison memory allocation function, `_getvec`. Memory allocated by `getvec` is then treated as arrays or matrices as required. For shared data structures, the originating activity allocates a working copy space and a shared copy space, and both are initialized to the startup conditions (usually to 0's). All arrays and matrices, whether local or remote, are referenced by pointers. Local semaphores are referenced directly whereas remote semaphores are referenced by pointers.

The discussion of local and remote here refers to references internal or external to the referencing activity. The remote data may belong to an activity actually allocated to the same processor or to an activity allocated to another processor. After all local variables are initialized, the remote pointers must be properly addressed. This is done in the next phase of the initialization, either by accessing exported variables from activities on the same processors or by messaging with activities on other processors. Note that messaging cannot be used between activities allocated to the same processor due to the single task structure employed. The blocking characteristic of the *_send* primitive excludes this possibility. Exported variable referencing is familiar to most programmers, but messaging between processors may not.

To initiate messaging capability, each activity registers its name with the Unison *_Directory* task using the *_register_name* function, passing the *task id* obtained from the *_my_id* function. Registering the name allows other activities to determine the *task id* and thus direct messages to that task. Therefore, the next step is to access the *task id* of the activity to be messaged via the Unison function *_lookup_name* which, given the activity name, returns the *task id*. Since the other activity may not yet have initialized itself, the activity must continue attempting to look up the name until the request succeeds.

Once the *task id* is known, a message is composed providing an identifier describing the sending activity and the addresses of its local semaphores that the receiving task will write to remotely when the associated data is available. The activity then sends the message using the Unison *_Send* function, which will block the task until a reply is received. The

receiving activity must execute the Unison *_Receive* function which blocks until the message is received, then set its remote semaphore pointers up with the addresses passed in the message. The receiving activity then composes a response message providing the addresses of the data copy variables that will be remotely accessed by the originating activity. This message is then returned to the originator using the *_reply* function, and the receiving activity then continues. The sending activity is then unblocked from the *_Send* function, and sets up its remote data pointers with the addresses provided in the reply message.

In this manner, the shared memory addressing is initialized. There is a caveat to this initialization process, and that is the sequencing amongst multiple processors. It is extremely simple to deadlock in this initialization phase because the *_Send* and *_Receive* functions block the task until they are completed. Furthermore, when activities share the same processor (and thus the same task), the addressing between them must be completed after variable instantiation. To solve these problems it is necessary to analyse the messaging requirements so that a sequential initialization schedule can be devised that will allow all instantiation to take place before the corresponding addresses are required.

This scheduling is solved by starting from the first activity in the control cycle, in this case the *Sampling* activity. Since it requires only the hardcoded A/D converter addresses, its variables and pointers may be initialized without requiring any interaction with other activities. Next the activities that access the *Sampling* activity data may be at least partially initialized. These include the *Feedback Controller* activity, the *Phi Update* activity, and the *Prediction Error* activity. The *Feedback Controller* activity also requires access to the

control output data of the *Summing* activity, so the instantiation section of that activity should precede the initialization of the *Feedback Controller*. The *Summing* activity cannot be completed, however, since it also requires access to the *Feedback Controller* output.

Following this logic, the initialization of each activity is broken up into instantiation and (possibly multiple) messaging parts, and a sequential ordering is derived. This ordering is then maintained regardless of the processor allocation by placing it into the Main procedure of each processor's single task initialization code.

The final stage of initialization involves sending the initial data required to initialize the algorithm. This stage provides the functionality of the 'Start' transition of the Petri Net models in Figures 4.8 and 4.15, where the initial data required are identified by the initial token placement in the net. This stage is accomplished by startup calls to the appropriate sending sub-activities.

4.5 Summary of Parallel Design Process

This section summarizes the process followed throughout the design of this parallel software system. The techniques discussed work well in this robotics control problem and are equally applicable to other small-scale process control problems. As the size and complexity of the problems increase, the methods described become more useful in allowing the design team to grasp the entire problem from a functional viewpoint while applying parallelism where it will gain the most benefit.

The first, and perhaps most important step in the design process is the **algorithm decomposition**. This step requires an analysis of the problem structure and identification of the individual activities involved. The 'atomicity' of the activities defines the eventual task granularity that will be available for allocation to processors later in the design.

Following the decomposition, an **analysis of the dataflow constraints** is required. This step involves detailed review of the equations that make up the control algorithm and how they must interact with each other. It is important to take a flexible viewpoint at this stage so as to ensure that the fastest throughput will be attainable. This may require, for example, that parameter estimates a cycle old be used instead of waiting for the current cycle's results, or that the estimator run at a different rate than the controller. Small sacrifices that do not have large effects upon system stability may be exploited to great advantage. This constraint analysis will result in a dataflow precedence graph which can then be combined with the activities previously defined to provide an Activity Sequence Network model. This model should reflect the full functional parallelism available.

The next step in the system design is to select the hardware architecture, if this is not imposed, and to determine the **architectural constraints**. Choice of hardware has only briefly been discussed in this research (Section 3.1), but should depend on the throughput required, the number of manipulator links to be controlled (i.e. the processing complexity), the number of parameters to be identified (i.e. the communication load), and the parallelism exhibited by the problem. The architectural constraints provide a weighting for the computation versus communication when analysing for the expected performance of the

system. They also provide enough information to determine what communication method will be employed for the parallel implementation.

Next is the development of a **sequential implementation and its analysis**. This step involves development of the majority of the functional code from the ASNs previously derived. It is important to keep reusability in mind. The code should be targetted towards parallel implementation and thus should be developed in the parallel system environment if possible. The code should be tested for proper functionality, then analysed for determining individual activity complexity. This analysis may be performed by determining instruction sequence timings, but ideally some form of performance monitor will provide more accurate results faster and easier. The Unison development environment is ideal for this design stage, since the Unison sequential emulator can run under Unix allowing profiling with the Unix *gprof* tool. In the case where the profiling is determined on a system other than the target processor, the architectural differences between the processors must be factored in to obtain reasonable performance predictions.

The following design stage involves the **allocation of activities to multiple processors**. This is the stage in which the designer determines how much of the functional parallelism exhibited in the ASN should be exploited. First, the sequential paths should be derived from the ASN. Next, using the predicted activity timings, the limiting duration of these sequential paths should be estimated. The most time complex sequential path should then be minimized while maintaining the ASN structure, and this should be allocated to one processor. The other processors now take on the role of supporting processors, and the designer should allocate the remaining sequential paths to these

processors in such a way that data is always available without delay for the original processor's activities, if possible. This allocation is then analysed for the communications penalties imposed by the distribution, due to interprocessor access delays, synchronization, and possibly resource contention. These penalties are then added to the corresponding sending and receiving activity durations. A deterministic Petri Net model, if available, can then be extremely useful in confirming the throughput achievable, and in checking for boundedness of the design. The Gantt Chart tool developed provides further aid in finding system bottlenecks uncovered by the Petri Net model. Usually this entire stage will be iterated as required to obtain allocations closer to the optimal.

Once an acceptable allocation has been found using functional parallelism, it may be improvable by the **exploitation of inherent parallelism** within the critical sequential path. Matrix operations are a prime candidate as illustrated within the example system. Any extra communication penalties are again analysed and added, and the Petri Net and Gantt Chart analyses can be repeated. Again this and the previous stage may be iterated to obtain more optimal allocations.

Finally the **implementation details** must be considered. The design team must determine how to implement synchronization and communication between the processors so as not to unnecessarily penalize communications. Any initialization issues such as ordering must also be considered. Above all, the implementation must be designed in a flexible way to allow interchange of modules along with possible reallocation of activities to processors. After all of these issues have been addressed, the parallel implementation

can be implemented and tested. If further performance improvements are required, the allocation design stages may be revisited.

5.0 Results of Parallel Implementation

In this chapter the experimental performance and control results are discussed. The discrepancy between predicted and actual performance is studied with the objective of providing greater insight into real-time throughput predictability. The control system improvement is also discussed, illustrating the improvement gained by the increased throughput attained. Finally, further possible improvements to the implementation are discussed.

5.1 Real-Time Performance

Once the parallel software designs had been coded, they were augmented with compilation-conditional calls to the Unison Event Monitor logging facility. These calls were to log user-defined events at the sub-activity level so that detailed timing measurements could be obtained. The results are provided in Tables 5.3a, 5.3b, and 5.4. The corresponding Gantt Charts appear in Figures 5.5 to 5.10.

The actual performance figures in Tables 5.3a, 5.3b, and 5.4 give ranges of timing as reported by the Event Monitor. The lack of total determinism results from the Event Monitor's resolution of 50.5 μ s and its intrusiveness. Clearly the resolution was not sufficient to monitor the sub-activities since most of them are in the 1 to 2 step (50.5-101.0 μ s) range. In fact, a 0 step duration was never achieved during testing, even when a null activity was measured (i.e. a call to an empty procedure). Furthermore, the intrusiveness

of the monitor when 8-12 events were logged each cycle was proven by comparing the timing obtained when only one activity duration was logged during a run against the timing obtained when multiple sub-activity durations were logged. The single event logging showed that the cycle period decreased by approximately 10%-25%. For this reason, each of the sub-activity durations were monitored separately to obtain the final performance figures. All of these problems indicate that the individual sub-activity timings obtained were very approximate with a likely margin of error in the 25%-50% range. However, the timings of the actual activities and the entire cycle period can be assumed to be much more accurate (likely $\pm 5\%$) as the intrusiveness plays a much smaller part.

5.1.1 Comparison Between Predicted and Actual Results

Taking the error margins into account, the results for the single processor target implementation closely resemble the predicted values, where the obtained cycle time was 6.77 milliseconds versus the predicted 6.85 milliseconds implying an overestimation of about 1.2%. The three processor implementation achieved a cycle period of 4.34 milliseconds against the predicted period of 4.09 milliseconds, an underestimation of about 250 μs or 6.1%. Finally, the six processor implementation achieved a 3.08 millisecond period, an underestimation of 720 μs or 30.5% versus the predicted 2.36 milliseconds.

A major reason for the discrepancy between the estimated and actual performance was the resource contention approximations used. Contention for the VMEBus results in increased function execution times due to the delays in gaining access to the bus. Contention for memory leads to both increased bus utilization due to retries and to increased function

execution times. Examples of the increased function execution times can be seen by comparing predicted differences in Tables 4.4a, 4.4b, and 4.5 with those obtained in Tables 5.3a, 5.3b, and 5.4. For example, the *update_parms* sub-activity within the *Theta* activity shows a predicted increase of 69 μ s between the sequential and the multiprocessor versions. The actual increase was about 100 to 200 μ s which shows substantial contention delays during the remote access of the K and d variables. Another example is the *calculate_kd* sub-activity within the *Matrix-Vector Product* activity, where a predicted increase of 172 μ s differs from the measured three processor result of 303 μ s. These results show the need to provide improved modelling of system resource contention at the prediction stage.

Bus contention, the percentage increase in bus access wait time, can be better approximated by methods such as Stochastic Petri Nets, Markov Chains, and Queuing Networks as shown in [Mar83]. In that paper the authors give examples of contention values in 2, 3, and 4 processor systems connected by 1 to 4 busses under varying bus traffic. Although their results are based on non-deterministic communication patterns, they at least provide enough insight to show that the contention to expect with the VMEBus testbed probably makes up a significant portion of the discrepancy between actual and predicted performance. Table 5.1 provides the approximate bus contention predicted for two single bus examples given by Marsan, Balbo, Conte, and Gregoretti in [Mar83]. The values in Table 5.1 were obtained by determining the percentage loss in processing power for the three and four processor single-bus queuing network simulations provided by Marsan et al.

The bus utilization figures for the VMEBus execution were measured with a logic analyser as 10.2% (442.0 μ s per cycle) with 3 processors and 51.2% (1580 μ s per cycle) with 6 processors. Estimating from the predictions given by Marsan et al, the VMEBus contention expected would be approximately 10% for the three processor implementation and approximately 50% for the six processor implementation. This would result in predictions increased by 10% of 404 μ s, or 40 μ s for the three processor implementation, and by 50% of 1223 μ s, or 612 μ s for the six processor implementation.

Memory contention is just as difficult both to predict and to measure. The estimated actual memory contention is the percentage of extra bus time per cycle above the predicted values calculated in Tables 4.8 and 4.10. The predicted bus access time for the three processor version was 404 μ s, but the actual access time was 442 μ s, an increase of 9.4%. The six processor predicted bus access time was 1223 μ s, but the actual obtained was 1580 μ s, implying an increase, or memory contention, of 29.2%. Table 5.2 gives these results, which imply the three processor version was further delayed by 9.4% of 404 μ s, or 38 μ s, and the six processor version was further delayed by 29.2% of 1223 μ s, or 357 μ s.

Applying these new bus and memory contention figures to the estimates from Sections 4.3.3 and 4.3.4 results in an extra 78 μ s per cycle for the three processor implementation and an extra 969 μ s in the six processor implementation. This adjustment reduces the three processor discrepancy to an underestimation of 172 μ s (4.1%) and the six processor discrepancy to an overestimation of 249 μ s (7.5%). These new predictions are superior to those obtained in Chapter 4, and show the requirement for accurate modelling of resource contention during the prediction phase. They also show that the uniform distribution

assumed by Marsan et al for modelling of bus contention is at least roughly applicable to this robotics problem.

Bus Utilization	3 Processor Model Bus Contention	4 Processor Model Bus Contention
5%	5%	5%
10%	10%	10%
20%	19%	20%
30%	27%	29%
40%	35%	36%
50%	40%	43%
60%	47%	48%
80%	55%	58%
100%	60%	63%

Table 5.1 - Bus Contention for Varying Utilizations [Mar83]

Number of Processors	Measured Bus Utilization	Estimated Memory Contention
1	0.45%	-
3	10.2%	9.4%
6	51.2%	29.2%

Table 5.2 - Estimated Memory Contention from VMEBus Utilization

A further possibility for discrepancy between the predicted and actual performance is due to indeterminacy in the system that was not modelled in the prediction. Indeterminacy arises in many places in the model of the system used, from conditional loops in the code to memory refresh cycles causing contention on the internal processor card buses. The code, for example, does contain conditional statements in the *Feedforward Controller* activity to

determine whether or not to apply feedforward control depending upon the convergence of the model parameters. Depending upon the trajectory used, this condition will be different between a sequential implementation using the latest estimates versus a multiprocessor implementation using cycle-old estimates.

It is interesting to note from close analysis of the Gantt Chart results that the difference in activity timings cause a limited ripple effect in the dependent activities, resulting in extended synchronization waiting times. The effect is illustrated between Figures 5.1 and 5.2, where the predicted and actual six processor implementation activities, respectively, are shown in Gantt Chart-like style. These figures have been approximated from the Gantt Charts of Figures 4.16, 4.17, 5.8, and 5.9. The difference between the predicted and actual cycles is that the predicted cycle included time spent in the *Feedforward Controller* activity waiting for the previous cycle's *Update Parameters* activity, whereas the actual implementation shows that the parameters are available before the controller executes. This analysis indicates that, depending upon the algorithm in use, the accuracy of the predicted timings may be critical in determining the final throughput. Small errors to either side may cause ripple effects that skew the results due to required synchronization.

Figure 5.1 - Predicted Six Processor Activity Cycle

Figure 5.2 - Actual Six Processor Activity Cycle

5.1.2 Measured Speedup and Efficiency

Following the method given in Section 4.3.5, the speedup obtained by the two parallel implementations over the single processor version is plotted in Figure 5.3 (cf. Figure 4.15) and the efficiency is plotted in Figure 5.4 (cf. Figure 4.16).

The bounds and attained speedups shown in Figure 5.3 are very close to the predicted results of Figure 4.15. The software bound is slightly higher at a speedup of 4.14 versus the predicted 4.03, but the difference is of little significance given the problems associated with obtaining the service demand measurements (i.e. the Event Monitor resolution). The increased service demands of the activities due to resource contention caused the drop in the attained speedup of the six processor implementation from a predicted 2.9 to the actual 2.2.

The measured efficiency as shown in Figure 5.4 also reflects the resource contention problem, where the six processor efficiency is down to 37% from an expected 48%.

Figure 5.3 - Measured Speedup with Upper Bounds

Figure 5.4 - Measured Efficiency

This research has shown that estimating performance of a multiprocessor-based real-time control system is definitely possible but involves several tricky issues. Modelling resource contention and system indeterminacy form the greatest obstacles since determining appropriate models and parameters is not necessarily intuitively obvious. The problem is exacerbated by the "ripple effect" that inaccurate activity timing estimations may have on expected throughput.

However, with understanding of the software requirements and hardware architecture, estimations adequate for most planning purposes may be obtained from simple models relatively easily. Lessons learned from this prototyping activity show the importance of modelling resource contention along with the models of the basic activities and communications. Including these factors early in the design allows for very close estimations of the achievable performance.

Activity	Sequential Timings (μ sec)	3 Proc. Timings (μ sec)	6 Proc. Timings (μ sec)
Sample	404.0-454.5	505.0-606.0	454.5-606.0
- read v_1	101.0	101.0	50.5-101.0
- read v_2	101.0	101.0	101.0 - 151.5
- read y	101.0	101.0	101.0
- send old y	101.0	101.0-151.5	101.0-151.5
- send y_v	101.0-151.5	101.0-151.5	101.0
Feedback Law	404.0-454.5	454.4	404.0-454.5
- read trajectory	101.0-151.5	101.0-151.5	101.0-151.5
- receive y_v	50.5-101.0	50.5-101.0	50.5-101.0
- calculate u_b	353.5-404.0	353.5-404.0	353.5-404.0
- send u_b	101.0	101.0	101.0
Feedforward Law	909.0-1010.0	909.0-1111.0	909.0-1111.0
- receive e	50.5-101.0	50.5-101.0	50.5-101.0
- receive θ	50.5-101.0	50.5-101.0	50.5-101.0
- read trajectory	151.5	151.5	151.5
- calculate u_f	303.0-404.0	404.0-454.5	404.0-454.5
- check whiteness	202.0-252.5	202.0-252.5	202.0-252.5
- check δ	353.5-404.0	454.5-505.0	454.5-505.0
- send u_f	101.0	101.0	101.0
Sum	252.5-303.0	252.5-303.0	252.5-303.0
- receive u_b	50.5-101.0	50.5-101.0	50.5-101.0
- receive u_f	50.5-101.0	50.5-101.0	50.5-101.0
- calculate u	202.0	202.0	202.0
- write to robot	101.0	101.0	101.0
- send u	101.0	101.0	101.0
Phi	202.0-252.5	303.0	303.0-353.5
- receive u	101.0	50.5-101.0	50.5-101.0
- receive y	101.0	101.0	101.0
- update ϕ	101.0-151.5	202.0	202.0
- send ϕ	101.0-151.5	202.0	202.0-252.5
Pred. Error	454.5-505.0	505.0-555.5	555.5
- receive ϕ	50.5-101.0	50.5-101.0	50.5-101.0
- receive y	101.0	101.0	101.0
- receive θ	50.5-101.0	50.5-101.0	50.5-101.0
- calculate error	404.0-454.5	404.0-454.5	404.0-454.5
- send e	101.0	101.0	101.0
Theta	505.0-555.5	3383.5-3484.0	1969.5-2070.0
- receive e	50.5-101.0	50.5-101.0	50.5-101.0
- receive kd	50.5-101.0	2676.5-2828.0	1313.0-1464.5
- update parms	404.0-454.5	555.5	555.5-606.0
- send θ	101.0-151.5	151.5-202.0	151.5

Table 5.3a - Actual Performance (in microseconds)

Activity	Sequential Timings (μ sec)	3 Proc. Timings (μ sec)	6 Proc. Timings (μ sec)
Mat-Vec Product	1565.5-1717.0	1969.5-2020.0	see Table 5.4
- receive P	50.5-101.0	50.5-101.0	
- receive phi	50.5-101.0	101.0	
- calculate kd	1565.5-1616.0	1868.5-1919.0	
- send kd	151.5-202.0	151.5-202.0	
Covariance	2272.5-2474.5	2474.5-2575.5	
- receive kd	50.5-101.0	101.0	
- update covariance	2070.5	2121.0-2170.5	
- send P	404.0-454.5	404.0-454.5	

Table 5.3b - Actual Performance (in microseconds)

Activity	Processor 3 Timings [μ s]	Processor 4 and 5 Timings [μ s]	Processor 6 Timings [μ s]
Mat-Vec Product	1868.5-2020.0	1313.0-1363.5	1161.5-1212.0
- receive P	656.5-757.5	505.0-606.0	101.0-151.5
- receive phi	101.0-151.5	101.0	101.0-151.5
- calculate kd	959.5-1060.5	909.0-1060.5	1010.0-1111.0
- send kd	151.5-202.0	0	0
Covariance	1060.5-1111.0	1515.0-1565.0	1818.0-1919.0
- receive kd	101.0-151.5	404.0-505.0	252.5-353.5
- update covariance	959.5-1060.5	1111.0-1161.5	1262.5-1363.5
- send P	101.0-151.5	101.0	404.0-454.5

Table 5.4 - Six Processor MVP & Cov Activity Actual Performance

(Note: 'Processor N Timings' refers to the activity on processor number N)

Figure 5.5 - Three Processor Gantt Chart (Detailed)

Figure 5.6 - Three Processor Gantt Chart (Detailed) - continued

Figure 5.7 - Three Processor Gantt Chart (Overview)

Figure 5.8 - Six Processor Gantt Chart (Detailed)

Figure 5.9 - Six Processor Gantt Chart (Detailed) - continued

Figure 5.10 - Six Processor Gantt Chart (Overview)

5.2 Control Results

The parallel implementation reduced the control system sampling period from 6.8 milliseconds (147 Hz) to 4.4 milliseconds (227 Hz) using three processors and to 3.1 milliseconds (323 Hz) using six processors. This section discusses the effect that this increase in sampling frequency had on parameter convergence, and presents the control results obtained.

An increase in sampling rate is not always beneficial for an adaptive controller. Very short sampling intervals can result in instability of sampled data models in various ways, such as leading to sensitivity in the model determination or changing the system to be non-minimum phase (see [Sod89] and [Ast89]). Also, if instability is avoided, increasing the sampling rate will not provide any appreciable improvement in the quality of control. However, increasing the sampling rate will increase the amount of information available to the estimator, specifically in the higher frequency range. This may be beneficial in final convergence of the parameters, but is dependent upon the unmodelled dynamics and noise in the system.

The metrics used to measure control quality include the average positioning accuracy in trajectory tracking and the maximum absolute tracking error. These metrics illustrate how good the control system is at tracking minute changes in the manipulator trajectory, or how accurate such a system can be expected to be.

Due to the adaptation algorithm and the resulting continuous improvement in parameter estimations, the above metrics are variable depending on the time period over which measurements are taken. For example, before the parameters have converged the PID controller alone is used to control the manipulator. After convergence, the estimations continue to improve, providing improving quality of control. Thus another metric used for measurement of control quality is the rate of convergence of the parameters. All of these metrics are shown in the following subsections to provide some insight into the quality of control attained.

5.2.1 Parameter Convergence Observations

Figures 5.11 through 5.26 each show the parameter errors for each of the single processor (solid line), three processor (dashed line), and six processor (dotted line) implementations. Each figure illustrates the varying convergence of a single parameter over the three implementations. The figures are grouped such that: the first four (Figures 5.11 through 5.14) represent the errors in the four parameters of link 1 under the slow trajectory; the following four (Figures 5.15 through 5.18) represent the errors in the four parameters of link 2 under the slow trajectory; the next four (Figures 5.19 through 5.22) represent the link 1 parameter errors under the fast trajectory; and the final four (Figures 5.23 through 5.26) represent the link 2 parameter errors under the fast trajectory.

The parameter error graphs show some interesting characteristics. As would be expected, the fast trajectory is providing more information to the estimator than the slow trajectory, so the parameters in Figures 5.19 through 5.26 converge much better than their slow

trajectory counterparts. This is especially obvious for the less massive link 2, where convergence has not yet taken place after 14 seconds during the slow trajectory (see Figures 5.15 and 5.16) and convergence occurs at around 1.5-2.5 seconds during the fast trajectory. Even the seemingly identical convergence profiles for parameter a_1 in Figures 5.11 and 5.19 show the improvement in final accuracy after 2.5 seconds. This is true for all three implementations, regardless of the sampling frequency.

The differences between the three implementations is marked in all of the graphs, and appears in two stages. In the first stage, the three and six processor versions show larger parameter errors than that of the single processor version. This seemingly counterintuitive observation is a direct result of an increase in the number of samples taken. When the sampling period is short, the trajectory change between subsequent samples is smaller than when the sampling period is long, thus providing less information at each cycle. The shorter period also implies more accuracy in prediction, resulting in a smaller prediction error from Equation (4.9),

$$e_{\theta}(k) = y(k) - \hat{\theta}^T(k-1)\varphi(k) \quad (5.1)$$

at each cycle. Then, from equation (4.8),

$$P(k) = \frac{1}{\lambda} \left[P(k-1) - \frac{K(k)K^T(k)}{d(k)} \right] \quad (5.2)$$

where,

$$K(k) = P(k-1) * \varphi(k) \quad (5.3)$$

$$d(k) = \lambda + K^T(k) * \varphi(k) \quad (5.4)$$

and equation (4.10),

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{K(k)}{d(k)} * e_{\theta}(k) \quad (5.5)$$

it is clear that with more information available $P(k)$ will decrease faster and with larger prediction errors $\hat{\theta}(k)$ will converge faster.

This result is confirmed in Figures 5.35 through 5.41, where the trace of the covariance matrix $P(k)$ is plotted. Again, the solid lines illustrate the single processor implementation, the dashed lines the three processor implementation, and the dotted lines the six processor implementation. Note that the scales on these figures differ between each other in order to highlight the interesting points in each. Each of the graphs shows that the covariance matrices reach something approximating a steady-state condition after the first second.

After the covariance matrices cease the rapid decrease, a second stage of parameter convergence characteristics begins. The higher valued covariance matrices of the higher sampling frequency implementations provides a greater ability for new information to affect the converging parameters. The higher values are maintained through the effect of the forgetting factor, which increases the covariance matrix in periods of little or no new information, as follows:

$$P(k) = \frac{1}{\lambda} [P(k-1)] \quad (5.6)$$

Equation (5.6) is then be executed more often during periods of no information at the higher sampling frequencies, resulting in the higher peaks seen in Figures 5.37, 5.38, and 5.41.

The cumulative effect of these two stages is seen in the parameter error graphs, which show that the higher gains available in the higher frequency estimators provides more accuracy in the final estimation. This result is as expected since the higher sampling frequency yields more information over the 22 second trajectory.

Table 5.5 provides the time at which the various parameter errors went below a limit of 1.0×10^{-4} and shows that in all cases but one this limit was achieved faster under higher sampling frequencies. The one exception occurs for the b_1 and b_2 parameters for link 1 under the fast trajectory. This exception is due to the extremely rapid convergence of the parameters over 3-4 seconds. Where no time is given in Table 5.5, the parameter errors did not reach the 1.0×10^{-4} limit over the 22 second trajectory. In all these instances, the ending parameter error was better for the higher frequency implementations.

Link	Parm.	Fast Trajectory			Slow Trajectory		
		1 Proc.	3 Proc.'s	6 Proc.'s	1 Proc.	3 Proc.'s	6 Proc.'s
1	a ₁	11.6	10.7	9.2	--	--	21.4
1	a ₂	13.1	11.8	10.7	--	--	21.5
1	b ₁	2.9	3.6	4.7	5.1	3.1	2.9
1	b ₂	2.9	3.6	4.7	5.3	3.1	2.9
2	a ₁	19.8	16.7	13.6	--	--	--
2	a ₂	21.3	18.5	14.0	--	--	--
2	b ₁	6.3	6.3	3.9	--	14.0	12.1
2	b ₂	6.6	6.4	3.8	--	14.0	12.1

Table 5.5 - Seconds to Parameter Convergence Error < 1.0e-4

The results of this section show the need to determine the relationship between the sampling frequency, the initial value of the covariance matrix, the forgetting factor, and the parameter convergence rate. This is beyond the immediate research area of this thesis.

5.2.2 Trajectory Tracking Observations

Two separate factors dictate the quality of control achieved, the accuracy of the feedforward controller parameters and the control bandwidth used. Since the control bandwidth of the PID control was unchanged, any improvement is solely due to parameter accuracy improvements. The preceding section has shown that a higher sampling frequency results in faster and more accurate parameter convergence, due to the capture of more information. This section presents the effect the accuracy improvement has had on the control quality.

Figures 5.27 through 5.30 illustrate the slow trajectory tracking errors. Figures 5.27 and 5.28 show the target trajectories for links 1 and 2, respectively, and the single processor version tracking is superimposed as a dashed line. Figures 5.29 and 5.30 then show the corresponding positional tracking errors for the single (solid line), three (dashed line), and six processor (dotted line) implementations. These figures are then repeated for the fast target trajectory, where Figures 5.31 and 5.32 illustrate the target trajectories with the single processor version tracking superimposed, and then Figures 5.33 and 5.34 show the positional tracking errors for the single (solid line), three (dashed line), and six (dotted line) processor implementations.

The slow trajectory tracking errors quite clearly show the improvement in tracking obtained with the three and six processor implementations. The link 1 improvement shows up at about the 6 second mark, where the six processor implementation provides superior control over the remaining trajectory. The link 2 improvement shows up much sooner at about the 2 second mark, but there is a slight decrease in control quality just at the beginning of the stopping trajectory of the final second. Furthermore, there is a degradation in control quality for both links in the first second or two.

All of these effects are more pronounced in the fast trajectory example. The link 1 tracking error improvement shows up after about 7 seconds, and the link 2 improvement after about 10 seconds. The initial degradation in control quality is markedly worse on both links, especially link 1 between 4 and 7 seconds, and the stopping trajectory degradation is only apparent in the link 2 positional error.

Several effects are in play here. The initial and final control degradations can be partially attributed to the non-differentiable change in the commanded acceleration at the end of the startup trajectory and at the beginning of the stopping trajectory. This effect causes a slight disturbance in the control signal that takes time to be dampened. This would likely improve if the trajectories were chosen to provide 3rd order differentiability at these points.

The main effect causing the control degradation in the time before improvement is noticed (see the points marked A in Figures 5.29, 5.30, 5.33, and 5.34) is due to imperfect convergence detection. The algorithms used to detect parameter convergence allow the feedforward loop to operate under some circumstances before parameter convergence is achieved. These circumstances occur when the prediction error is very small and the parameters therefore change only slightly over several cycles. The prediction error may be small if the information received over the initial part of the trajectory can fit more than one system model. This is likely due to the lack of persistent excitation in the startup three segments of the trajectories. As observed, this results in a short-term parameter convergence to a different system model that acts similarly over the early trajectory.

Finally, the other effect causing degradation is the "bumpless transfer" problem. This is caused by the sudden addition of the feedforward signal to the feedback control signal. The addition acts as another disturbance on the system, and can actually cause the entire system to go unstable. The method used in this example was to apply only 10% of the feedforward signal during the initial cycle of convergence detection, 20% the following cycle, and increasing 10% per cycle until 100% of the signal is applied in the tenth cycle. This method smooths the disturbance, allowing the feedback controller to gradually adjust.

This "bumpless transfer" procedure occurs many times as the feedforward signal is added and removed as the parameters approach convergence. Each time, the initial disturbance results in a short-term increase in tracking errors. A few examples are marked with 'B's in Figures 5.33 and 5.34 (the results are unobservable in the slow trajectory examples).

Once the parameters have adequately approached convergence, the tracking errors show the improvement achieved. Visually regarding the error plots, it is obvious that the effect of more accuracy in the feedforward parameters resulted in maximum tracking errors of magnitude decreased by about 30% (to about ± 0.3 radians) for the three processor implementation and by about 50% (to about ± 0.2 radians) for the six processor implementation over the single processor version (at about ± 0.4 radians).

A more formal view is provided by Table 5.6, where the positional tracking percentage accuracy over the last five seconds of the trajectory is presented. The positional tracking percentage accuracy is defined by

$$\text{percentage accuracy} = \left(1 - \frac{\sum_{k=1}^N |y_d(k) - y(k)|}{\sum_{k=1}^N |y(k)|} \right) * 100$$

where,

N = number of samples in last five seconds of trajectory;

$| \dots |$ = absolute value;

$y_d(k)$ = desired position at cycle k ;

$y(k)$ = actual position at cycle k .

Trajectory	Link	1 Processor	3 Processors	6 Processors
Slow	1	98.17	98.81	99.27
Slow	2	99.53	99.58	99.64
Fast	1	98.91	99.29	99.50
Fast	2	98.88	99.04	99.10

Table 5.6 - Positional Tracking Accuracy Over Final 5 Seconds

The table shows the improvement is consistent across both links under both trajectories.

There are two caveats to the results presented here. First, the parameter accuracy improvements result from the simple plant model used and the noise-free simulation environment. This will not generally be achievable in non-ideal systems due to the presence of noise and the unmodelled dynamic effects of an actual manipulator. Secondly, the fact that the trajectory tracking errors illustrated in Figures 5.29, 5.30, 5.31, and 5.32 do not subside after parameter convergence was not expected, illustrating either a timing problem within the software, insufficient parameter accuracy, or a sluggishness within the PID control.

5.3 Further Improvements

Much has been learnt from this initial implementation of a parallel adaptive controller, specifically in the areas of optimizing the throughput and the resulting accuracy and speed. Much of what has been presented includes knowledge gained through the experimentation stage, but there are several issues that could be addressed to gain further improvement.

An area not pursued in this research is that of improving the tracking accuracy through increasing the control bandwidth by increasing the proportional gain K_p in the PID controller. As discussed in Section 2.3.4, one of the limits on the control bandwidth is the sampling frequency. Thus with an increased sample rate, the control bandwidth may be correspondingly increased.

One well-known method of improving throughput is to optimize the software code, both through compiler optimization and through hand assembling the critical parts. Such a task is usually extremely time consuming, but critical sequential code sections could be improved substantially.

Another improvement could be obtained by reworking the software design once again. For example, the rows of the covariance matrix could easily be distributed in the six processor implementation since P is not accessed by other processors. Such distribution would avoid unnecessary communication in the *Matrix-Vector Product* and the *Covariance Update*

activities. Another example is to avoid the wait by the *Phi Update* activity for its input from the *Sampling* activity. Currently it waits unnecessarily until the start of the following cycle.

Pipelining of the control software is another way to further increase the achievable throughput. The estimation routine is a likely candidate for this since the MVP-Cov activity combination forms a sequential bottleneck. To pipeline the estimation software, the *Matrix-Vector Product* activities could be executed in one cycle followed by the *Covariance Update* activities in the next. Other breakdowns are possible with a finer grain of pipeline. The problem with this method is that the time-dependent parameters, b_1 and b_2 , would then converge more slowly to a different value based on the reduced cycle rate of the estimator. From equation (2.12),

$$y(k) = 2.0y(k-1) - y(k-2) + \frac{T^2}{2H}u(k-1) + \frac{T^2}{2H}u(k-2)$$

an estimated b parameter should converge to $\frac{T^2}{2H}$, where T is the estimation cycle period. If the control system sampling rate is then T_s , the actual parameter to be used is then,

$$b_s = X * b$$

$$\text{where } X = \frac{b_s}{b} = \frac{\frac{T_s^2}{2H}}{\frac{T^2}{2H}} = \left(\frac{T_s}{T}\right)^2$$

so
$$b_s = \left(\frac{T_s}{T}\right)^2 * b$$

This pipelining method would allow the cycle period to be completely defined by the critical path formed by the controller equations, isolating the estimator. The question of whether or not improved throughput obtained by pipelining would increase the quality of control is not easy to answer. This would result in an increased sampling rate and thus the ability to increase the control bandwidth. The tradeoff is a decreased convergence rate and final accuracy in the parameters. The tradeoffs will differ depending upon the system, with factors such as external disturbances and whether or not the parameters are time-varying playing a major part. Thus pipelining is a tool that is useful for some, but not all problems.

Besides the improvements possible through increased control bandwidth, tracking accuracy may also be improved through more conventional methods such as improving the feedforward control algorithm itself and analysing the dynamics to improve the choice of estimator parameters. The model of the robot itself can be improved through the use of non-linear models and corresponding controllers, and improved choice of estimation parameters can result in faster and more accurate parameter convergence. Furthermore, a more detailed investigation into the 'bumpless transfer' problem could also result in significant tracking improvements.

The above changes would certainly require a revisit of the parallel design procedure illustrated in Chapter 4. This procedure itself may be improvable through the use of non-deterministic models of the software reflecting the few conditional code sections and other

non-deterministic aspects of the system, such as local bus activity. Furthermore, the results of Section 5.1 illustrate the need to improve the method of determining contention for system resources.

Figure 5.11 - Link 1 Parameter a_1 Convergence (Slow Trajectory)

Figure 5.12 - Link 1 Parameter a_2 Convergence (Slow Trajectory)

Figure 5.13 - Link 1 Parameter b_1 Convergence (Slow Trajectory)

Figure 5.14 - Link 1 Parameter b_2 Convergence (Slow Trajectory)

Figure 5.15 - Link 2 Parameter a_1 Convergence (Slow Trajectory)

Figure 5.16 - Link 2 Parameter a_2 Convergence (Slow Trajectory)

Figure 5.17 - Link 2 Parameter b_1 Convergence (Slow Trajectory)

Figure 5.18 - Link 2 Parameter b_2 Convergence (Slow Trajectory)

Figure 5.19 - Link 1 Parameter a_1 Convergence (Fast Trajectory)

Figure 5.20 - Link 1 Parameter a_2 Convergence (Fast Trajectory)

Figure 5.21 - Link 1 Parameter b_1 Convergence (Fast Trajectory)

Figure 5.22 - Link 1 Parameter b_2 Convergence (Fast Trajectory)

Figure 5.23 - Link 2 Parameter a_1 Convergence (Fast Trajectory)

Figure 5.24 - Link 2 Parameter a_2 Convergence (Fast Trajectory)

Figure 5.25 - Link 2 Parameter b_1 Convergence (Fast Trajectory)

Figure 5.26 - Link 2 Parameter b_2 Convergence (Fast Trajectory)

Figure 5.27 - Link 1 Slow Trajectory and Single Processor Tracking

Figure 5.28 - Link 2 Slow Trajectory and Single Processor Tracking

Figure 5.29 - Link 1 Positional Tracking Error (Slow Trajectory)

Figure 5.30 - Link 2 Positional Tracking Error (Slow Trajectory)

Figure 5.31 - Link 1 Fast Trajectory and Single Processor Tracking

Figure 5.32 - Link 2 Fast Trajectory and Single Processor Tracking

Figure 5.33 - Link 1 Positional Tracking Error (Fast Trajectory)

Figure 5.34 - Link 2 Positional Tracking Error (Fast Trajectory)

Figure 5.35 - Link 1 Covariance Matrix Trace (Fast Trajectory)

Figure 5.36 - Link 2 Covariance Matrix Trace (Fast Trajectory)

Figure 5.37 - Link 1 Detailed Covariance Matrix Trace (Fast Trajectory)

Figure 5.38 - Link 2 Detailed Covariance Matrix Trace (Fast Trajectory)

Figure 5.39 - Link 1 Covariance Matrix Trace (Slow Trajectory)

Figure 5.40 - Link 2 Covariance Matrix Trace (Slow Trajectory)

Figure 5.41 - Link 1 Detailed Covariance Matrix Trace (Slow Trajectory)

6.0 Conclusions

This research has investigated the application of parallel processing to manipulator joint control algorithms with the goal of attaining higher sampling rates. Unlike the majority of research on this subject in the literature, the thesis addresses the subject of adaptive control algorithms for manipulator control. The parallelism inherent in adaptive control algorithms for robotic manipulator control was utilized to increase the system sampling rate while maintaining identical functionality. A set of parallel design tools was used to show the method of design by performance analysis.

The design and implementation of the parallelized control algorithm attained speedups of 1.58 using three processors and 2.19 with six processors, with efficiencies of 52.7% and 36.6%, respectively. This was relatively close to the performance predicted during the design phase although the differences indicate the requirement for improved modelling of system resource contention. The low speedup bounds indicate that this problem is not particularly amenable to parallel processing, but the cycle rate increase achievable is significant. These performance improvements begin to show the potential of applying parallelism to adaptive controllers in order to increase the available processing power for increasing the sampling rate or adding additional functionality such as signal filtering.

The computational performance results obtained in this research are typical of those mentioned in the literature for shared bus architectures as seen in Table 3.2. These architectures involve an inherent limitation since problems with bus loading and the

resulting contention are inevitable as parallelism increases. This limitation causes designers to attempt to minimize interprocessor communications by using coarser task granularity. The problem can be ameliorated by the ongoing development of higher bandwidth buses and by the use of multiple buses. However, alternate interconnection strategies such as the serially interconnected Transputer hypercube will always have the advantage in communications throughput.

Alternate architectures have their own limitations, however. The disadvantage appears in the increased design effort required to deal with the interconnection network structure. Performance advantages are gained solely by exploiting the topology and distributed memory of such architectures, not necessarily an easy task with adaptive manipulator control algorithms since they do not naturally conform to a symmetric network. Thus the shared memory multiprocessor forms a good generalized structure to obtain modest speed improvements with relatively little effort. For exceptionally high throughput, special purpose VLSI systolic arrays and pipelined processors are likely the ideal choice, although at a much higher design effort cost.

6.1 Summary of Contributions

The research involved with this thesis has shown the usefulness of using the Activity Sequence Network notation in parallel software design. The ASN notation provides a clear and concise tool for modelling parallel software. Coupling the ASN notation with the automated GreatSPN timed Petri Net tool provided a design environment ideally suited to

the design of real-time software to meet a hard deadline, in this case the sampling period of an adaptive manipulator controller. The Petri Net tool with the extension to generate Gantt charts of the activity timing allowed quick identification of critical paths in the predicted software cycle, promoting quick turnaround during the design phase. These tools also provide enough information in the design phase to estimate achievable speedup at an early stage, thus avoiding possibly unpleasant surprises much later.

The performance analysis prediction of the achievable sampling rate proved to provide estimations within 30% of the actual, given sequential implementation timings. These results highlighted the necessity of accurately modelling the contention for shared resources, in this case the VMEBus and the shared memory.

This research has resulted in the development of a testbed ideally suited to rapid prototyping of parallel real-time adaptive control algorithms for the Carleton SCARA-type robotic manipulator. This testbed provides a high performance base for the simulated testing and evaluation of fast adaptive manipulator control algorithms, and will require only a modest effort to modify for testing with the actual manipulator. A design methodology has been provided to simplify further experimentation with high performance testing of alternative adaptive algorithms. A trajectory generation routine has also been provided, allowing automated generation of smooth trajectories.

Finally, the design and implementation have indicated that the self-tuning regulator algorithm for manipulator control that was utilized in this thesis is not highly suited to parallelism. However, the sampling rate improvement illustrated by this implementation

3 of/de 3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

has shown that significant improvement is achievable and rates of close to four times the sequential rate should be attainable with only modest effort.

6.2 Areas for Further Research

There are many areas of research identified by this thesis that can be pursued. As discussed in Section 5.3, several design opportunities exist for improving the computational performance. One of the most obvious would be to design a multi-rate software architecture where a small and fast feedforward/feedback control loop would provide a very high sampling rate, with parameter estimation updating on a much slower cycle, possibly pipelined. Many variations on that theme are possible.

Furthermore, the discussion in Section 3.3.3 could be pursued to determine if an optimal or heuristic allocation tool would provide both better allocation and faster determination of the allocation. Integrated optimal and rule-based allocation tools would form ideal expansions to the Activity Sequence Editor described in [Woo90], allowing for faster design cycles and thus more experimentation. Other improvements to the system that would aid in the design and experimentation are the addition of a graphical tool for visualization and visual analysis, and a tool based on the event monitor uploader for uploading the data captured during an experiment.

An area that has been left untouched is that of the non-deterministic effects of the hardware and operating system. Perhaps improved software modelling techniques including

stochastic effects could lead to further performance enhancement. Also relevant to this area is the modelling of the memory and bus contention under this type of control problem. The experimentation here has shown that simple estimation techniques do not provide adequate estimations of resource contention.

The question of hardware architecture must also be addressed. As the survey in Section 3.1 showed, there has been much research into alternative hardware architectures for the Newton-Euler inverse dynamics problem, but very little applied to the direct dynamic models or to robotic adaptive control problems. An investigation is warranted into the advantages and disadvantages of the different parallel machine classes for this type of manipulator control problem.

The example feedforward control algorithm used in this research is relatively simple, and must be replaced before the simulation can be removed so the testbed can be connected to the Carleton manipulator. Replacement is a requirement since the link lengths do not meet the link decoupling conditions. The nonlinear feedback control law using the recursive least squares estimator discussed by Schwartz, Warshaw, and Janabi [Sch90] is an ideal candidate, but would require some redesign of the existing software. It has increased computational complexity over the linear system used here, and the feedback nature will change the system data flow.

Besides the actual control algorithm, other additions will be required when operating with the physical manipulator, such as a filtering capability to deal with noise coming from the robot transducers, motors, and wiring. For example, a Kalman filter could be implemented

in a separate processor to smooth incoming sampled data. Experiments could also be carried out to determine the quality of control under conditions of time-varying parameters. Finally, if the controller is to be used over long periods of time, numerical instability will likely become an issue. Research into parallelization of the square root formulation of the recursive least squares estimator for small-scale problems should be investigated.

Glossary

AC	adaptive control
A/D	analog to digital conversion
ALU	arithmetic logic unit
APU	arithmetic processing unit
ASN	Activity Sequence Network
Back	Feedback Controller activity
Chk Delta	Parameter Delta Checking activity
Chk White	Whiteness Checking activity for prediction error
Cov	Covariance Update activity
CPU	central processing unit, or microprocessor
direct drive robot	manipulator whose motor shafts are directly coupled to the link joints
DM	distributed memory
DOF	degrees of freedom
DSP	digital signal processing
Forward (Fwd)	Feedforward Controller activity
G	gravities
Hz	hertz (cycles per second)
indirect drive robot	manipulator whose motor torques are transmitted to the link joints via gear trains
JIFFE	a very long instruction word processor designed for robot control (see [And89])
KHz	kilohertz
LSR	least squares recursion
Mb	megabit
MHz	megahertz (1000 cycles per second)
MIMD	multiple instruction (stream) multiple data (stream)

MRAC	model referenced adaptive controller
ms	milliseconds
m/sec	meters per second
MVP	matrix-vector product activity - solves for K and d
NE	Newton-Euler inverse dynamics method
P3	three processor restricted parallelism implementation
P6	six processor restricted parallelism implementation
PD	proportional-derivative controller
Phi	Phi Update activity
PID	proportional-integral-derivative controller
Pred	Prediction Error activity
RCCL	Robot Control C Library
RTC	Real-Time Control system
Sample	Sampling activity
SCARA kinematics	refers to the Selective Compliance Assembly Robot Arm configuration employing a closed loop five bar mechanism
SIMD	single instruction (stream) multiple data (stream)
SISD	single instruction (stream) single data (stream)
SM	shared memory
STR	self-tuning regulator
Sum	Summing (or applying) activity
Theta	Parameter Update activity
UDU	upper triangular and diagonal factorization method for recursive least squares estimation method
μs	microseconds

Bibliography

- [Ahm89] Ahmed, S. and Li, B. Robot Control Computation in Microprocessor Systems with Multiple Arithmetic Processors Using a Modified DF/IHS Scheduling Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics* 19, 5 (Sept/Oct 1989), 1167-1178.
- [And83] Andrews, G.R. and Schneider, F.B. Concepts and Notations for Concurrent Programming. *Computing Surveys* 15, 1 (March 1983), 3-43.
- [And89] Andersson, R.L. Computer Architectures for Robot Control: A Comparison and a New Processor Delivering 20 Real MFLOPS. In *Proceedings of the IEEE Conference on Robotics and Automation*, 1989, pp. 1162-1167.
- [Arm89] Armstrong, B. On Finding Exciting Trajectories for Identification Experiments Involving Systems with Nonlinear Dynamics. *The International Journal of Robotics Research* 8, 6 (December 1989), 28-48.
- [Asa86] Asada, H. and Slotine, J.J.E. *Robot Analysis and Control*, John Wiley and Sons (1986).
- [Asa87] Asada, H. and Youcef-Toumi, K. *Direct-Drive Robots, Theory and Practice*, The MIT Press, Cambridge, Massachusetts (1987).
- [Ast71] Åström, K.J. and Eykhoff, P. System Identification - A Survey. *Automatica* 7(1971), 123-162.
- [Ast73] Åström, K.J. and Wittenmark, B. On Self Tuning Regulators. *Automatica* 9(1973), 185-199.
- [Ast89] Åström, K.J. and Wittenmark, B. *Adaptive Control*, Addison Wesley (1989).
- [Bai89] Bailey, D.L. Configuration Guide for Unison/DY-4 Multiprocessor Testbed. Tech. Rept. Technical Report, Carleton University, April, 1989.
- [Bai90] Bailey, D.L. The Unison Kernel Event Monitor: User's Manual and Implementation Guide. Tech. Rept. Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, September, 1990.
- [Bie76] Bierman, G.J. Measurement Updating Using the U-D Factorization. *Automatica* 12(1976), 375-382.

- [Bie77] Bierman, G.J. *Factorization Methods for Discrete Sequential Estimation*. Academic Press (1977).
- [Bor89] Borenstein, J. and Koren, Y. Real-Time Obstacle Avoidance for Fast Mobile Robots. *IEEE Transactions on Systems, Man, and Cybernetics* 19, 5 (Sept./Oct. 1989), 1179-1187.
- [Bra83] Brand, D. and Zafiropolo, P. On Communicating Finite-State Machines. *Journal of the Association for Computing Machinery* 30, 2 (April 1983), 323-342.
- [Bro84] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery* 31, 3 (July 1984), 560-599.
- [Bro90] Brodtkin, J. and Patel, R.V. An Architecture For An Advanced Robot Controller. In *Canadian Conference on Electrical and Computer Engineering*, IEEE, Ottawa, Ontario, Canada, September 1990, pp. 26.4.1-4.
- [Bru89] Bruno, G. and Marchetto, G. Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems. *IEEE Transactions on Software Engineering* (February 1989), 146.
- [Buh89a] Buhler, M., Whitcomb, L.L., Levin, F., and Koditschek, D.E. A Distributed Message Passing Computational and I/O Engine for Real-Time Motion Control. In *Proceedings of the American Control Conference*, 1989, pp. 478-483.
- [Buh89b] Buhr, R.J.A. A Short Description of Machine Charts. Tech. Rept. SCE Report 89-4, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, March, 1989.
- [Chi87] Chiola, G., *GreatSPN Users' Manual*, Universita degli Studi di Torino, Torino, Italy, Version 1.3, September, 1987.
- [Chi88] Chisci, L. and Zappa, G. High Throughput Parallel Implementation of RLS. In *Proceedings of the IFAC Identification and System Parameter Estimation Symposium*, 1988, pp. 561-566.
- [Chi90] Chinneck, J.W., Goubran, R.A., Karam, G.M., and Lavoie, M. A Design Approach for Real-Time Multiprocessor DSP Applications. Tech. Rept. SCE-90-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, February, 1990.
- [Cra86] Craig, J.J. *Introduction to Robotics, Mechanics and Control*, Addison-Wesley (1986).

- [Cra87]** Craig, J.J., Hsu, P., and Sastry, S.S. Adaptive Control of Mechanical Manipulators. *The International Journal of Robotics Research* 6, 2 (1987), 16-28.
- [Cra88]** Craig, J.J. *Adaptive Control of Mechanical Manipulators*, Addison-Wesley (1988).
- [Dom87]** Domey, J. and Burtnyk, N. 3D Vision for Robotics and Automation. In *Proceedings of the Canadian Engineering Centennial Convention*, Montreal, Quebec, May 1987, pp. 8.3.1-8.3.13.
- [DY-88]** *DVME-134 Single Board Computer (SBC) Operations Manual*, DY-4 Systems Inc., 1.b, March, 1988.
- [Eag89]** Eager, D.L., Zahorjan, J., and Lazowska, E.D. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers* 38, No. 3 (March 1989).
- [Elg88]** Elgazzar, S., Green, D., and Gentleman, W.M. Open System Architecture of a Multiprocessor Robot Controller. In *Proceedings of the 7th Canadian CAD/CAM & Robotics Conference*, Toronto, Ontario, June 1988, pp. 63-69.
- [Fau88]** Faulk, S.R. and Parnas, D.L. On Synchronization in Hard Real-Time Systems. *Communications of the ACM* 31, 3 (March 1988), 274-287.
- [Fle89]** Fleming, P., Jones, D., and Jones, S. Parallel Processing Architectures And Applications For Real-Time Control. In *Proceedings of the American Control Conference*, 1989, pp. 2708-2714.
- [For81]** Fortescue, T.R., Kershenbaum, L.S., and Ydstie, B.E. Implementation of Self-tuning Regulators with Variable Forgetting Factors. *Automatica* 17, 6 (1981), 831-835.
- [Fra89]** Franks, R.G. *Experience in Concurrency in OSI Communications Software: A Comparative Performance Study*, Master's thesis, Sys.Comp.Eng., Carleton University, July 1989.
- [Fre71]** Frederick, D.K. and Carlson, A.B. *Linear Systems in Communication and Control*, John Wiley and Sons, Inc. (1971).
- [Gar90]** Garcia-Nocetti, F., Thompson, H.A., DeOliveira, M.C.F., Jones, C.M., and Fleming, P.J. Implementation of a Transputer-based Flight Controller. In *IEE Proceedings*, May 1990, pp. 130-136.
- [Gen73]** Gentleman, W.M. Least Squares Computations by Givens Transformations Without Square Roots. *Journal of the Institute of Mathematics Applications* 12(1973), 329-336.

- [Gen89] Gentleman, V.M., MacKay, S.A., Stewart, D.A., and Wein, M. *Using the Harmony Operating System: Release 3.0*, NRCC, NRC, Ottawa, Ontario, ERA-377 (February 1989).
- [Gre89] Green, D., Liscano, R., and Wein, M. Real-Time Control of an Autonomous Mobile Robot using the Harmony Operating System. In *Proceedings of the Fourth IEEE International Symposium on Intelligent Control*, Albany, NY, September 1989.
- [Gri89a] Grigg, K.E. *Use of the Intel iPSC Hypercube for Robotic Inverse Dynamics Computations*, Term Proj., Parallel Algorithms 95.574, Carleton University, April 1989.
- [Gri89b] Grigg, K.E. *Survey of Parallel Methods in Robotic Inverse Dynamics Computations*, Term Proj., Parallel Algorithms 95.574, Carleton University, April 1989.
- [Gri90] Grigg, K.E., Miguët, S., and Robert, Y. Symmetric Matrix-Vector Product on a Ring of Processors. *Information Processing Letters* 35, 5 (August 1990), 239-248. Also appears in *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, North Carolina, April 1990..
- [Has88] Hashemipour, H.R., Roy, S., and Laub, A.J. Decentralized Structures for Parallel Kalman Filtering. *IEEE Transactions on Automatic Control* 33, 1 (January 1988), 88-94.
- [Has89] Hashimoto, K. and Kimura, H. A New Parallel Algorithm for Inverse Dynamics. *The International Journal of Robotics Research* 8, 1 (February 1989), 63-76.
- [Hay82] Haynes, L.S., Lau, R.L., Siewiorek, D.P., and Mizell, D.W. A Survey of Highly Parallel Computing. *IEEE Computer* 15, 1 (January 1982), 9-24.
- [Hou87] Houstis, C.E. and Aboelaze, M. The Mapping of Applications to Multiple Bus and Banyan Interconnected Multiprocessor Systems: A Case Study. *Information Processing Letters* (1987).
- [Hsi86] Hsia, T.C. Adaptive Control of Robot Manipulators - A Review. In *Proceedings of the IEEE Conference on Robotics and Automation*, April 1986, pp. 183-189.
- [Ise82] Isermann, R. Parameter Adaptive Control Algorithms - A Tutorial. In *Proceedings of the IFAC Symposium on Automatic Control*, 1982, pp. 513-528.

- [Ish89] Ish-Shalom, J. and Kazanzides, P. Signal Processor Architecture for High-Performance Real-Time Applications. In *IEEE CH2803-5/89*, 1989, pp. 184-193.
- [Jon89] Jones, D.I. and Entwistle, P.M. Parallel Computation of an Algorithm in Robotic Control. In *IFAC Symposium*, 1989, pp. 438-443.
- [Kab89] Kabuka, M. and Escoto, R. Real-Time Implementation of the Newton-Euler Equations of Motion on the NEC uPD77230 DSP. *IEEE Micro* (February 1989), 65-76.
- [Kai80] Kailath, T. *Linear Systems*, Prentice-Hall, Inc. (1980).
- [Kim88] Kim, S.J. and Browne, J.C. A General Approach to Mapping of Parallel Computations Upon Multiprocessor Architectures. In *Proceedings of the International Conference on Parallel Processing*, IEEE, 1988, pp. 1-8.
- [Kim90] Kim, S. and Agrawal, D.P. Least-Squares Multiple Updating Algorithms on a Hypercube. *Journal of Parallel and Distributed Computing* 8(1990), 80-88.
- [Koi83] Koivo, A.J. and Guo, T.H. Adaptive Linear Controller for Robotic Manipulators. *IEEE Transactions on Automatic Control* AC-28, 2 (February 1983), 288-297.
- [Kos87] Kossman, D. and Malowany, A. A Multi-processor Robot Control System for RCCL under iRMX. In *IEEE International Conference on Robotics and Automation*, 1987, pp. 1298-1306.
- [Kri88] Krieger, M., Junaid, A., and Harvey, R.A. Centrally Coordinated Embedded Systems and Their Design Using Process Activity Language. Tech. Rept. Dept. of Electrical Engineering, University of Ottawa, Ottawa, Ontario, 1988.
- [Kun78] Kung, H.T. and Leiserson, C.E. Systolic Arrays (for VLSI). In *Proceedings of the Symposium on Sparse Matrix Computations and their Applications*, 1978, pp. 256-282.
- [Lai88] Lainiotis, D.G., Likothanasis, S.D., and Katsikas, S.K. Adaptive Control Algorithms: A Comparative Computational Analysis-Parallelism. *Control and Computers* 16, 1 (1988), 22-27.
- [Lat85] Lathrop, R.H. Parallelism in Manipulator Dynamics. *The International Journal of Robotics Research* 4, 2 (1985), 80-102.
- [Lee86a] *Tutorial on Robotics*, IEEE Press (1986).

- [Lee86b]** Lee, C.S.G. and Chang, P.R. An Efficient Parallel Algorithm for Robot Inverse Dynamics Computation. *IEEE Transactions on Systems, Man, and Cybernetics* 16, 4 (July 1986).
- [Leu88]** Leung, S.S. and Shanblatt, M.A. Computer Architecture Design for Robotics. In *IEEE Conference on Robotics and Automation*, 1988, pp. 453-456.
- [Leu90]** Leung, Y.T. and Suri, R. Performance Evaluation of Discrete Manufacturing Systems. *IEEE Control Systems Magazine* 10, 4 (June 1990).
- [Lis89]** Liscano, R. and Green, D. Design and Implementation of a Trajectory Generator for an Indoor Mobile Robot. In *Proceedings of the IEEE/RSJ International Workshop on Intelligent Robots and Systems*, IEEE/RSJ, Tsukuba, Japan, September 1989.
- [Liu73]** Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery* 20, 1 (1973), 46-61.
- [Mar83]** Marsan, M.A., Balbo, G., Conre, G., and Gregoretti, F. Modeling Bus Contention and Memory Interference in a Multiprocessor System. *IEEE Transactions on Computers* C-32, 1 (January 1983), 60-72.
- [Mey87]** Meystel, A. Intelligent Control: Highlights and Shadows. In *Proceedings of the IEEE International Symposium on Intelligent Control*, IEEE, 1987, pp. 2-8.
- [Mok84]** Mok, A. The Decomposition of Real-Time System Requirements into Process Models. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1984, pp. 125-134.
- [Mon85]** Money, S.A. *Microprocessors in Instrumentation and Control*, William Collins Sons & Co. Ltd. (1985).
- [Mon89]** Monforton, G.G. *A Fast Task Allocator for Multiprocessing Systems*, Master's thesis, Carleton University, Ottawa, Ontario, Canada, July 1989.
- [Mot87]** *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, Motorola Inc., First Edition, 1987.
- [Mot88]** *Motorola MC68020 Programmer's Reference Manual*, Motorola Inc., 1.b, 1988.
- [Mul87]** *Unison Integration Toolkit*, Multiprocessor Toolsmiths, Inc., Kanata, Ontario, Software Release 1.1, November, 1987.

- [Nad79] Nader, A. Petri Nets for Real Time Control Algorithms Decomposition. In *Distributed Computer Control Systems, Proceedings of the IFAC Workshop*, Harrison, T.J., International Federation of Automatic Control, Pergamon Press, Tampa, Florida, October 1979, pp. 197-210.
- [Nar89a] Narendra, K.S. and Annaswamy, A.M. *Stable Adaptive Systems*, Prentice-Hall (1989).
- [Nar89b] Narasimhan, S., Siegel, D.M., and Hollerbach, J.M. CONDOR: A Coarse-grained Parallel Architecture for Robot Control. In *Proceedings of the American Control Conference*, 1989, pp. 484-488.
- [Nec89] Neculescu, D.S., Jassemi-Zargani, R., and Graham, W.B. The Methods of Artificial Potential Fields and Impedance Control in Robotics. In *Proceedings of the CASI Symposium on Space Station*, Canadian Aeronautics and Space Institute, Radisson Hotel, Ottawa, Canada, November 1989.
- [Nei88] Neilson, J.E., Woodside, C.M., Petriu, D.C., and Miernik, J.W. A Short Tutorial on Stochastic Rendezvous Network Models. Tech. Rept. CSM-7.1, Carleton University, Ottawa, Ontario, May, 1988.
- [Nig85] Nigam, R. and Lee, C.S.G. A Multiprocessor-Based Controller for the Control of Mechanical Manipulators. *Journal of Robotics and Automation RA-1*, 4 (December 1985), 173-182.
- [Ost87] Ostroff, J.S. and Wonham, W.M. Modelling, Specifying and Verifying Real-time Embedded Computer Systems. In *Proceedings of the 1987 Real-Time Systems Symposium*, IEEE, 1987, pp. 124-132.
- [Par90] Parameswar, A. and Sinha, N.K. A Transputer Based Parallel Processing Architecture For Robotics Applications. In *Canadian Conference on Electrical and Computer Engineering*, IEEE, Ottawa, Ontario, Canada, September 1990, pp. 27.5.1-4.
- [Pet77] Peterson, J.L. Petri Nets. *Computing Surveys* 9, 3 (September 1977), 223-252.
- [Phi90] *Design Principles for Shared Memory Multiprocessors - A collection of papers from the MUMS project*, Lund University, Sweden (May 1990).
- [Raj89] Rajguru, S.S. *Analysis and Design of a Direct Drive Robot*, Master's thesis, Carleton University, Ottawa, Ontario, September 1989.
- [Riv88] Rivin, E.I. *Mechanical Design of Robots*, McGraw-Hill (1988).
- [Sod89] Söderström, T. and Stoica, P. *System Identification*, Prentice Hall, Systems and Control Engineering (1989).

- [Sch89]** Schwartz, H.M. and Warshaw, G. On the Richness Condition for Robot Adaptive Control. In *Proceedings of the W.A.M.*, 1989.
- [Sch90]** Schwartz, H.M., Warshaw, G., and Janabi, T. Issues in Robot Adaptive Control. In *Proceedings of the 1990 American Control Conference*, 1990.
- [Sha88]** Shah, S.L. and Cluett, W.R. *Implementation of Self-Tuning Controllers*, Peter Peregrinus Ltd. (1988), pp. 23-40, Chapter 2.
- [Tak89]** Takanashi, N., Ikeda, T., and Tagawa, N. A High-Sample-Rate Robot Control System Using A DSP Based Numerical Calculation Engine. In *Conference on Robotics and Automation*, IEEE, 1989, pp. 1168-1173.
- [Uni]** *PUMA Series Manipulator User Manual*, Unimation, Inc..
- [Veg89]** Vega, P., Serrano, J., and Prada, C. A Comparative Study of Automatic Tuning Methods for Industrial PID Controllers. *Control and Computers* 17, 3 (1989), 67-70.
- [War89]** Warshaw, G. *Investigation of Adaptive Control for a Direct Drive Manipulator*, Master's thesis, Sys.& Comp. Eng., Carleton University, April 1989.
- [Woo85]** Woodside, C.M., Neron, E., Ho, E.D.S., and Mondoux, B. An 'Active Server' Model for the Performance of Parallel Programs Written Using Rendezvous. Tech. Rept. ARTT 85-1, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, 1985.
- [Woo88a]** Woodside, C.M. Activity Sequence Network Models for Software Performance Engineering. Tech. Rept. Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, October, 1988.
- [Woo88b]** Woodside, C.M. An Introductory Description of the Activity Sequence Net Notation. Tech. Rept. Draft, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, September, 1988.
- [Woo90]** Woodside, C.M. and Romaniuk, R. An Activity Sequence Network Specification Tool for Software Engineering. In *Canadian Conference on Electrical and Computer Engineering*, IEEE, Ottawa, Ontario, Canada, September 1990, pp. 11.1.1-4.
- [Zub87]** Zubair, M. and Madan, B.B. Time Efficient Systolic Architecture for Matrix-Vector Multiplication. *Information Processing Letters* 24(1987), 225-231.

END

03/03/92

FIN