INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Performance and reliability of a parallel robot controller

Hamilton, Deirdre Lynne, M.S.
Rice University, 1992
RICE UNIVERSITY

Performance and Reliability of a Parallel Robot Controller

by

Deirdre L. Hamilton

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. John K. Bennett, Co-Chairman
Assistant Professor
Electrical and Computer Engineering

Dr. Ian D. Walker, Co-Chairman
Assistant Professor
Electrical and Computer Engineering

Dr. Joseph R. Cavallaro
Assistant Professor
Electrical and Computer Engineering

Houston, Texas
April, 1992
Abstract

Performance and Reliability of a Parallel Robot Controller

Deirdre L. Hamilton

Most robot controllers today are uniprocessor architectures. As robot control algorithms become more complex, these serial controllers have difficulty providing the desired response times. Additionally, with robots being used in environments that are hazardous or inaccessible to humans, fault-tolerant robotic systems are particularly desirable. A uniprocessor control architecture cannot offer tolerance of processor faults. Use of multiple processors for robot control offers two advantages over single processor systems. Parallel control provides a faster response, which in turn allows a finer granularity of control. Processor fault tolerance is also made possible by the existence of multiple processors. There is a trade-off between performance and level of fault tolerance provided. The work of this thesis shows that a shared memory multiprocessor robot controller can provide higher performance than a uniprocessor controller, as well as processor fault tolerance. The trade-off between these two attributes is also demonstrated.
Acknowledgments

I would like to express my appreciation to my advisors, Dr. Ian Walker and Dr. John Bennett, for their support and encouragement. I would also like to thank Dr. Joseph Cavallaro for serving on my thesis committee, and for his help with this work. I would especially like to thank Dr. Walker for his calming demeanor when I really needed it.

I thank my parents for the love and support that has brought me this far, and my brother who always has the greatest faith in me. I am grateful for friends who have helped me to continue to move forward with their words of encouragement (electronic and otherwise). I owe special thanks to Russell and Valerie for enduring practice presentations, and to Stanley for calling me at five o’clock every morning. Thanks also go to Monica for helping to make various code changes painless, and for her tips on latex and printing “screendumps”.

Finally, I must thank those who endured great pain and humiliation so that I could do more than only dream.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables ix

1 Introduction 1

2 Background 9

2.1 Robot Control ........................................ 9
2.1.1 Dynamic Model ................................... 10
2.1.2 Multiprocessor Robot Controllers .............. 12
2.2 Shared Memory ....................................... 13
2.3 Processor Fault Tolerance .......................... 15
2.3.1 Hardware Fault Tolerance ......................... 16
2.3.2 Software Fault Tolerance ......................... 19

3 Related Work 21

3.1 Parallel Computing in Robotics .................... 21
3.2 Multiprocessor Robot Control Architectures .......... 21
  3.2.1 Message-passing .................................... 21
  3.2.2 Shared Memory ....................................... 22
  3.3 Fault Tolerant Multiprocessor Robot Controllers ..... 24

4 Robot Controller Implementation ......................... 29
  4.1 Architecture ............................................ 29
  4.2 Robot Control without Processor Fault Tolerance ...... 31

5 Implementing Processor Fault Tolerance ...................... 43
  5.1 Controller with High Fault Tolerance .................... 47
  5.2 Controller with Lower Fault Tolerance .................. 57

6 Handling Joint Limits Dynamically .......................... 60

7 Evaluation .................................................. 72
  7.1 Performance Results ..................................... 72
  7.2 Analysis of Architecture .................................. 76
    7.2.1 Inefficiencies of Current Architecture .............. 76
    7.2.2 Architectural Alternatives ........................... 77

8 Future Work ................................................ 81
Illustrations

1.1 Design Trade-offs ............................................. 5

2.1 A Robot Model ................................................. 10
2.2 Multiprocessor Memory Models .......................... 14

4.1 System Configuration ......................................... 30
4.2 Controller and Robot .......................................... 36
4.3 Model Display: Beginning of Demonstration with No Failures .... 37
4.4 Middle of Demonstration with No Failures .................... 38
4.5 End of Demonstration with No Failures ...................... 39
4.6 Serial task decomposition ..................................... 40
4.7 Parallel task decomposition (without fault tolerance) .......... 42

5.1 Processor Reconfiguration Scheme .......................... 45
5.2 Processor Reconfiguration Example .......................... 46
5.3 Comparison Vector and Working Set Suggestion for Processor 0 . . . 49
5.4 Compare and Vote Example with One Failure .......................... 49
5.5 Compare and Vote Example with Two Failures ....................... 50
5.6 Parallel task decomposition (for HFT) ................................. 52
5.7 Model Display: Beginning of Processor Failure Demonstration ..... 54
5.8 Failure of Processor 2 ................................................. 55
5.9 Recovery of Processor 2 ............................................... 56
5.10 Parallel task decomposition (for LFT) ................................ 58

6.1 Locked Joint Example ................................................ 61
6.2 Workspace After Joint Failure ........................................ 61
6.3 $\theta_2$ Limit Change for Locked $\theta_1$ ................................ 64
6.4 $\theta_1$ Limit Change for Locked $\theta_2$ ................................ 65
6.5 $\theta_3$ Limit Change for Locked $\theta_2$ ................................ 66
6.6 $\theta_2$ Limit Change for Locked $\theta_3$ ................................ 67
6.7 Model Display: Beginning of Locked Joint Demonstration ........ 68
6.8 Joint 1 Reaches Limit .................................................. 69
6.9 Joint 0 Compensates .................................................... 70
6.10 End of Locked Joint Demonstration ................................... 71
Tables

4.1 Inertia Matrix Expressions ......................... 33
4.2 Centrifugal and Coriolis Torque Vector Expressions ................. 34
6.1 Limit Constraints for Locked $\theta_0$ ..................... 63
6.2 Limit Constraints for Locked $\theta_1$ ..................... 64
6.3 Limit Constraints for Locked $\theta_2$ ..................... 65
6.4 Limit Constraints for Locked $\theta_3$ ..................... 67
7.1 Relative Performance of Controllers ..................... 74
Chapter 1

Introduction

Robots are used to perform tasks in the place of humans for several reasons: the task is tedious and could be performed by a robot more expeditiously than a human and at a lower cost; or the task may require greater precision than most humans can provide; or the environment of the task may be hazardous or inaccessible to humans. The tasks that robots perform have become more complex, requiring increasingly finer motion control and faster controller response time. In addition to the need for improved performance, robot reliability has assumed increasing importance, especially in critical tasks such as space exploration and operation in environments contaminated with hazardous materials.

Efficient and fast calculation of robot dynamics is a key issue in robot manipulator control. Advances in robot control theory have produced algorithms that are difficult to implement on a uniprocessor system in real time. Parallel architectures have therefore been proposed to fulfill this need [20, 12]. Robotic kinematics and dynamics calculations are well suited for parallelism due to the numerous matrix and vector manipulations involved. Multiple processors are then used to speed up calculations by having each processor work on a different row or column of a matrix. Executing such
operations in parallel allows faster manipulation of the robot than a serial controller, permitting finer control of motion. Since different joints of a robot are typically controlled the same way, some areas where parallelism may be exploited are usually explicit.

Reliable robotic systems are most desirable for use in environments where humans cannot go for safety or health reasons. Modular redundancy can be used to improve the reliability of robotic systems. Possible points of failure are duplicated so that a failed module may be replaced, and work is not halted. Mechanical redundancy may exist in any of the following forms:

- multiple motors per degree of freedom
- multiple sensors of the same or different types, but feeding back the same information
- extra degrees of freedom (more than the minimum required for the workspace)

A robot's workspace is the union of all points that it can reach. This area is affected by the number of degrees of freedom of the robot. Loss of the only motor providing movement in a direction results in loss of access to some of the workspace. This is also true for loss of a degree of freedom due to a locked joint. The sensors provide knowledge of the actual position of the robot. If the only sensor attached to a joint fails, then detection of that joint's activity and position are lost. While the joint can
still be moved, there is no way of detecting whether it has reached its desired position. Thus, control of the joint is lost.

Besides mechanical failures, the robot may suffer from corruption of the information used to plan and control its movement due to failure in the controlling software or processor hardware. The use of multiple processors provides fault tolerance via processor redundancy. If multiple processors are available, the failure of one processor, or a few processors, does not result in system failure. However, incorporating this fault tolerance, which requires extra computational overhead, may degrade controller performance. Depending on the level of fault tolerance (i.e. the number of redundant processors) incorporated into the system, this degradation in performance may be significant. In environments where "down time" is not critical and where system components may be repaired or replaced, the importance of fault tolerance is diminished. Here maximum speedup can be achieved through efficient task decomposition and the use of a number of processors equal to the maximum number of tasks executed in parallel [17]. However, in some environments, fault tolerance may be the most important factor in system design. A robot used in space cannot be easily repaired or replaced. A robot used in an environment that is hazardous to human health, for example a radioactive environment, cannot easily be serviced once it is exposed to the hazard. In these instances, robot reliability is paramount. Rather than the processors performing different tasks in parallel for better performance, some tasks are executed on every processor in parallel to produce redundant data for com-
parison and detection of processor failure. To ensure that the correct data is being compared, all processors must synchronize just before the data comparisons between processors. For these two reasons — the addition of comparison code and additional synchronization — performance is diminished by the incorporation of fault tolerance.

The attainable speedup and the level of fault tolerance provided in a parallel architecture are both limited by the number of processors available. At the cost of more processors, it is possible to have a fault-tolerant architecture that can also provide maximum speedup. The number of processors required would be \( m \times n \), where \( m \) is the maximum number of different tasks performed in parallel for maximum speedup, and \( n \) is the number of processors that must perform the same task in parallel for fault tolerance. Thus, there is a clear trade-off between performance, cost, and the level of fault tolerance desired, as shown in Figure 1.1. The most cost-effective performance improvement would be such that when fault tolerance is incorporated into the design, the resulting inherent degradation in performance does not affect the quality of the robot control solution beyond that of reducing control performance to unenhanced levels.

We have developed a robot control architecture that provides improvement in speed performance and fault tolerance through the use of a shared memory multiprocessor. We developed several versions of the controller, representing points along the performance/fault tolerance curve for a given number of processors. A serial robot controller (SC) was developed as a benchmark for analysis of the three parallel con-
trollers developed. A program simulating a planar four-link robot was also developed for visual analysis of the robot control solution [13]. This simulation program was used with a serial controller for two other projects at Rice University. A motion planner was added to the controller by Deo [8]. The planner includes detection and avoidance of singular positions. Visinsky incorporated simulated sensors, along with detection of and recovery from sensor failures [34]. This work has been incorporated into the serial and parallel control codes described in this thesis.

The first parallel controller (PC) that we developed had no tolerance of processor failure. This controller provides maximum speedup (given the algorithm used) at minimum cost. The speedup for \( n \) processors performing the same work as one is ideally \( n \) times. However, the necessity of processor synchronization, as well as
other factors, limits the maximum attainable speedup. The speedup for this parallel controller using the above simulated robot was 1.53, for \( n = 4 \). With the foundation of a serial controller and a parallel controller with maximum speedup, controllers with varying levels of fault tolerance were developed and analyzed.

The first fault-tolerant controller (HFT) has the greatest amount of software redundancy. All of the processors calculate each element of their copies of the inertia matrix and the centrifugal and Coriolis torque vector (see Section 2.1). Each processor then compares each element of its private copy of the inertia matrix to those of the other processors. The results of the comparisons determine which, if any, processors are faulty.

The second fault-tolerant controller (LFT) has less redundant software than the first. Each processor calculates the diagonal of the inertia matrix (\( n \) elements, where \( n \) is the number of joints). After this data is compared, the functioning processors calculate the row elements of the matrix and vector in parallel, as done in the parallel controller without fault tolerance. As fewer elements are compared, the likelihood that a disagreement (failure) may be overlooked increases. Thus, this controller offers better control performance than the first at the cost of using less information to detect a processor failure (a lower level of fault tolerance). Our results indicate that fault-tolerant parallel robot controllers can provide comparable performance to serial controllers, with costs that depend on the level of fault tolerance required. Analysis of the relative performance of the controllers developed determines the amount of
synchronization and redundant data required for detection of a processor failure. This information aids in the development of a rule by which the most efficient balance of performance and processor fault tolerance may be determined for a given general processor architecture.

Some of the design considerations in development of our control architecture are discussed in Chapter 2. The choices for the dynamic model of the robot are explained. In addition to the dynamic model, several issues relevant to our architecture are discussed: parallel robot control, shared memory multiprocessors, hardware fault tolerance, and software fault tolerance. Performance, level of fault tolerance, and system cost are interdependent factors of design for parallel fault-tolerant robot control architectures. This thesis provides a comparative analysis of parallel robot control architectures with varying levels of processor fault tolerance. Multiprocessor architectures have been considered mostly for their performance improvement over serial robot control architectures. However, some multiprocessor controllers have considered processor fault tolerance, as well. Chapter 3 summarizes some of the related work in the areas of parallel robot control and processor fault tolerance in robot controllers.

The two robot controllers without fault tolerance, SC and PC, are detailed in Chapter 4. Comparison of these two controllers shows the speedup made possible by the coarse grained parallelism incorporated in PC. Chapter 5 discusses the two fault-tolerant controllers, HFT and LFT, which exhibit the trade-off between performance
and fault tolerance. The cost factor was eliminated by the decision to not use spare processors.

Limitations on joint motion were added to our simulation primarily to be able to correctly model a physical robot. A path is planned for a robot based on its current position and its desired destination. Failure of a joint or the presence of an obstacle may prevent safe movement along the path mapped by the planner. Therefore, it is necessary to have some control over the range of motion of the joints. Another result of a joint failure is that the processor corresponding to the failed joint no longer has to control it. Thus, it could possibly be used to do some other work to increase the speed performance of the system. The addition of joint motion limits makes possible some future experimental work in processor reconfiguration upon joint failure. These additions are described in Chapter 6.

In Chapter 7, the relative performance of the four controllers is analyzed. This chapter also points out the areas where the efficiency of our architecture may be improved and evaluates some of the architectural alternatives to our design. The effects of these alternatives on performance and fault tolerance are explored. One of the goals of our future work is to produce general criteria for making the design decisions for processor fault tolerance, with given cost and performance constraints. Chapter 8 outlines some of our future research. Chapter 9 concludes with a brief summary of what we strove to accomplish and our results.
Chapter 2

Background

2.1 Robot Control

A basic robot (Figure 2.1) consists of links, joints connecting the links, and an end effector. The joints may be prismatic, providing linear motion, or revolute, allowing rotational motion. The number of joints the robot has determines the degrees of freedom of the robot. The position of each link is measured in the frame of reference of the previous link. The union of the set of points that can be reached by the links defines the workspace of the robot. Attached to the joints are motors for moving the joints and sensors that can monitor joint position and velocity.

A path planner maps a path for the robot. This plan is used by the controller, along with knowledge of the current position of the robot, to determine the force/torque necessary to move each joint. These forces are communicated to the motors. The sensors feedback position and velocity information to the planner/controller.
2.1.1 Dynamic Model

In order to control a robot efficiently, its dynamic properties must be known. Most robots use closed-loop control: the controller sends commands to the robot, it moves, and sensory information is returned from the robot to the controller to identify its new position. The dynamic equations characterizing a robot may be expressed by the Lagrangian formulation or by the Newton-Euler formulation. The major difference between the two forms is in how the robot is viewed. In the Lagrangian form, the robot is analyzed as a whole. This formulation defines a general relationship between the angular positions of each joint and the torques required to move them. It provides closed-form equations for the relationship. However, in the Newton-Euler form, each
link is considered separately. Since the links are coupled, the equations for each link contain coupling forces that are also in the same equations for neighboring links. The Newton-Euler equations are recursive, and define a numerical position/torque relationship at a particular time [32]. This is convenient for fast simulation, but lacks the closed-form nature desired for controller analysis and design.

The Lagrangian equations are used for our model. The dynamic model takes the following form:

\[
\tau = [M(\bar{\theta})]\ddot{\bar{\theta}} + \bar{N}(\dot{\bar{\theta}}, \dot{\bar{\theta}}) + \bar{G}(\bar{\theta}) + [V]\ddot{\theta}
\]  \hspace{1cm} (2.1)

where \( \bar{\theta} \) is the \( n \times 1 \) vector of joint angles for an \( n \)-joint robot, \( \bar{\tau} \) is the \( n \times 1 \) joint torque vector, \([M]\) is the \( n \times n \) inertia matrix, \( \bar{N} \) is the \( n \times 1 \) Coriolis and centrifugal torque vector, \( \bar{G} \) is the \( n \times 1 \) gravity torque vector, and \([V]\) is the \( n \times n \) viscous friction coefficient matrix [32]. The inertia matrix, \([M]\), and Coriolis and centrifugal torque vector, \( \bar{N} \), characterize the robot dynamics. Their equations are functions of the parameters and position/velocity of the robot. The torque vector contains the forces needed to move the robot. The two components of motion control are 1) determining the forces necessary to follow a planned path, and 2) rejecting disturbances not accounted for in the dynamic model.

We consider in this thesis the classical "computed torque" robot controller (basically local linearization plus a P(osition)D(erivative)-compensator) [32]. The dynamic equations of a robot are nonlinear due to inertial loading, coupling between neighboring joints, and the gravitational loading of the links [19], and are very difficult to
solve. The PD-compensator provides feedback to minimize the effects of steady state and tracking errors encountered in motion control. The position (or proportional) gain, \([K_P]\), reduces the position errors, and the derivative gain, \([K_D]\), reduces the errors in the velocity. Oscillation in the motion of a robot before reaching steady state is controlled by the damping factor. Position and derivative gains are calculated for critical damping to produce the fastest non-oscillatory response. Thus, the torque is calculated as:

\[
\tau = [M(\theta)]\{\ddot{\theta}_d + [K_D](\ddot{\theta}_d - \ddot{\theta}) + [K_P](\dot{\theta}_d - \dot{\theta})\} + \dot{\theta} + \ddot{\theta} + \bar{G}(\theta) + [V]\ddot{\theta} \quad (2.2)
\]

for \(\theta \in \mathbb{R}^n\), \(M \in \mathbb{R}^{n \times n}\), \(K_D \in \mathbb{R}^{n \times n}\), \(K_P \in \mathbb{R}^{n \times n}\), \(N \in \mathbb{R}^n\), \(G \in \mathbb{R}^n\), and \(V \in \mathbb{R}^{n \times n}\)

and where \(\ddot{\theta}_d\) is the desired trajectory.

### 2.1.2 Multiprocessor Robot Controllers

Most computer robot controllers are uniprocessor systems. With increased functionality of robots (particularly high speed and close control operations), control algorithms have become much more compute-intensive. It has become increasingly difficult to produce real-time responses [20, 12]. The advent of multiprocessor architectures has opened new avenues for the real-time control problem. The kinematics and dynamics calculations involved in robot manipulation are well suited for parallelism and are therefore easily adaptable to multiprocessor architectures. Numerous parallel algorithms for solving the kinematics and dynamics have been written using the Lagrangian and the Newton-Euler equations. These algorithms, some of which are
reviewed further in the next chapter, exploit parallelism at various levels. Task level parallelism could be exercised by performing the calculation of the inertia matrix in conjunction with calculation of the centrifugal and Coriolis torque vector. Parallelism at the instruction level would be calculation of different rows of a matrix at the same time. Due to the interdependence of data in a robot control system, efficient use of parallelism is also affected by the memory architecture.

2.2 Shared Memory

A parallel architecture would typically have either shared memory, with a common address space, or distributed memory, with a disjoint address space. These two memory models are depicted in Figure 2.2. In shared memory multiprocessors, cooperating processes communicate through shared data in memory. Every processor can write to and read from all memory locations. Therefore, no special considerations must be made for ensuring communication between processors. In a distributed memory environment, each processor has direct access to only a portion of the system memory. If the memory is not mapped to a common space, then for one processor to obtain data from the memory module accessed by another processor, some form of message passing must take place between the requesting processor and the one that has access to the information. This explicit communication is more difficult for the programmer to work with than the implicit communication of a shared memory environment. In this work, we consider a shared memory architecture.
Figure 2.2  Multiprocessor Memory Models
A shared memory multiprocessor environment is easier to program in because all data is available to all processors. To the programmer, this is very much like the sequential programming environment with which they are familiar. An important difference between a shared memory multiprocessor architecture and a uniprocessor architecture is that in the multiprocessor environment, it is possible for multiple processors to attempt to update the same memory location at the same time. The programmer must take some care to ensure that this does not happen. Therefore, synchronization of processors is required at times. This is one of the reasons that linear speedup cannot be achieved when a program being executed on a uniprocessor is moved to a multiprocessor.

Most shared memory multiprocessors do not scale well to large numbers of processors. This lack of scalability is primarily due to bus contention and memory latency. A typical robot would have fewer than ten joints. Thus, for a robot controller with one processor per joint, there would likely be no more than ten processors. Hence, the contention and latency problems often associated with large shared memory multiprocessors do not affect performance in our application. Our reasons for choosing a shared memory architecture are explained further in the next chapter.

2.3 Processor Fault Tolerance

A main emphasis of this thesis is tolerance of processor failure in the control architecture. A processor failure is caused by an error, which is the result of a fault
existing in the system. The fault is an incorrect condition of the system. This condition may be the result of a design flaw, deterioration, component failure, software error, or a number of other causes. Processor faults can be permanent (fail once and remain inoperable), intermittent (failures that randomly come and go), or transient (a "glitch"). Transient and intermittent faults are more prevalent than permanent faults. Unfortunately, they are also more difficult to detect because they may not exist when the resulting failure is detected. In this work, we consider all of the above types of faults. Detection of and recovery from processor failure requires some form of redundancy [27]. Redundancy in the software that performs the control calculations allows detection of processor failures through data comparisons, and recovery by having the same program on a different processor to assume the work of the failed processor. Hardware redundancy allows recovery from processor failures by moving the work from a failed processor to a working processor. Failure cannot be detected if multiple processors are not producing the same information. These options are further explained below. In Chapter 5, details and motivation of the fault tolerance techniques chosen are explained.

2.3.1 Hardware Fault Tolerance

Various processor configurations can provide different levels of fault tolerance. The number of processors available to do work determines the number of processor failures that can be tolerated. Having multiple processors working together and comparing
iterative results is one way of detecting processor failures. The processors working together are called the “working set”. When a failure occurs, the working set size is reduced. This is called “time redundancy” because no new hardware is added when a processor fails [5]. Instead, the existing hardware is reconfigured. The number of processor failures that can be tolerated using time redundancy is m, where n is the number of processors originally in the working set, and n − m = 2. This allows two processors to continue working and comparing. When a disagreement occurs with only two processors in the working set, it is not possible to determine which processor is faulty. There are two options at this point: 1) the robot can discontinue operation, or 2) one processor can be chosen to continue working based on a log of previous maintenance or results of self-testing by each processor [27]. Either option is no worse than the case for a uniprocessor controller.

For some reasons it may be necessary to have spare processors:

- if the size of the working set must remain constant
- if tolerance of a larger number of failures than n − 2 is desired
- if the per-processor workload is heavy.

Processors that remain powered off until a working processor fails are termed cold spares. There can be one or more cold spares for every hot processor (those in the working set). When an operating processor is determined to be faulty, a cold spare is brought online to replace it. Once it is brought online, it must be synchronized with
the working set. Another option is to have only one or two total cold spares in case of failure of a working processor [24].

Hot spares are processors that perform the same work as those in the working set, but do not participate in the data comparisons and do not control any joints. Since the hot spares are working along with the members of the working set, if each processor in the working set is doing different work (for example, calculating a different row of a matrix), then there must be at least one hot spare for each member. When a working set member fails, a corresponding hot spare joins the working set in its place. As with cold spares, it is possible to have more than one hot spare per hot processor [24].

The advantage of cold spares over hot spares is that they are not using any power doing unused work. It is more economical to use cold spares if the reconfiguration time for the working set is not critical. This is because bringing a cold spare online and into the same state as the other members of the working set takes more time than bringing a hot spare into the working set. When time is the most important factor to consider, hot spares have the advantage of already being online and in the same state as the working set.

We have chosen to have no spare processors in this architecture. The working set size is initially $n$, where $n$ is the number of joints (degrees of freedom) of the robot. This is the most economical option, since it utilizes the smallest number of processors, and it is easier to implement than a cold spare configuration.
2.3.2 Software Fault Tolerance

Software redundancy is another means of detecting processor failures. A simple way to do this is to have all processors execute the same code to perform an operation. As long as all processors are fault-free, they should produce the same results. The results of intermediate calculations are compared, and a working set is derived for sending data to the robot. The number of comparisons that take place per iteration is dependent upon the level of fault tolerance desired. Exact agreement in the data must be maintained in this case [18].

Another option is to have each processor perform the same operation, but in a different way. For a robot controller system, for example, the processors could receive input from different types of sensors. Each would make calculations based on the data received from its corresponding sensor. This approach takes into account the possibility of generic faults in the software by using different versions of the software to produce the same result. This programming option may also be used to mask sensor failures [34]. Using this approach, the working processors may not all produce exactly the same results. Thus, some type of convergent voting algorithm must be used to maintain approximate agreement. Allowing software (and hardware) design diversity complicates fault detection, as well as system management [18].

In this thesis, each processor executes the same code for software fault tolerance. Checking for exact agreement in the comparisons takes less time than checking for
approximate agreement because it requires fewer instructions. Also, only one program needs to be maintained for this fault tolerance scheme.
Chapter 3

Related Work

3.1 Parallel Computing in Robotics

Parallel computing has been used for its speedup advantages in a number of robot controller architectures [1, 14, 25, 26, 35]. The performance improvement is achieved using algorithms with parallelism at varying levels. Such algorithms have been written to solve the forward and inverse dynamics [6, 10, 23, 33, 36], as well as the kinematics [35, 38]. Smaller tasks, such as computation of the inertia matrix, have also been parallelized [9, 21].

3.2 Multiprocessor Robot Control Architectures

3.2.1 Message-passing

One of the variations in controller architectures is in the memory structure (data communication). Due to the interdependence of joint information in robotics control, all processors must have access to data pertaining to every joint. Both shared memory multiprocessors and message-passing multiprocessor architectures have been utilized as robot controllers.
At Stanford University (J.B. Chen, et al.) [7], a multiprocessor manipulator controller has been developed that uses message-passing for data communication. It has seven National Semiconductor (NSC 32016) single board computers, which are responsible for the real-time computing. A SUN workstation host is essentially a server to the 32K computers. A proposed use of the system was to control the Stanford/JPL hand. The hand has three fingers, each with three degrees of freedom. Each finger would have two dedicated computers, one for tactile processing and one for implementing the force control loop. The seventh computer would be a shared resource for the three fingers, handling the joint servo processing [7].

3.2.2 Shared Memory

We utilize shared memory because our goal was to provide the greatest efficiency in a fault-tolerant system. A distributed private memory architecture would require explicit communication between processors due to the interdependence of data in robot control. This would adversely affect the response time of the controller. A group at the MIT Artificial Intelligence Laboratory (S. Narasimhan, et al.) [26] developed a shared memory architecture that uses message-passing to communicate data. However, when maximum efficiency is desired, pointers can be set up from processors to the memory of other processors to take advantage of the shared memory. Two versions of hardware were developed for their system. The Version II hardware provides some enhancements to the Version I hardware. A SUN-3 host connects to
the control processors (single board computers with floating point units) through a VME bus. A bus-to-bus adaptor allows transparent access from the host into the dual-ported shared memory on the control microprocessors. The control architecture has been used to control the Utah-MIT hand [26].

Some special-purpose architectures have been developed for robot control. At the University of California at Santa Barbara (Y. Wang, et al.) [4, 37], a special-purpose robot controller architecture was developed. They identify four levels of parallelism:

- job/program level
- task/procedure level
- interinstruction level
- intrainstruction level

Their architecture utilizes job, procedure, and intrainstruction level parallelism. The maximum configuration consists of one host processor, four sensory subsystems, and four robotic processors, each with five dual-axis servo controllers and the necessary interfacing hardware. The robotic processor is the main computational unit. The memory of each control processor is mapped into the master processor’s address space in this architecture, to provide distributed shared memory. The Recursive Newton-Euler equations are evaluated for the inverse kinematics and inverse dynamics [4, 37].

Andersson points out in his survey of robot control architecture options [1] that overspecialization could hinder performance. Tasks that cannot be performed by the
specialized processor must be deferred to a more general-purpose processor. The performance of the system is then still dependent upon a general-purpose processor. Andersson's architecture is a special-purpose system that attempts to avoid over- and underspecialization. The three efforts in the design of this system were to:

1. keep operand and result bandwidth high

2. minimize work that will dominate the timing, such as data transfers

3. be able to run substantial amounts of less structured code.

The memory is physically replicated four times, so that both arguments can be fetched for both ALU's at the same time [1]. This is done to ensure that the two ALUs are always busy.

3.3 Fault Tolerant Multiprocessor Robot Controllers

Most of the literature on parallel robot controllers that we have studied describes architectures that utilize multiple processors only to achieve speedup. None of the systems described above offer fault tolerance of any kind. However, many of the uses of robots today necessitate a fault-tolerant architecture. In a chapter of Fleming's book on transputers [11], Jones and Fleming cite several applications of transputers for control. One of the applications is for parallel control of a robot. The architecture maintains a "processor farm", a number of processors waiting to be allocated a task by the master processor. The authors note that the processor farm architecture allows
tolerance of processor failures. However, they do not perform any simulations that utilize this fault tolerance in their system [11].

Bejczy and Szakaly [3] have developed an architecture for controlling space telerobots at the Jet Propulsion Laboratory. It was designed to be a general architecture that would meet the needs of all elements of space telerobots. Self-test and diagnostic capabilities are provided at the motor drive level as part of a higher level failure diagnosis and error recovery scheme. These failure tolerance procedures increase the reliability of the system. However, there is no processor fault tolerance. The memory is shared to simplify processor synchronization.

Some multiprocessor architectures with processor fault tolerance have been developed for various uses, including real-time computing. One such architecture, MAFT, was developed by Kieckhafer, et al. [18]. MAFT was designed to provide high performance and reliability for a wide range of real-time applications. The performance requirements for a commercial flight-control system were the minimum design goals for this architecture. This design separates executive functions (such as internode communication and synchronization, data voting, error detection, task scheduling, and system reconfiguration) from applications functions (such as reading sensors, sending commands to actuators, and performing control calculations). Each node consists of one special purpose processor for the executive functions (OC) and one application program processor (AP). There is a separate data memory for each node's executive function processor. These memory modules contain exactly the same information
(data redundancy). Information is communicated between nodes via message-passing. For fault tolerance, the architecture allows Byzantine and Approximate Agreement in order to support the use of multiversion hardware and software. All messages received by an OC are passed through the Message Checker where they are subjected to physical and logical checks before being passed on to the appropriate subsystem. The four types of messages (data, scheduling, synchronization, and error-management) undergo other checks in their subsystems. For example, data messages undergo "reasonableness checks" before being voted on. The voting is on-the-fly, meaning that as new data is received, it is compared with data already at the node. When the paper was published, two prototypes were being implemented. Four of six planned nodes had been assembled and operated as a system for one version of MAFT. The architecture will support up to eight nodes [18].

There are several differences between our design and MAFT. One important difference is that MAFT has replicated memory for fault tolerance, and message-passing. This memory architecture has the difficulty and overhead of communicating data between nodes. Also, due to memory replication, maintaining consistency in all copies is an added task. Our shared memory architecture provides an easier programming environment. However, if the required level of fault tolerance necessitates memory replication, we could add message-passing capability on top of the shared memory, as the group at the MIT AI Lab did (see section 3.2.2). The MAFT architecture supports up to eight nodes. Each node has a task scheduler that schedules tasks for
all nodes, for maintaining consistency. However, each node takes assignments only from its own task scheduler. In our controller, each processor has its own program running. Thus, there is no need for the task scheduling overhead. The major difference between MAFT and our architecture is that MAFT was designed to support a variety of applications. Providing the generality required for such an effort may complicate, and thus slow some functions. Our architecture was designed specifically to control robots.

Ozguner and Kao [28] have explored using a parallel architecture to achieve a processor fault-tolerant robot control system. Their architecture is more specialized than that proposed here, but it also utilizes the notion of voting among processors. There are four Intel 86/30 single board computers, which can be configured in three modes of operation: triple modular redundancy, duplex, and simplex. There are four buses over which data is communicated between the computers. The system has some shared memory which is only used to communicate simulation data to and from their simulation of the OSU Hexapod (a six-legged experimental vehicle). The architecture uses hardware comparators to compare data. Hardware comparators were chosen to speed up the data throughput. Voting to determine which processors, if any, are faulty is handled in software. Once a processor is considered faulty, it is not utilized again until it has been repaired [28].

Our architecture is similar to Ozguner and Kao's, in the manner in which we detect processor failures. However, ours is a shared memory architecture, whereas
they only use shared memory for simulation purposes. Our architecture scales with
the size of the robot (rather than being limited to a maximum configuration of four
processors). Also, to allow for intermittent processor failures, each processor performs
calculations and participates in the determination of the working set in each iteration.
Thus, processors can be readmitted into the working set in our architecture without
user intervention.
Chapter 4

Robot Controller Implementation

4.1 Architecture

We began our work with a serial controller as a benchmark for our performance analyses. For this case, only one processor is utilized to control the robot and perform the necessary simulation. A parallel controller without fault tolerance followed this. The processor architecture for this controller consists of one hot processor per joint, for an \( n \)-joint robot, and one designated processor for initialization and execution of simulation code. Each of the processors commands one joint of the robot. A block diagram of the parallel architecture used is shown in Figure 4.1.

We employ a logically shared memory architecture, allowing direct access to all data by each processor and eliminating the need to explicitly move data during the computation of the robot control problem solution. The underlying memory may be physically distributed (by employing distributed shared memory techniques), or simply replicated so that the memory subsystem will not represent a single point of failure. In a distributed shared memory system, each processor has a piece of the total address space locally. This is the portion that is accessed most often by that processor. Since each processor must have access to the current position of each joint
Figure 4.1 System Configuration
of the robot, distributed private memories would require some communication among the processors.

The machine used is a Sequent Symmetry S81. It has twenty Intel 386 processors and 80 Mbytes of memory, along with the Weitek WTL1167 floating point accelerators. It also has a 53 Mbyte/sec pipelined system bus. The operating system used is DYNIX, a version of UNIX 4.2BSD. When code is being executed on multiple processors, there is one parent processor and one or more child processors. Each processor has a private cache, using the copyback coherence scheme. The write-through coherence scheme is supported for backward compatibility with the Sequent Balance [22].

4.2 Robot Control without Processor Fault Tolerance

The foundation for the multiprocessor controllers developed is a serial controller, written in the C programming language. The user provides the initial position of the robot in the form of joint angular positions, the desired destination of the end effector, and some other initialization information. A planner, developed by Deo [8], maps the desired path of the robot, carefully avoiding singular configurations, which are configurations subject to both physical and numerical instability [32].

A simulation tool was developed for observing and testing our robot controllers. A Lagrangian dynamics formulation is used to characterize the model due to the closed-form nature of the equation. The architecture presented here is part of our larger
effort in robot fault tolerance, in which we seek to reconfigure both the computational paths and the control strategy in response to faults [13]. In particular, we wish to exploit the structure of the dynamics by modifying the model/controller in real time in response to joint failures, while also tolerating processor faults. The explicit closed-form expressions given by the Lagrangian formulation are more convenient for this purpose than a recursive Newton-Euler type dynamics formulation, in which the structure is implicit, but less accessible to the controller. We therefore wish to explore the extent to which our controller can be successful with Lagrangian dynamics. The robot model used currently is a planar four-link arm with a free base. The links of the arm are cylinders with spherical ends. Since the robot model is planar, the gravity vector is orthogonal to the plane of motion of the arm. Thus, there are no gravity torques to consider. The viscous friction component is also neglected in our implementation, to simplify the model. Hence, Equation 2.2 becomes:

$$\tau = [M(\bar{\theta})](\ddot{\theta} + [K_D](\ddot{\theta} - \bar{\theta}) + [K_P](\dot{\theta} - \bar{\theta})) + \bar{N}(\bar{\theta}, \dot{\theta})$$  (4.1)

For this robot model, expressions for the elements of the inertia matrix and the centrifugal and Coriolis torque vector have been determined by Seraji and Colbaugh, and are shown in Tables 4.1 and 4.2 [30]. We chose diagonal gain matrices, $[K_D]$ and $[K_P]$, to decouple the tracking errors experienced by the robot. Each joint of the simulated robot has two sensors, a tachometer and an encoder, and one motor. In the simulation, we maintain calculated joint positions, based solely on the torques calculated, and "actual" positions, based on simulated sensor information. Sensor
\[ M[0][0] = 73.33 + 50 \cos(\theta_1) + 30 \cos(\theta_2) + 30 \cos(\theta_1 + \theta_2) + 10 \cos(\theta_3) + 10 \cos(\theta_2 + \theta_3) + 10 \cos(\theta_1 + \theta_2 + \theta_3) \]
\[ M[0][1] = 40 + 25 \cos(\theta_1) + 15 \cos(\theta_1 + \theta_2) + 30 \cos(\theta_2) + 10 \cos(\theta_3) + 10 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) \]
\[ M[0][2] = 16.67 + 10 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) + 15 \cos(\theta_1 + \theta_2) + 15 \cos(\theta_1 + \theta_2) \]
\[ M[0][3] = 3.33 + 5 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) \]
\[ M[1][0] = 40 + 25 \cos(\theta_1) + 15 \cos(\theta_1 + \theta_2) + 30 \cos(\theta_2) + 10 \cos(\theta_3) + 10 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) \]
\[ M[1][1] = 40 + 30 \cos(\theta_2) + 10 \cos(\theta_3) + 10 \cos(\theta_2 + \theta_3) \]
\[ M[1][2] = 16.67 + 10 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 15 \cos(\theta_2) \]
\[ M[1][3] = 3.33 + 5 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) \]
\[ M[2][0] = 16.67 + 10 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) + 15 \cos(\theta_2) + 15 \cos(\theta_1 + \theta_2) \]
\[ M[2][1] = 16.67 + 10 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 15 \cos(\theta_2) \]
\[ M[2][2] = 16.67 + 10 \cos(\theta_3) \]
\[ M[2][3] = 3.33 + 5 \cos(\theta_3) \]
\[ M[3][0] = 3.33 + 5 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) + 5 \cos(\theta_1 + \theta_2 + \theta_3) \]
\[ M[3][1] = 3.33 + 5 \cos(\theta_3) + 5 \cos(\theta_2 + \theta_3) \]
\[ M[3][2] = 3.33 + 5 \cos(\theta_3) \]
\[ M[3][3] = 3.33 \]

**Table 4.1** Inertia Matrix Expressions
\[ N[0] = -50 \dot\theta_0 \dot\theta_1 \sin(\theta_1) - 30 \ddot\theta_0 \dot\theta_2 \sin(\theta_2) - 30 \dot\theta_0 (\dot\theta_1 + \dot\theta_2) \sin(\theta_1 + \theta_2) - 10 \dot\theta_0 \ddot\theta_3 \sin(\theta_3) - 10 \dot\theta_0 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 10 \dot\theta_0 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \dot\theta_1 (\dot\theta_1 + \dot\theta_2) \sin(\theta_1 + \theta_2) - 30 \dot\theta_1 \ddot\theta_2 \sin(\theta_2) - 10 \dot\theta_1 \dot\theta_2 \sin(\theta_3) - 10 \dot\theta_1 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 10 \dot\theta_2 \dot\theta_3 \sin(\theta_3) - 5 \ddot\theta_2 (\dot\theta_1 + \dot\theta_2) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \dot\theta_2 \dot\theta_3 \sin(\theta_2 + \theta_3) - 5 \ddot\theta_3 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 5 \dot\theta_2 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \dot\theta_1 \ddot\theta_2 \sin(\theta_2) - 5 \ddot\theta_2 \dot\theta_3 \sin(\theta_3) - 5 \dot\theta_3 \ddot\theta_3 \sin(\theta_3) - 15 \dot\theta_3 \ddot\theta_3 \sin(\theta_3) - 5 \ddot\theta_3 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 5 \ddot\theta_3 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 5 \ddot\theta_3 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \ddot\theta_3 \sin(\theta_3) - 25 \dot\theta_3 \dot\theta_3 \sin(\theta_3) \\
N[1] = -30 \ddot\theta_0 \dot\theta_2 \sin(\theta_2) - 10 \dot\theta_0 \ddot\theta_3 \sin(\theta_3) - 10 \dot\theta_0 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 5 \dot\theta_0 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \dot\theta_1 \ddot\theta_2 \sin(\theta_2) - 30 \ddot\theta_1 \dot\theta_2 \sin(\theta_2) - 10 \dot\theta_1 \dot\theta_2 \sin(\theta_3) - 10 \dot\theta_1 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 10 \dot\theta_2 \dot\theta_3 \sin(\theta_3) - 5 \ddot\theta_2 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 15 \dot\theta_2 \ddot\theta_3 \sin(\theta_2) - 5 \ddot\theta_3 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 5 \ddot\theta_3 (\dot\theta_1 + \dot\theta_2 + \dot\theta_3) \sin(\theta_1 + \theta_2 + \theta_3) - 15 \dot\theta_3 \ddot\theta_3 \sin(\theta_3) - 25 \ddot\theta_3 \sin(\theta_3) + 15 \dot\theta_1 \dot\theta_2 \sin(\theta_1 + \theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_1 + \theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_1 + \theta_2 + \theta_3) + 30 \dot\theta_0 \dot\theta_1 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_2 + \theta_3) + 15 \dot\theta_0 \ddot\theta_1 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_1 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) \\
N[2] = -10 \ddot\theta_0 \dot\theta_3 \sin(\theta_3) - 10 \dot\theta_0 \ddot\theta_3 \sin(\theta_3) - 10 \dot\theta_0 (\dot\theta_2 + \dot\theta_3) \sin(\theta_2 + \theta_3) - 5 \dot\theta_0 \ddot\theta_3 \sin(\theta_3) + 15 \dot\theta_0 \dot\theta_3 \sin(\theta_3) + 15 \dot\theta_0 \ddot\theta_3 \sin(\theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_1 + \theta_2 + \theta_3) + 15 \dot\theta_0 \ddot\theta_1 \sin(\theta_2 + \theta_3) + 30 \dot\theta_0 \dot\theta_1 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_1 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) \\
N[3] = 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_2 \sin(\theta_1 + \theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_1 + \theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_2 + \theta_3) + 5 \ddot\theta_0 \dot\theta_3 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_1 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) + 10 \dot\theta_0 \dot\theta_2 \sin(\theta_2 + \theta_3) \\

Table 4.2  Centrifugal and Coriolis Torque Vector Expressions
and motor failure, detection, and isolation have been incorporated by Visinsky [34]. Also, some modeling error is injected in the simulation program to model parameter uncertainty more closely. The error is introduced in both the inertia matrix and the centrifugal and Coriolis torque vector by changing the first element of the matrix and vector by a random value.

A Silicon Graphics Personal Iris workstation using the Trick/MAGIK graphics simulator, which was developed at NASA/Johnson Space Center [2], is used to simulate and display the robot arm. Trick provides a dynamic simulation environment that allows the user to avoid the executive code development involved in simulation development tasks. Thus, the user must only develop and maintain the math model needed for his simulation. Much of the code needed to generate and run a simulation is generated within Trick. This allows faster development of simulation tools. The graphics control program, TDM, is part of the "Third Party Software" provided. The working environment for our experiments is shown in Figure 4.2. Pictures of the simulation display for one program run are shown in Figures 4.3 - 4.5. The small circle in the display marks the desired destination (target) for the end of the robot arm. Processor status indicators were added to the display for the simulations of processor fault tolerance. The four squares represent processors $P_0 - P_3$.

The sequence of events of the serial controller (SC) is maintained in the parallel versions. A task graph for SC appears in Figure 4.6. Tasks 2, 3, and 4 are the controller tasks – the only tasks considered for parallelizing. The first parallel controller
developed (PC) has no fault tolerance incorporated. The parallelism is incorporated at the instruction level. Rather than executing multiple tasks in parallel, individual tasks are divided and the partitions are executed in parallel. The task graph of this controller, shown in Figure 4.7, illustrates this parallelism. Each processor calculates one row of the inertia matrix, one element of the centrifugal and Coriolis torque vector, and finally one element of the torque vector. For an $n$-joint robot, the dimensions of the matrices and vectors used in the dynamics calculations are $(n \times n)$ and $(n \times 1)$, respectively. Therefore, the coarse grained parallelism used here ensures that at least $n$ processors are used in parallel. This task decomposition does not provide the greatest amount of parallelism possible with the algorithm used. It
Figure 4.3  Model Display: Beginning of Demonstration with No Failures
Figure 4.4  Middle of Demonstration with No Failures
Figure 4.5  End of Demonstration with No Failures
Task 0: Initialization

1: Inverse kinematics
2: Calculate inertia matrix elements
3: Calculate centrifugal and Coriolis force vector elements
4: Calculate torque vector elements
5: Calculate angular positions (simulation)

Figure 4.6 Serial task decomposition
Task 0: Initialization

1: Inverse kinematics
2: Calculate one row of inertia matrix
3: Calculate one element of centrifugal and Coriolis force vector
4: Synchronize
5: Calculate one element of torque vector
6: Calculate angular positions (simulation)

Figure 4.7 Parallel task decomposition (without fault tolerance)
simply takes advantage of the obvious parallelism present. There are subtasks which could be parallelized, such as cos and sin calculations. Computation of $[M]$ and $\bar{N}$ could be distributed more evenly among the processors. The first row of $[M]$ and the first element of $\bar{N}$ require more calculations than the remaining rows and elements, respectively (see Tables 4.1 and 4.2). In future work, we will look for more efficient ways to parallelize the control algorithm.

Only the control code is timed for performance analysis. Benchmarking begins just after the planner maps a usable path, and ends just after the torque vector elements are calculated. The speedup occurs within tasks 2, 3, and 5 in Figure 4.7. When work is being done in parallel, it is sometimes necessary to synchronize the processors at various points of execution. The synchronization added to the parallel controller reduces the performance of the parallel code to less than $n$ times faster than the serial code, where $n$ is the amount of parallelism incorporated (number of processors performing in parallel).
Chapter 5

Implementing Processor Fault Tolerance

Once the control code was parallelized, some modifications were made to the controller to support tolerance of processor failures. As stated in Chapter 2, there are several processor architecture options available. The base architecture still consists of one hot processor per joint, each controlling one joint of the robot. We employ no cold or hot spares. Use of hot or cold spares would allow the working set size to remain constant, and therefore, the workload of each processor to remain constant. However, cold spares require additional time for initialization when brought online and time for ensuring that their state is the same as that of the other hot processors (already in the working set). The overhead of hot spares is due to their continued operation when they are not controlling the robot. Even when using spare processors, the effects of reducing the working set size must be considered, when all of the spares have been used. Therefore, we chose to implement time redundancy to determine the performance degradation resulting from processor failures.

When a processor fails, one of the other members of the working set assumes control of the joint corresponding to the failed processor. This is based on the assumption that the faulty processor will recognize that it is not a member of the working set
(i.e. that it will not experience read errors). The processor will read the element of the working set suggestions of all hot processors that corresponds to its status. If fewer than \((n/2 + 1)\) of \(n\) processors have included the processor, then it does not perform any more computations until the next iteration. A more robust design would include an error detection or correction circuit with some disabling circuitry between the outputs of each processor and the system bus, to ensure that the output of a faulty processor is not passed to the robot [27]. Another possibility is to use a master processor to pass the outputs of the working processors to the robot. This processor would receive input from all working processors, and check the working set suggestions for each before passing its data on to the robot. Then, some provision for fault detection in this processor would have to be added to the system.

A round-robin reconfiguration scheme determines which processor assumes the work of the failed processor.

**Algorithm 5.1** For \(n\) processors, processor \(a\) (a member of the working set), takes the place of processor \((a + 1) \mod n\) when it fails. Then, processor \(a\) continues checking processors \((a + i) \mod n\), for \((1 < i \leq (n - 1))\), until it encounters another member of the working set.

In other words, each processor remaining in the working set first calculates the torque for its corresponding joint. Then, as shown in Figure 5.1, it reads the working set suggestions set by all available processors for a neighboring processor. If that processor is no longer considered a member of the working set, the torque for its correspond-
The working processor will continue checking processors in the same direction and calculating corresponding torque elements until it encounters another member of the working set. Derivation of the working set suggestions is explained in Section 5.1. The example illustrated in Figure 5.2 shows the result when processor $P_1$ fails, followed by failure of processor $P_0$. This round-robin checking ensures that each element of the torque vector is calculated by only one processor that is guaranteed to be fault-free. The scheme will handle failed processors interleaved with working processors, as well as consecutive failed processors. The processor reconfiguration scheme will tolerate up to $(n-1)$ processor failures at a time, leaving one processor working. However, all fault tolerance is lost once there are fewer than three processors available for data comparisons. The
disadvantage of this reconfiguration scheme is that it could result in an unfair distribution of work when consecutive processors fail. With a maximum of four processors in the working set, the uneven workload distribution is not as noticeable as it would be for a maximum of ten processors. The results recorded for two processor failures in Chapter 7 show the performance degradation when two consecutive processors have permanent failures for the entire run. This problem will be addressed in our future work (see Chapter 8).
5.1 Controller with High Fault Tolerance

The fault-tolerant control programs are written in the C programming language. The program user inputs the identity number of the processor to fail, the number of failures that the processor will suffer, and the start and end times of each failure. Processor failures are simulated by changing one element of the data that will be compared in the comparison routine.

When considering fault tolerance, we chose to have all processors calculate the redundant data the same way and accept only exact agreement to make fault detection easier. The processors vote to determine which processors are still functioning properly and should remain in the working set. Since some processor failures are transient or intermittent, each processor always performs some tasks, even though it may not be a member of the working set and may still be faulty. Voting always occurs before output is considered valid, and a processor determined to be faulty does not produce any output.

The first fault-tolerant controller developed (HFT) is a fault-tolerant version of the serial controller. The processors working in parallel all perform exactly the same algorithm up to calculation of the torques. Each processor calculates its own version of $[M]$ and $\hat{N}$, then compares each element of its own copy of $[M]$ (and $\hat{N}$, if desired) to those of each of the other processors. From the comparisons, each processor produces a comparison vector that indicates if there was disagreement for at least one element. Once a disagreement between two sets of data is noted (the comparison flag set), the
comparisons of the remaining elements in the two sets are irrelevant. Then a working set suggestion, based on this vector, is produced for each processor. The working set suggestion vectors can be derived from a lookup table, as shown in Figure 5.3. If the comparison vector of one processor indicates that its data did not agree with that of any other processors, then this processor does not include itself in its working set suggestion. When a processor excludes itself from the working set, it includes all other processors. If the data produced by one processor agrees with that of at least one other processor, then its working set suggestion includes itself and all other processors that produced the same result. A processor becomes a member of the working set if at least \((n/2+1)\) of \(n\) processors have included (voted for) that processor in their working set suggestions. Even if a faulty processor includes itself in its own working set suggestion, it will not be included in the working set because the working processors have not included it in their suggestions. Therefore, it will not have enough votes to be included (see Figure 5.4). In our model, if at least three of the four processors include a processor in their working set suggestions, then that processor will calculate the torque for its corresponding joint. This is based on the assumption that if two processors have failed, they will not have the same incorrect data. Thus, their comparison vectors will indicate disagreement with all other processors and their working set suggestions will include all other processors, as shown in Figure 5.5. Each of the working processors will have four votes and the two failed processors will each have one vote from each other. If a deadlock situation arises (a processors
for comparison vector \( (cv[0..3]) \): 0 = no differences
1 = at least one difference

for working set suggestion vector \( (ws[0..3]) \): 0 = failed
1 = member

<table>
<thead>
<tr>
<th>cv[0..3]</th>
<th>ws[0..3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1111</td>
</tr>
<tr>
<td>0001</td>
<td>1110</td>
</tr>
<tr>
<td>0010</td>
<td>1101</td>
</tr>
<tr>
<td>0100</td>
<td>1011</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
</tr>
<tr>
<td>0101</td>
<td>1010</td>
</tr>
<tr>
<td>0110</td>
<td>1001</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>

**Figure 5.3** Comparison Vector and Working Set Suggestion for Processor 0

![Comparison Vector Diagram]

**Figure 5.4** Compare and Vote Example with One Failure

![Compare and Vote Example Diagram]
Figure 5.5  Compare and Vote Example with Two Failures
reach one solution and a processors calculate a different solution), then the program is terminated with a message indicating a deadlock condition. A better way to handle this problem is to retry the redundant data calculations and compare and vote again. This alternative is discussed in Section 7.2.2.

As the number of processors increases, the working set decision becomes more difficult. For example, with a total of eight processors, having four processors in one working set suggestion, two in another, and two in a third, should result in the set of four being chosen as the working set, or a retry of calculations and compare/vote. However, none of the processors would perform any calculations, since none have at least \((n/2 + 1)\) votes, where \(n = 8\). Thus, this is a deadlock situation that should not occur. In later work, we will consider "majority rule" criteria for determining the working set.

In the task graph for this controller, shown in Figure 5.6, tasks 2 and 3 are the same as tasks 2 and 3 in the serial controller (Figure 4.6). Tasks 5 and 6 are added for fault tolerance. In Task 7, each processor calculates only one element of the torque vector, unless a neighbor processor has failed. In this case, a neighboring processor of the failed processors will calculate additional torques, as described in the first part of this chapter and shown in Figure 5.2. This fault-tolerant controller could be made faster by having each processor calculate its own version of \(\tilde{\tau}\) and compare those \(n\) numbers, rather than comparing the \(n^2\) elements of \([M]\). From its own version of \([M]\) and \(\tilde{N}\), each processor would calculate \(\tilde{\tau}\). The processors would compare these
Task 0: Initialization
1: Inverse kinematics
2: Calculate inertia matrix elements (4 matrices)
3: Calculate centrifugal and Coriolis force vector elements (4 vectors)
4: Synchronize
5: Compare inertia matrices &
   Vote to produce working set suggestions
6: Synchronize
7: Calculate one element of torque vector
8: Calculate angular positions (simulation)

Figure 5.6 Parallel task decomposition (for HFT)
values, and when a processor must calculate the torque for a neighboring joint, it will use its versions of $[M]$ and $\bar{N}$.

The processor status indicators are green when the processors are working properly and red when they have failed. The links also change color to indicate sensor and motor failures. Simulation examples with failed processors are shown in Figures 5.7 - 5.9. At the beginning of the run, all processors are functioning properly (Figure 5.7). Failure of processor $P_2$ (Figure 5.8) is indicated by the darker (red) color of the third processor. The processor's failure is intermittent, so the processor begins working again within the same run.
Figure 5.7  Model Display: Beginning of Processor Failure Demonstration
Figure 5.8 Failure of Processor 2
Figure 5.9  Recovery of Processor 2
5.2 Controller with Lower Fault Tolerance

The level of fault tolerance can be adjusted to improve the controller performance of our system. Reducing the amount of redundant data decreases the execution time of the dynamics solution. However, this also decreases the level of fault tolerance since less information is compared. The second fault-tolerant controller exhibits the trade-off between high performance and a high level of fault tolerance. Each processor calculates its own version of the diagonal of $[M]$. Then the diagonals are compared to detect processor failures, rather than comparing every element of the matrix. For an $n$-joint robot, this saves each processor $(n^2 - n)$ comparisons, as well as multiplications, additions, and cos/sin calculations. The comparison results are again used to produce the working set suggestions of the processors. After the suggestions are formed, the processors that are members of the working set calculate the rows of $[M]$ and the elements of $\tilde{N}$. As in the parallel controller without fault tolerance, when all processors are functioning properly, each one calculates one row of $[M]$ and one element of $\tilde{N}$. Using the round-robin checking method, each element of $[M]$, $\tilde{N}$, and $\tilde{r}$ is calculated by only one processor. This fault-tolerant controller provides the redundancy required for detection of a failure. However, the primary reason that it has better performance than the previous fault-tolerant controller is that fewer elements are compared. The task graph in Figure 5.10 shows that there is one more synchronization step here than in HFT. The speed performance of the controller is still better than that of HFT because fewer numbers are calculated in each iteration.
Task 0: Initialization
1: Inverse kinematics
2: Calculate diagonal elements of inertia matrix (4 sets)
3: Synchronize
4: Compare diagonal elements &
   Vote to produce working set suggestions
5: Synchronize
6: Calculate one row of inertia matrix
7: Calculate one element of centrifugal and Coriolis force vector
8: Synchronize
9: Calculate one element of torque vector
10: Calculate angular positions (simulation)

Figure 5.10 Parallel task decomposition (for LFT)
The calculation of the diagonal elements involves four serial calculations. There are also four serial calculations in task 6, and one serial calculation in task 7. In HFT, there are sixteen serial calculations in task 2 and four in task 3. Thus, the extra synchronization does not make the controller with a lower level of fault tolerance slower.
Chapter 6

Handling Joint Limits Dynamically

The path planner for a robot determines the incremental positions of the robot between its initial position and the final desired destination. Originally, all joints were allowed a full circle range of motion (±360 degrees). With no limitation set on the range of motion of the joints, it was possible for them to collide. We added limitations to the ranges of motion for each joint primarily to allow our simulator to be extended to real hardware. These limits also make joint failure compensation and obstacle avoidance possible. Due to some interference, a joint may not be able to follow its planned path. When a joint locks, due to a motor failure or a mechanical failure in the joint, a permanent elbow is created. This elbow causes a "virtual" link that has not been accounted for in the equations modeling the robot (see Figure 6.1). Therefore, the range of motion of the other joints may change due to the elbow. The motion limits change based on the current position of the failed joint. Figure 6.2 shows the reduced workspace of a planar robot with joint 2 locked. The joints are all capable of a full circle range of motion. The dotted line encircles the robot's workspace when all four links are able to move.
Figure 6.1  Locked Joint Example

Figure 6.2  Workspace After Joint Failure
If each joint is being controlled by one processor, then failure of a joint leaves the corresponding processor with less work to do. This "extra" processor may be used to improve the overall performance of the controller. If, for example, Processor 1 has failed and Processor 0 is performing its computations, the extra processor can take over the work of the failed processor to reduce the amount of serial work done by Processor 0. This reduces the total response time of the system. Addition of the joint motion limits will allow us to simulate joint failures for investigation of processor reconfiguration due to joint failure (see Chapter 8).

Incorporation of the limits must be accompanied by recursive path planning. Before a plan is passed to the controller in each iteration, the desired angles for each joint are compared to the joint limits. If any exceeds its limit, then the joint is locked in its current position. The limits of the other joints are adjusted according to Tables 6.1 - 6.4. For example, when joint 2 reaches its positive limit, only the positive limits of joints 1 and 3 are altered. The negative limits of the three functioning joints are not affected. A new plan is mapped for the robot that only requires movement of the joints that have not reached their limits. These new positions are compared to their limits. If necessary, other joints are locked. This recursive process continues until a plan is mapped that does not require movement of any joint beyond its limit. This plan is passed to the controller. In the next iteration, the joints that were locked because they were near their motion limits are unlocked before a new path is planned. Then the planner constructs a new path for the robot, beginning the process again.
The equations in Tables 6.1 - 6.4 were derived based on the specific case of a limit of 120 degrees in each direction for each joint. Currently, if a joint is made immobile for some reason other than reaching its motion limit, the limits of the other joints are not adjusted. The general equations that allow joints to lock at angles other than their limits are more complicated, but allow more flexibility. The equations are more complicated because the movement of link $i$ is the cumulative result of the movements of the previous links. For instance, when joint 3 locks, the movement of joint 2, which was already dependent upon joint 1, becomes dependent upon joint 3 as well. The general equations will be incorporated to add flexibility and to accommodate motor failure simulation (see Chapter 8).

<table>
<thead>
<tr>
<th>$\theta_0$: at positive limit</th>
<th>at negative limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg. limit</td>
<td>pos. limit</td>
</tr>
<tr>
<td>$\theta_1$</td>
<td>not affected (NA)</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>NA</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 6.1 Limit Constraints for Locked $\theta_0$
\[ A = B \]
\[ a = \frac{\theta_1}{2} \]
\[ y = -180 - a \]

**Figure 6.3** \( \theta_2 \) Limit Change for Locked \( \theta_1 \)

<table>
<thead>
<tr>
<th>( \theta_1 ): at positive limit</th>
<th>at negative limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg. limit</td>
<td>pos. limit</td>
</tr>
<tr>
<td>( \theta_2 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_3 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_0 )</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Table 6.2** Limit Constraints for Locked \( \theta_1 \)
\[ A = B \]
\[ x = \theta_2/2 \]
\[ y = -180 - x \]

**Figure 6.4** \( \theta_1 \) Limit Change for Locked \( \theta_2 \)

<table>
<thead>
<tr>
<th>( \theta_2 ): at positive limit</th>
<th>at negative limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>neg. limit</td>
</tr>
<tr>
<td>( \theta_3 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_0 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_1 )</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Table 6.3** Limit Constraints for Locked \( \theta_2 \)
\[ B = C \]
\[ b = \theta_2/2 \]
\[ y = -180 - b \]

Figure 6.5  \( \theta_3 \) Limit Change for Locked \( \theta_2 \)
\[ C = D \]
\[ d = \theta_3/2 \]
\[ y = -180 - d \]

**Figure 6.6** \( \theta_2 \) Limit Change for Locked \( \theta_3 \)

<table>
<thead>
<tr>
<th>( \theta_3 ): at positive limit</th>
<th>at negative limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg. limit</td>
<td>pos. limit</td>
</tr>
<tr>
<td>( \theta_0 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_1 )</td>
<td>NA</td>
</tr>
<tr>
<td>( \theta_2 )</td>
<td>( &lt; (+180 - \theta_3/2) )</td>
</tr>
</tbody>
</table>

**Table 6.4** Limit Constraints for Locked \( \theta_3 \)
The simulation examples included in this chapter show how the unlocked joints are used to compensate for a locked joint. Figure 6.7 is a picture of the beginning of a run in which joint 1 will reach its motion limit and lock. The darkened (red) link in Figure 6.8 indicates that joint 1 has locked. In Figure 6.9, it is clear that joint 0 is moving more because joint 1 cannot move. Finally, the robot arm reaches the target, in Figure 6.10, with joint 1 still locked.

**Figure 6.7** Model Display: Beginning of Locked Joint Demonstration
Figure 6.8  Joint 1 Reaches Limit
Figure 6.9  Joint 0 Compensates
Figure 6.10  End of Locked Joint Demonstration
Chapter 7

Evaluation

7.1 Performance Results

Our controller architecture was simulated on a Sequent Symmetry S81. Our model utilizes five of the twenty available processors under a moderate multiprogrammed workload. One processor is used during initialization, and to perform robot simulation. Four additional processors are the per-joint controllers. A primary focus of our work has been the relative performance of various fault-tolerant robot controllers to controllers without fault tolerance. To perform this analysis, we timed the solution of the robot dynamics for the four versions of control code described in Chapters 4 and 5. All four controllers solve the Lagrangian equations. Although use of the Newton-Euler equations could result in faster response times, we chose to evaluate the performance of the controllers using a Lagrangian dynamics formulation due to its closed-form structure. We would expect a significant control speedup using recursive Newton-Euler dynamics, which have been shown to be inherently faster (by a factor of 3) than recursive Lagrangian formulations [16]. However, reconfiguration of the system would be more difficult without the closed-form nature of Lagrangian
Dynamics. Since we are interested in relative performance, the Lagrangian equations are better suited to our purpose.

Table 7.1 shows that the parallel version of the controller without fault tolerance is about 153% faster than the serial version. The fault-tolerant controller, HFT, with no processor faults is 34% slower than the serial version. These results exhibit the trade-off between high performance and high fault tolerance. The controller with less redundant data, LFT, performs 49% better than the serial version when there are no failed processors. This controller is 99% faster than HFT.

When one or more processors fails, in a round-robin configuration, a neighboring working processor of the failed processor assumes control of the joint corresponding to the failed processor. When one processor is calculating multiple torque vector elements, the system response time is slower partly because those calculations are done serially. In the results, the runs with processor failures have permanent failures of one and two processors. The failures begin in the second iteration of the test run, and end in the final iteration. Execution times for runs with intermittent failures are faster. Since in HFT, each processor has calculated its own version of the entire inertia matrix and the centrifugal and Coriolis force vector, the only additional calculation that a processor must perform when taking the place of a failed processor is the calculation of an additional torque vector element. This slight load increase avoids the need for any load balancing code to distribute the work more evenly. In Table 7.1, the time required to execute the dynamics for the cases of zero, one, and two processor failures
<table>
<thead>
<tr>
<th>Controller</th>
<th>No. Failed Processors</th>
<th>Relative Execution Time</th>
<th>16MHz 80386</th>
<th>40MHz SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td>0.165</td>
</tr>
<tr>
<td>Parallel w/o Fault Tol.</td>
<td></td>
<td>0.395</td>
<td>* 0.065</td>
<td></td>
</tr>
<tr>
<td>Parallel w/ High</td>
<td>0</td>
<td>1.337</td>
<td>* 0.221</td>
<td></td>
</tr>
<tr>
<td>Fault Tol.</td>
<td>1</td>
<td>1.467</td>
<td>* 0.242</td>
<td></td>
</tr>
<tr>
<td>Parallel w/ Lower</td>
<td>2</td>
<td>1.536</td>
<td>* 0.253</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.670</td>
<td>* 0.111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.832</td>
<td>* 0.137</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.170</td>
<td>* 0.193</td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.1 Relative Performance of Controllers (* calculated numbers)**

is recorded. The slight increases in execution time indicate that there is gradual degradation in performance in the presence of the failure of half of the available processors for HFT. The system performance when all four processors are available is 9.7% better than when one processor has failed, and 14.8% better than when the maximum of two processors have failed.

In LFT, each processor does not have its own version of the inertia matrix and the centrifugal and Coriolis force vector. Here when a processor is removed from the working set, the processor that assumes its functions must calculate the rows of the matrix and vector that corresponded to the failed processor. Therefore, the performance degradation is much higher than for HFT. From four processors to two, we see a 74.6% performance degradation. However, LFT is still faster with one failure than the serial controller. Thus, using a shared memory multiprocessor architecture, we have shown that a high performance fault-tolerant robot controller is feasible.
When a processor fails, there is some amount of slow-down, but not loss of control. We anticipate similar speedup and degradation results for the case of general robot dynamics (our robot arms in the lab have eight joints, requiring an eight processor architecture in ultimate implementation).

Speedup can be achieved by adding hardware (a multiprocessor architecture), or replacing existing hardware with faster hardware, or both. We show that we can achieve speedup by using more processors to do the same work. To demonstrate the speedup made possible by running on a faster processor, we created serial code similar to the serial robot control code run on the Symmetry to run on a uniprocessor with a faster processor than the Symmetry’s. The processor used in this system is a 40MHz SPARC processor. Since we could not time a small section of code on the Sun workstation, as we did on the Symmetry, we ran a program that did not include the initialization and simulation work. This way, we could interpret the time required to execute the entire program as the time required to solve the dynamics. The results shown in Table 7.1 indicate that the execution time of the serial code is an order of magnitude smaller on the SPARC processor architecture than on the Intel 386 processor architecture of the Symmetry. The relative execution times for the parallel versions of our controller were calculated for comparison, since no multiprocessor SPARC was available at the time. We expect that a multiprocessor architecture using SPARC processors would exhibit the same relative speedup as was produced on the Symmetry multiprocessor system.
7.2 Analysis of Architecture

The architectural decisions made for our system have been discussed in Chapters 4 and 5. The areas that could be handled more efficiently are discussed below. Some of the areas of system behavior that would be affected by different design decisions are also discussed below.

7.2.1 Inefficiencies of Current Architecture

The current task decomposition provides coarse grained parallelism. With this, the response time of LFT is a little over half that of the serial controller when there are no processor failures. However, some of the tasks could be further divided into subtasks and executed in parallel. With further task decomposition, the parallel controller without fault tolerance (PC) and LFT could have better response times. More synchronization may be necessary when more tasks are done in parallel. Thus, there is also a point at which more parallelism could degrade performance.

When the processors compare redundant data and vote to determine the working set, it is assumed that faulty processors will not produce the same erroneous result. Therefore, there is no provision for the deadlock situation described in Chapter 5. The controller halts when this situation occurs. Other ways of handling this situation are described in Chapter 8.

The round-robin reconfiguration scheme described in Chapter 5 ensures that each element of the torque vector is calculated only once. The unfairness of this scheme
is exhibited in the example in Figure 5.2. One processor has three times its normal workload, while the other is doing no extra work. A more evenly distributed workload would be completed faster. This becomes more important as the number of processors increases. A more complex reconfiguration scheme in which the remaining functioning processors are equally loaded would reduce performance degradation for the controller.

Each processor in the working set controls one joint. When a joint fails (motor failure or mechanical failure), the processor designated to manage its movement is no longer needed for that purpose. Processor reconfiguration at this point may be desirable to reduce the workload of another processor or to speed up overall execution. However, if the processor is controlling more than one joint, when one of the joints fails it should not be given more work. Addition of processor reconfiguration upon joint failure would marginally improve the overall performance. The amount of performance improvement is dependent upon the type of processor failures that occur most frequently. This issue will be addressed further in future work.

7.2.2 Architectural Alternatives

In many theoretical discussions of fault tolerance, triple modular redundancy (TMR) is used to provide various forms of fault tolerance. A TMR architecture offers the same level of redundancy as our design for a robot with three degrees of freedom. However,
since most robots have more than three degrees of freedom, our architecture would provide a higher level of processor fault tolerance.

Through the use of hot and/or cold spares, it is possible to maintain a constant working set size. We chose not to replace a processor that is removed from the working set with a spare because the performance degradation caused by the loss of one or two processors did not warrant the cost of having spare processors. However, it is possible that in a one-processor-per-joint architecture with a large number of joints, the decrease in performance after a number of failures would warrant inclusion of spare processors. In an architecture with ten processors, for example, it may be acceptable to not replace faulty processors with spares until there are only five processors remaining in the working set. At this point, system performance may degrade beyond acceptable levels, requiring the addition of spare processors. When faulty processors are replaced by spares would be determined by the level of performance degradation caused by processors being removed.

Maintaining a constant working set size would have the added expense of more processors. Hot spares execute the exact same code as the processors they may have to replace. Therefore, the number of hot spares required is dependent upon the program structure. When each processor in the working set is performing a different task in parallel, there must be multiple hot spares for each task at that task level. When each member of the working set is performing the same task in parallel (for compare/vote), there can be fewer hot spares at that task level because each spare
may replace any of the working set members. Cold spares do not execute any code until they are brought into the working set. Therefore, they can replace any processor that is found faulty. In this case, the level of fault tolerance provided is determined by the total number of spare processors used.

Spinlocks are used for synchronization in this system. They are satisfactory because the locks are expected to be held for a short time [31]. In a spinlock synchronization scheme, each processor continually checks a shared variable to determine whether an operation has completed. All processors (except the processor that set the variable) spin until the variable changes value. This method generates much bus traffic, and therefore does not scale well to many processors [15]. However, for the number of processors that are considered for our robot control architecture (ten processors, at most), this is not a problem. If spinlocks are not desirable and there is no hardware support for test-and-set, semaphore, or monitor type instructions, hardware synchronization could be used [31]. Hardware synchronization is also faster than software synchronization.

The comparison and voting used for detection of a processor failure are executed in software. By using logic gates, these functions could be implemented in hardware for faster execution. However, a software implementation is less expensive.

Approximate agreement can mask generic faults in the software or hardware [18]. The same information is obtained by different means. If one of the methods of obtaining information is faulty, the information gained from that source is masked
after voting on results. Use of the approximate agreement “rule” complicates fault
detection because the information being compared cannot be expected to be in exact
agreement. However, if the probability of a “generic” fault is high, then approximate
agreement is the best choice.

We chose to implement a shared memory architecture instead of private memory
associated with each processor. However, if private memory is used, a message-
passing scheme would have to be used to get the necessary information, such as
joint positions, to all processors. The necessity of this interprocessor communication
would motivate a different architecture. Recent advances in VLSI have improved
hardware interprocessor communication. A processor, such as the TMS320C40 – a
programmable digital signal processor, has dedicated interprocessor communication
ports. These ports allow communication of data between processors in parallel with
processor work. This allows fairly simple parallelization of calculations. Future work
may include porting of the fault tolerance system to this architecture.
Chapter 8

Future Work

In our future work, we will make some modifications to improve the efficiency of our architecture. We will also continue to study processor failures, detection of these failures, and the effects of various processor configurations on their detection.

Efficient task decomposition is key in providing high-speed performance. Therefore, we will re-examine the task graphs to find tasks that can be decomposed for finer grained parallelism. The added parallelism may require more synchronization, which will reduce the response time. Therefore, we will experiment with varying levels of parallelism to find an optimal task decomposition.

Currently, when the results of redundant data comparisons are deadlocked, the controller stops working. However, when all processors are executing the same code, it is not highly likely two faulty processors will reach the same result. Therefore, in future work, we will recalculate the redundant data and perform the comparison/votes again whenever deadlock occurs. If the votes are still deadlocked after the retry, either the system can be halted, or a working set can be chosen based upon a log of previous failures.
Processor reconfiguration, a necessary component of system recovery, affects the speed performance of the controller architecter. Therefore, the round-robin reconfiguration scheme described in Chapter 5 will be modified to provide a more evenly distributed workload after processor failure.

In order to measure the benefit of reconfiguring processors when a joint motor fails, it is necessary to simulate that failure. As with processor failures, joint failure information will be input by the user. Currently, the range of motion limits are not changed when a joint becomes locked. However, the limits should be adjusted in the same manner as they are when a joint reaches its limit. This functionality will be added for correct joint failure simulation. In addition to this change, the code that checks and adjusts the range of motion limits for the joints will have to be modified some. There is an initial limitation set on the range of motion for each joint, which is set in initialization. Presently, after the limits have been modified and the plan executed, they are reset to this initial value. However, when joint failures are possible, the limits may need to be reset to something other than their initial values. If, for example, joint 2 suffers a failure, the limit for joint 1 is changed. Then, when joint 1 reaches its limit, it is locked, and a new path is planned. Before the plan for the next iteration is developed, the limit for joint 1 is reset to its initial value. However, since joint 2 is still immobile, the limit for joint 1 should be different from the initial value.
To improve the fault tolerance of the entire system, several checking schemes may be used. Periodic writes to and reads from memory check the memory module. Processors can execute self-testing procedures to check for failures. A log of processor failures would be useful if two processors were allowed to control the robot. Upon disagreement between the two processors, the processor with the most failures logged would be chosen to be the faulty processor.

We intend to explore the nature of processor failure more carefully. The characteristics of processor failures affect the detection and recovery (reconfiguration) methods, and therefore the level of fault tolerance required. In an environment where transient or intermittent errors may occur most often, more frequent data comparisons may be needed for processor failure detection. Rather than one compare/vote per iteration, the processors may need to compare data multiple times. However, if permanent failures are more common, then it may be wasteful to have a failed processor participate in data calculations in each iteration. It may be more efficient to leave the processor out of the working set for a fixed number of iterations. This would reduce the amount of time spent executing compare and vote code, since fewer processors would be participating at times.

It is not always necessary to have the highest degree of fault tolerance. The criteria that should be used for determining when the level of fault tolerance needs to be high (or low) is one research area that we intend to explore. The amount of redundant data and the frequency of their comparisons are part of what defines the
level of fault tolerance provided. Our work should provide a means of relating this information. The level of fault tolerance is also affected by the processor configuration chosen. We will continue to research processor redundancy and its effects. We will perform simulations with various cold and hot spare configurations, and experiment to determine the effects of adding spares at different levels. For instance, no spares may be added until the working set size drops below its voting threshold \( \left( \frac{n}{2} + 1 \right) \) processors. We will determine the cost (performance and expense) of using cold and hot spares.

A user interface that allows run-time modifications to the control will be crucial to user acceptance. The simulation facility will be enhanced to provide this functionality. Simulations will be made more interactive by allowing the user to lock joints and fail processors during a run. This will allow the user to dynamically verify the path planning and fault tolerance.
Chapter 9

Conclusion

Robots are used to perform numerous tasks, from simple pre-programmed tasks to more difficult tasks requiring dynamic path planning and high-speed control. Robot control algorithms are becoming more complex as their tasks increase in difficulty. Uniprocessor architectures have great difficulty executing these algorithms at the speeds desired, or even required for some tasks. Additionally, as robots are used to take the place of humans in various environments, fault tolerance becomes a design imperative to increase safety and reliability. This fault tolerance is provided by the use of redundant modules, such that failure of one module does not cause system failure. An architecture with a single control processor, therefore, does not allow tolerance of processor faults.

Use of multiple processors in a robot control architecture addresses both of these problems. Decomposing the tasks of the controller into tasks that can be executed in parallel reduces the response time of the controller. Having multiple processors available also means that failure of one processor does not cause failure of the robot. However, the incorporation of processor fault tolerance requires the existence of redundant data for comparisons. Calculation of this data and comparisons between
the processors add to the execution time, and thus decrease the response time of the parallel controller. The decrease in speed performance caused by the addition of processor fault tolerance could possibly be masked with the addition of more processors and further decomposition of tasks into parallel subtasks. Thus, there is an obvious trade-off between high-speed performance, fault tolerance, and system cost (number of processors).

In this thesis, a parallel fault-tolerant robot controller was developed using a shared memory multiprocessor architecture. Several versions of the robot controller were developed for comparison. A robot simulation was also developed for control observation.

Comparison of a serial version of the controller (SC) and a parallel version without fault tolerance (PC) show the speedup possible with the coarse grained parallelism incorporated. The performance degradation due to the addition of processor fault tolerance is made evident by comparison of SC and PC with their fault-tolerant versions, (HFT) and (LFT), respectively. Comparison of HFT with LFT shows how varying the amount of redundant data affects performance. The results of this work exhibit the trade-off between speed performance and processor fault tolerance. Since a fixed number of processors were used, the cost factor is not included.

This work includes dynamic modification to joint motion limits and the motion plan. Limitations were added to the range of motion of the joints for more accurate modeling of actual hardware. When a joint reaches its set limit, it is temporarily
locked and new plan is mapped for the robot. These modifications are useful for obstacle avoidance. They will also allow the simulation of joint failures that is needed for analysis of processor reconfiguration when joint failure occurs.
Bibliography


