Parallel Fault-Tolerant Robot Control

D.L. Hamilton, J.K. Bennett, and I.D. Walker

Dept. of Electrical and Computer Engineering
Rice University, Houston, TX 77251

ABSTRACT

Most robot controllers today employ a single processor architecture. As robot control requirements become more complex, these serial controllers have difficulty providing the desired response time. 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 the level of fault tolerance provided. This paper describes a shared memory multiprocessor robot controller that is capable of providing high performance and processor fault tolerance. We evaluate the performance of this controller, and demonstrate how performance and processor fault tolerance can be balanced in a cost-effective manner.

1. INTRODUCTION

Traditionally, robots have been used to perform simple repetitive tasks in static environments. For instance, a robot would be used to move known objects from one point to another, along a straight line, with no obstacles. The starting point and end point would be constant where static, preprogrammed controllers could be used successfully. As the robot's operating environment and the task planning required of the robot have become more complex, control requirements have increased significantly. For robot manipulators, if the desired trajectory becomes more complex, or the dynamic properties of the objects are not precisely known, then the control force for the objects must change dynamically. If obstacles are encountered dynamically, then the controller must adapt to changes in the environment in real-time. Most robot controllers used today are uniprocessor systems. With expanded functionality of robots (particularly high speed and close control operations), it has become increasingly difficult for these serial controllers to produce real-time responses [6, 12].

In addition to the need for improved performance, robot reliability has assumed greater importance, especially in critical tasks such as space exploration and operation in environments contaminated with hazardous materials. The uniprocessor architectures of today have the added disadvantage of being intolerant of processor failures. Failure of the sole control processor may remain undetected until the robot causes damage to itself or its environment. If the processor failure is somehow detected before causing system failure, the robot cannot continue working because there is no other processor available for control. In this paper, we introduce and evaluate a parallel robot controller that exhibits tolerance of processor failures and demonstrate how the performance and level of fault tolerance of this controller can be balanced in a cost-effective manner.

2. MULTIPROCESSOR ROBOT CONTROLLERS

Development of multiprocessor architectures has opened new avenues for the real-time control problem. Advances in robot control theory have produced algorithms that are difficult to implement on a uniprocessor system in real time

[6]. Parallel architectures have therefore been proposed to fulfill this need [6, 12]. Robotic kinematics and dynamics calculations are well suited for parallelism due to the numerous matrix and vector manipulations involved. The use of multiple processors can speed up calculations by having each processor operate on a different row or column of a matrix, for example. Executing such operations in parallel allows faster updates of the robot control solution than that provided by a serial controller, permitting faster or finer control of motion. Numerous parallel algorithms for solving the kinematics and dynamics have been written using the Lagrangian and the Newton-Euler equations [3, 4, 22, 23, 25]. These algorithms exploit parallelism at various levels. Due to the interdependence of data in a robot control system, efficient use of parallelism is also affected by the memory architecture.

A parallel computer architecture has either shared memory with a common address space, or distributed memory with a disjoint address space [1]. 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. This is very much like the familiar sequential programming environment. In a distributed memory environment, each processor only has direct access to 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 these two processors. This explicit communication is usually more difficult for the programmer to manage than the implicit communication of a shared memory environment.

Most shared memory multiprocessors do not scale well to large numbers of processors, due primarily to bus contention and memory latency. Since typical robots would have fewer than ten joints and on the order of one processor per joint, scalability is not an important consideration. Hence, the contention and latency problems often associated with large shared memory multiprocessors do not appreciably affect performance in common robot control applications. For this reason, we have chosen to utilize a shared memory architecture in order to gain the attendant programming advantages.

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 this reliability. Possible points of failure are duplicated so that the work of a failed module may continue, and the operation is not halted. The use of multiple processors provides fault tolerance via processor redundancy. The failure of one processor, or a few processors, does not result in system failure. However, incorporating this fault tolerance, which requires added 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 [10]. 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 be serviced once it is exposed to the hazard. In these instances, robot reliability is paramount, and extra care to ensure fault tolerance is necessary. If the robotic system cannot be shut down for repair or replacement, then extra processors are needed to continue the work of the failed processor. Processor failures can be detected by comparing redundant data. This data is produced by executing some tasks on every processor in parallel. The results produced by properly functioning processors will be identical. To ensure that the correct data is being compared, all processors must synchronize just before the 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 in such a case 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. The most cost-effective performance improvement would

[1] Some systems, such as distributed shared memory, employ hybrid memory architectures.

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.

3. RELATED WORK

Speedup advantages of a multiprocessor architecture have been examined in several robot control architectures [8, 24]. The performance improvement is achieved using parallel algorithms [16] in various parts of the control code, such as the forward and inverse dynamics [**?**, 4, 13, 22, 23].

Most of the literature on parallel robot controllers that we have studied describes architectures that utilize multiple processors only to achieve speedup. However, many of the uses of robots today necessitate a fault-tolerant architecture. In a chapter of Fleming's book on transputers [5], 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 [5].

Bejczy and Szakaly [2] 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.

A reconfigurable, dual network, single-instruction-stream multiple-data-stream (SIMD) machine was developed by Lin and Lee [14] for robot kinematics and dynamics solutions. The processing elements are essentially arithmetic logic units, and operate in lock-step under the control of the central control unit. In this architecture, operations are pipelined through a three-stage switch network. Fault tolerance is provided by an additional switch stage at the input and the output of the switch network, and by the inclusion of spare processing elements. Thus, network faults and processing element faults can be overcome by this architecture.

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. [11]. 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 [11].

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 done by a group at the MIT Artificial Intelligence Lab [17]. In the MAFT architecture, 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 [19] 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 that 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 [19].

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.

4. ROBOT CONTROL

The robot dynamic equations may be expressed by the Lagrangian formulation or by the Newton-Euler formulation [21]. The Lagrangian formulation for the dynamics equations is used for our analysis. The closed-form nature of this form is better suited to our objective than the recursive Newton-Euler formulation. In our fault tolerance work, we roll back to a previous correct state, and modify the control strategy and computational paths in response to detection of a fault. This would be more difficult if a recursive formulation was used [24]. The dynamic model takes the following form:

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

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 [21].

We use the classical "computed-torque" robot controller [21] (with a PD-compensator) for our work. The PD-compensator minimizes the effects of steady state and tracking errors in motion control. The viscous friction component is neglected in our implementation to simplify the model. Hence, our control torques are formed by [21].

$$\bar{\tau} = [M(\bar{\theta})]\{\ddot{\bar{\theta}}_d + [K_D](\dot{\bar{\theta}}_d - \dot{\bar{\theta}}) + [K_P](\bar{\theta}_d - \bar{\theta})\} + \bar{N}(\bar{\theta}, \dot{\bar{\theta}}) + \bar{G} \tag{2}$$

for $\theta \in \Re^n$, $M \in \Re^{n \times n}$, $K_D \in \Re^{n \times n}$, $K_P \in \Re^{n \times n}$, $N \in \Re^n$, and $G \in \Re^n$, and where $\bar{\theta_d}$ is the desired trajectory. The control equation (2) is very complex for general manipulators, and has proven difficult to compute in real time using uniprocessor architectures [6, 12]. Our architecture computes (2) in real time using a shared memory multiprocessor.

5. OUR APPROACH

We have developed a robot control architecture that provides improved performance and increased fault tolerance through the use of a shared memory multiprocessor. For comparison, 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 was developed as a benchmark for analysis of the three parallel controllers developed. The multiprocessor used is a Sequent Symmetry S81 [15]. For visual analysis of the robot control solution, simulations of two robots – a planar four-link robot [7] and a six joint PUMA [18, 20] – were also developed. A Silicon Graphics Personal Iris workstation using the Trick/MAGIK graphics simulator, which was developed at NASA/Johnson Space Center [1], is used to simulate and display the robot arms. 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 appropriate arithmetic model required for simulation. Much of the code needed to generate and run a robot 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 first parallel controller that we developed had no tolerance of processor failures. This controller provides maximum speedup (given the algorithm used) using the minimum number of processors required to execute the control algorithm in parallel. 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. For this PUMA controller, $n = 10$.

Following this controller, two controllers with varying levels of fault tolerance were developed and analyzed. The first of these is a fault-tolerant version of the serial controller, exhibiting the greatest degree of software redundancy. This controller utilizes six processors operating in parallel. All of the processors calculate each element of their own copies of the inertia matrix and the centrifugal and Coriolis torque vector. Each processor compares elements 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.

A second fault-tolerant controller was developed that calculated less redundant data than the first. This controller is a fault-tolerant version of the original parallel controller. It also uses ten processors in its calculations. In this

case, 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 inertia matrix and one element of the centrifugal and Coriolis torque 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.

To perform our analysis, we timed the solution of the robot dynamics for the four versions of control code. 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 [9]. 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.

6. RESULTS

Table 1 shows that the parallel version of the controller without fault tolerance is about 421% faster than the serial version. However, the highly fault-tolerant controller, with no processor faults is 16.7% slower than the serial version. These results exhibit the trade-off between high performance and high fault tolerance. The controller with less redundant data performs 248% better than the serial version when there are no failed processors. 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. Thus, when one processor is calculating multiple torque vector elements, the system response time is slower partly because those calculations must be 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 run, and end in the final iteration. Execution times for runs with intermittent failures are faster. Since in the highly fault-tolerant controller, 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 1, the time required to execute the dynamics for the cases of zero, one, and two processor failures 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 this controller. The system performance when all processors are available is 1.66% better than when one processor has failed.

Controller	No. Failed Processors	Relative Execution Time 16MHz 80386
Serial		1.0
Parallel w/o Fault Tol.		0.192
Parallel w/ High Fault Tol.	0	1.167
	1	1.187
Parallel w/ Lower Fault Tol.	0	0.287
	1	0.290
	2	0.325

Table 1 Relative Performance of Controllers

In the lower level fault-tolerant controller, 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 higher than for the other fault-tolerant controller. From ten processors to eight, there is a 13.2% performance degradation. However, this controller is still faster with two failures 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 for the four-link arm 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 showed 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. We expect that a multiprocessor architecture using SPARC processors would exhibit the same relative speedup as was produced on the Symmetry multiprocessor system.

7. FUTURE WORK

Efficient task decomposition is paramount in providing high-speed performance. Our current implementation is designed to investigate the tradeoffs encountered due to processor fault tolerance, and takes advantage of only simple parallelism. We intend to continue to develop control algorithms that exhibit finer grained parallelism. This added parallelism may require more synchronization, reducing the response time. Thus, we will experiment with varying levels of parallelism in an effort to find an optimal task decomposition.

Processor reconfiguration, a necessary component of system recovery, affects the performance of the controller. Our round-robin reconfiguration scheme ensures that each element of the torque vector is calculated only once. This scheme becomes unfair, resulting in an unbalanced workload, when a number of processors in a row fail. In this case, the working processor at the end of the series will be performing the work for all of the malfunctioning processors. This issue assumes greater importance as the number of processors increases. A more robust processor reconfiguration scheme, in which the remaining functioning processors are equally loaded, will increase overall controller performance.

We added range-of-motion limitations for each joint in order to allow our robot simulator to better approximate actual hardware. These limits also make joint failure compensation and obstacle avoidance possible. 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, and the limits of the other joints are adjusted. 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 in turn to their limits, and, if necessary, additional joints are locked. This recursive process continues until a plan is mapped that does not require movement of any joint beyond its limit. In our next iteration of the controller, the joints that are locked because they are near their motion limits will be unlocked before a new path is planned, resulting in better path planning.

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 frequently occur, 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 to reduce the amount of erroneous data to be recalculated. 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.

It is not always necessary to have the highest degree of fault tolerance. We intend to explore the criteria that should be used for determining when the level of fault tolerance needs to be high. The amount of redundant data, and the frequency of comparison, 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 need be added until the working set size drops below its voting threshold ($\frac{n}{2} + 1$ processors). We will explore the effect on cost and performance using various cold and hot sparing schemes.

8. CONCLUSION

Robots are used to perform numerous tasks, from simple preprogrammed 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, obviously 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 improves 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 work, a parallel fault-tolerant robot controller was developed using a shared memory multiprocessor architecture. Several versions of the robot controller were developed and compared. A robot simulation was also developed for control observation.

Comparison of a serial version of the controller and a parallel version without fault tolerance showed the speedup possible with the coarse grained parallelism currently employed. The performance degradation due to the addition of processor fault tolerance was demonstrated by comparison of these controllers with their fault-tolerant versions. Comparison of the more fault-tolerant controller with the lower level fault-tolerant controller showed how varying the amount of redundant data affects performance. The results of our work demonstrate the trade-off between speed performance and processor fault tolerance.

8. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under Grants MSS-9024391 and CDA-

8619393, by the Department of Energy under Sandia National Laboratory Contract number 18-4379A, and by a NASA Graduate Fellowship.

9. REFERENCES

References

[1] R.W. Bailey and L.J. Quiocho. *Trick Simulation Environment: Simulation and Math Model Developer's Guide.* LinCom Corporation and NASA/Johnson Space Center, Beta-release edition, 1991.

[2] A.K. Bejczy and Z. Szakaly. Universal Computer Control System (UCCS) for Space Telerobots. In *Proceedings of the of 1987 IEEE International Conference on Robotics and Automation*, pages 318–324, Raleigh, NC, 1987.

[3] C.L. Chen, C.S.G. Lee, and E.S.H. Hou. Efficient Scheduling Algorithms for Robot Inverse Dynamics Computation on a Multiprocessor System. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pages 1146–1151, Philadelphia, PA, 1988.

[4] A. Fijany and A.K. Bejczy. Parallel Algorithms and Architecture for Computation of Manipulator Forward Dynamics. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 1156–1162, Sacramento, CA, 1991.

[5] P.J. Fleming, editor. *Parallel Processing in Control: the Transputer and Other Architectures*, volume 38 of *IEE Control Engineering Series*. Peter Peregrinus Ltd., London, 1988.

[6] J.H. Graham. Special Computer Architectures for Robotics: Tutorial and Survey. *IEEE Transactions on Robotics and Automation*, 5(5):543–554, October 1989.

[7] D.L. Hamilton, J.K. Bennett, and I.D. Walker. Simulation of a Reliable Parallel Robot Controller. In *1992 International Simulation Technology Conference*, Clear Lake, TX, November 1992. To appear.

[8] J.Y. Han and C.Y. Wang. Modeling and Performance Evaluation of Multiprocessor Systems for Real-Time Nonlinear Robot Control. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1016–1021, Scottsdale, AZ, 1989.

[9] J.M. Hollerbach. A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(11):730–736, November 1980.

[10] H. Kasahara and S. Narita. Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System. *IEEE Journal of Robotics and Automation*, RA-1(2):104–113, June 1985.

[11] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai. The MAFT Architecture for Distributed Fault Tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[12] C.S.G. Lee. Introduction to Special Issue on Robot Manipulators: Algorithms and Architectures. *IEEE Transactions on Robotics and Automation*, 5(5):541–542, October 1989.

[13] C.S.G. Lee and C.L. Chen. Efficient Mapping Algorithms for Scheduling Robot Inverse Dynamics Computation on a Multiprocessor System. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(3):582–595, May/June 1990.

[14] C. Lin and C.S.G. Lee. Fault-Tolerant Reconfigurable Architecture for Robot Kinematics and Dynamics Computations. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):983–999, September/October 1991.

[15] T. Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, St. Charles, IL, 1988.

[16] F. Naghdy, C.K. Wai, and G. Naghdy. Multiprocessing Control of Robotic Systems. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pages 975–977, Philadelphia, PA, 1988.

[17] S. Narasimhan, D.M. Siegel, and J.M. Hollerbach. Condor: A Revised Architecture for Controlling the Utah-MIT Hand. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pages 446–449, Philadelphia, PA, 1988.

[18] C.P. Neuman and J.J. Murray. The Complete Dynamic Model and Customized Algorithms of the Puma Robot. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(4):635–644, July/August 1987.

[19] F. Ozguner and M.L. Kao. A Reconfigurable Multiprocessor Architecture for Reliable Control of Robotic Systems. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 802–806, St. Louis, MO, 1985.

[20] R.P. Paul, M. Rong, and H. Zhang. The dynamics of the puma manipulator. In *1983 American Control Conference*, pages 491–496, San Francisco, CA, 1983.

[21] M.W. Spong and M. Vidyasagar. *Robot Dynamics and Control.* Wiley, New York, 1989.

[22] R.W. Toogood. Efficient Robot Inverse and Direct Dynamics Algorithms Using Micro-computer Based Symbolic Generation. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1827–1832, Scottsdale, AZ, 1989.

[23] W. Wang, K. Chen, Y. Lai, and C. Liu. Implementation of a Multiprocessor System for Real-Time Inverse Dynamics Computation. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1174–1179, Scottsdale, AZ, 1989.

[24] A.M.S. Zalzala and A.S. Morris. A Distributed Pipelined Architecture of Robot Dynamics with VLSI Implementation. *International Journal of Robotics and Automation*, 6(3):117–128, 1991.

[25] H. Zhang and R.P. Paul. A Parallel Inverse Kinematics Solution for Robot Manipulators Based on Multiprocessing and Linear Extrapolation. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 468–474, Cincinnati, OH, 1990.