

SIMULATION OF A RELIABLE PARALLEL ROBOT CONTROLLER

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

Department of Electrical and Computer Engineering
Rice University
Houston, TX 77251-1892

ABSTRACT

Parallel solutions to the robot control problem are an attractive alternative to single-processor systems as the need for fine motion robot control increases. Since multiple processors provide inherent redundancy, parallel solutions to the robot control problem also afford the opportunity to provide a degree of tolerance to both mechanical and control failure. In this paper we describe a new robot control architecture that provides improved performance and processor fault tolerance. Using simulation, we then evaluate the performance of a shared memory multiprocessor robot control architecture in the presence of processor failures.

INTRODUCTION

Advances in robotics have resulted in increasingly demanding tasks proposed for robot manipulators. These tasks have placed greater demands on the robot controller. More intricate movements require shorter sampling intervals between control commands. Therefore, the response time of the controller must decrease. Also, as robots are used in more dynamic environments, better sensor feedback must be used for navigation. This added information must be processed without increasing response time. Today's uniprocessor robot control architectures have difficulty providing real-time control of these systems. Multiprocessor controllers can provide faster response by executing the control algorithm in parallel (0, 0). Faster execution allows the finer control desired for the more demanding tasks.

Robots are also being entrusted with critical tasks – tasks that cannot be performed by humans, or tasks that cannot be interrupted due to system failures. In order to improve the reliability of these systems, modular redundancy is often incorporated. The use of multiprocessor architectures makes processor fault tolerance feasible. The cost of adding pro-

cessor fault tolerance is degraded response time, but this degradation can be mitigated by using additional processors. The additional software and hardware required for detection of and recovery from processor failures increases the amount of time it takes to provide a usable dynamics solution to the robot. Thus there is a trade-off between performance and fault tolerance.

The level of fault tolerance of a system may vary, depending on what factors are most important. One can have the highest possible levels of speedup and fault tolerance by using the necessary number of processor groups. For instance, if the optimal task decomposition requires the use of four processors working in parallel and the fault tolerance requirement is to have three processors performing each task, then four groups of three processors will provide the highest possible levels of speedup and processor fault tolerance. Some performance degradation will still be experienced due to the additional code for fault tolerance, unless this code is executed in parallel with the necessary control calculations. The performance/fault tolerance relationship is also affected by the number of processors available (i.e. the cost limitations for the system).

We have developed a robot control architecture that provides processor fault tolerance, as well as performance improvement, through the use of a shared memory multiprocessor. We have developed several parallel versions of the robot controller, and a serial benchmark for performance analysis. Through comparison of these robot controllers with the serial controller and with each other, we have demonstrated that improved performance and processor fault tolerance can be achieved in a single architecture. In this paper, we describe and evaluate this architecture.

CONTROLLER

Robot Control Strategy

The work presented in this section is described in greater detail in (0). Our architecture is designed to implement dynamics-based control for robot manipulators. The robot dynamic equations may be expressed by the Lagrangian formulation or by the Newton-Euler formulation (0). 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. The dynamic model takes the following form:

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

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 (0).

We use the classical “computed-torque” robot controller (0) (with a PD-compensator) for our work. The PD-compensator minimizes the effects of steady state and tracking errors in motion control. Since the robot model used currently is planar, the gravity vector is orthogonal to the plane of motion of the arm. Thus, there are no gravity torques to consider in our initial implementation. The viscous friction component is also neglected in our implementation, to simplify the model. Hence, our control torques are formed by (0).

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

for $\theta \in \mathfrak{R}^n$, $M \in \mathfrak{R}^{n \times n}$, $K_D \in \mathfrak{R}^{n \times n}$, $K_P \in \mathfrak{R}^{n \times n}$, and $N \in \mathfrak{R}^n$, 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 (0, 0). Our architecture computes (2) in real time using a shared memory multiprocessor.

Architecture

Our robot controller currently is implemented using five of twenty available processors of a Sequent Symmetry S81, a shared memory multiprocessor. One of these processors performs the initialization of various parameters and executes the simulation code. The other four processors make up the parallel controller. Each processor controls one joint/link of a simulated robot, which is described in the next section. The structure of our robot controller is explained in more detail in (0).

We have chosen a shared memory architecture because it eliminates the need for explicit communication between the control processors. Since each processor must have access to the current position of each joint of the robot, an architecture with distributed private memories would require communication among the processors.

The first version of our controller is a serial program. This controller is the baseline for benchmarking the parallel controllers. The matrix and vector manipulations of the serial robot control algorithm were divided among the processors for the first parallel control program. Each processor writes to only one row of a matrix and one element of a vector, thus, providing control information to one joint of the robot. Since this version of the controller does not have processor fault tolerance, a processor failure results in the loss of control of one joint.

Two fault-tolerant versions of the robot controller were also developed. The first of these is a fault-tolerant version of the serial controller. In this case, each processor executes the entire dynamics equation solution, and thus has its own version of the inertia matrix and the centrifugal and Coriolis force vector. The elements of each version of the inertia matrix are compared to all others to detect processor failures.

The second fault-tolerant controller adds processor fault tolerance to the original parallel controller. In this version, only the data that is to be compared is calculated redundantly. All other calculations are done per matrix row and per vector element. Each processor calculates its own version of the diagonal elements of the inertia matrix for comparison. Once the faulty processors have been removed from the working set, only the remaining processors calculate the elements of the inertia matrix and the centrifugal and Coriolis force vector.

The reconfiguration scheme for both fault-tolerant controllers is a round-robin replacement scheme. Each processor checks its neighbor to the right. Functioning processors continue checking right-side neighbors until another functioning processor is encountered. This scheme will tolerate up to $(n - 1)$ processor failures, where n is the number of processors. All fault tolerance is lost once there are fewer than three processors available for data comparisons. The disadvantage of this scheme is that it can cause unbalanced loading of processors. The work of several consecutive failed processors is taken by a single functioning processor. The workload of this functioning processor may be greatly increased, while the loads of the other working processors remain unchanged. For as few as four processors, this reconfiguration scheme is useful because it is simple and efficient enough. However, with more processors in the original configuration, the possibility of unfair loading a single processor increases. A more complex reconfiguration scheme in which the work of the failed processors is more evenly distributed among the working processors would reduce performance degradation for the controller.

Control Program Initialization

In our simulation, the user sets several parameters at the beginning of the program. Some information is also input for processor failure simulation in the control algorithms with processor fault tolerance. The user provides the processor number (0-3), the number of failures for each processor (with a maximum set), and the start and end times of each of these failures. Thus, permanent and intermittent failures can both be simulated. In each iteration of control, the data structures containing processor failure information are checked and the appropriate failure flags set.

The initial end effector position is calculated from the initial joint positions, which are input by the user. Then the desired destination of the end effector is input. This data is used to calculate the intended velocity of the robot, which is used in the inverse kinematics routine of the planner developed by Deo (0).

Limits are set for the range of motion of each joint in initialization. These limits may be changed during the run to simulate reconfiguration following joint failures. This capability would be used for experiments on processor reconfiguration following joint failure.

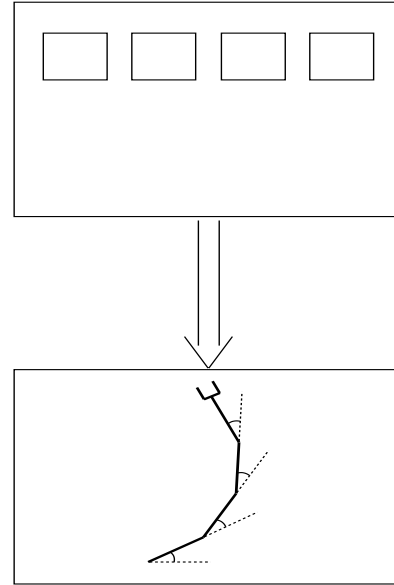


Figure 1: Controller and Robot

Processor Failure Simulation

Processor failures may be manifested in a number of areas. For example, the ability to properly read from or write to memory could be lost, or the arithmetic logic unit (ALU) could malfunction. We simulate failures in the ALU. To do this, one element of the redundant data is changed for those processors that the user has chosen to fail. Processor failures are detected through data comparisons. Each processor calculates its own version of some data, then compares its version with those of the other processors. From the comparisons, each processor produces a comparison vector, then a working set suggestion. The working set suggestions are the votes that determine whether a given processor has failed.

ROBOT SIMULATION

We simulate a four-link planar robot arm using a Silicon Graphics Personal Iris Workstation running the Trick/MAGIK graphics simulator, which was developed at NASA/Johnson Space Center (0). The working environment is shown in Figure 1.

Once the torques that drive the joints are calculated by the controller, we simulate movement of the robot by calculating the new positions of the joints.

Controller	No. Failed Processors	Relative Execution Time	
		16MHz 80386	40MHz SPARC
Serial		1.0	0.165
Parallel w/o Fault Tol.		0.395	* 0.065
Parallel w/ High Fault Tol.	0	1.337	* 0.221
	1	1.467	* 0.242
	2	1.536	* 0.253
Parallel w/ Lower Fault Tol.	0	0.670	* 0.111
	1	0.832	* 0.137
	2	1.170	* 0.193
* calculated			

Table 1: Relative Performance of Controllers

For this calculation, actual joint positions are different from the current angular positions known to the controller. At the beginning of the run, the initial joint positions known to the controller are the same as the actual initial positions. Elements of the inertia matrix and the centrifugal and Coriolis force vector are different from that used in the control torque calculation to simulate parameter modeling and measurement errors. The robot accelerations are calculated from the control torques and the robot inertia matrix and centrifugal and Coriolis force vector, using Equation (1). The equation is:

$$\ddot{\theta} = [M(\bar{\theta})]^{-1}\bar{\tau} - [M(\bar{\theta})]\bar{N}(\bar{\theta}, \dot{\bar{\theta}}) \quad (3)$$

From the accelerations, the robot velocities are calculated, then the positions. These new angles are passed to the graphics routine that displays our simulated robot.

RESULTS

We timed the solution of the robot dynamics equations using the builtin microsecond clock of the Sequent. Timing begins after initialization, at the beginning of the calculation of the inertia matrix. Timing ends after the torques for each joint have been calculated. The data in Table 1 is based on the average execution time of 120 iterations for each version of the controller. The runs with processor failures have permanent failures of one and two processors. The failures begin in the first iteration of the test run, and end in the final iteration.

As shown in the table, the fastest controller is the parallel without processor fault tolerance. It is about 181% faster than the serial version. Its disadvantage is

that failure of only one processor causes loss of control of the corresponding joint. The fault-tolerant version of the serial controller is the slowest of the four versions, due to the addition of the failure detection and reconfiguration code. This controller is 16.3% slower than the serial version. The trade-off between high performance and high fault tolerance is evidenced by these results. The fault-tolerant version of the original parallel controller performs 77.7% better than the serial controller when there are no failed processors, and 106% faster than the other fault-tolerant version.

In the fault-tolerant version of the serial controller, each processor has calculated its own version of the entire inertia matrix and the centrifugal and Coriolis force vector. Therefore, the only additional calculation performed by a working processor taking the place of a failed processor is the calculation of an additional torque vector element. In our implementation, this slight load increase avoids the need for any load balancing code to distribute the work more evenly. The small increases in execution time for the cases of one and two processor failures are an indication of the gradual performance degradation. The system performance when all four processors are available is 4.5% better than when one processor has failed, and 8.8% better than when the maximum of two processors have failed.

In the faster fault-tolerant controller, the processors share one copy of the inertia matrix and the centrifugal and Coriolis force vector. Thus, when a processor is removed from the working set, the processor that assumes its functions must calculate the elements of the matrix and vector that correspond to the failed processor. Therefore, the performance degradation is much higher than for the above fault-tolerant version. There is a 76.1% performance degradation from control by all four processors to control by only two processors. However, this controller is still faster with one failure than the serial controller.

To show what level of speedup is made possible by using faster processors, we created serial code to run on a uniprocessor with a faster processor than the Symmetry's. The processor used 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. The time required to execute the entire program could thus be interpreted as the time required to solve the dynamics. The relative execution times for the parallel versions of our con-

troller were calculated for comparison, since no multiprocessor SPARC was available at the time. We anticipate that a multiprocessor architecture using SPARC processors would exhibit approximately the same relative speedup as was produced on the Symmetry multiprocessor system.

FUTURE WORK

The architecture described in this paper is the initial step in ongoing work in the area of robotic fault tolerance (0). Our future work will include some modifications to improve the efficiency of our architecture, as well as further analysis of architectures with processor fault tolerance.

Efficient task decomposition is necessary to provide high-speed performance. Therefore, analysis of algorithms with possibly finer grained parallelism would be useful. Our current parallel robot control algorithm has incorporated coarse grained parallelism. A finer grained parallel algorithm may speed up the response time of the controller. However, the additional parallelism may require more synchronization, which would increase the execution time. Thus, the optimal task decomposition may not incorporate the greatest amount of parallelism possible.

Processor reconfiguration, a necessary component of system recovery, affects the speed performance of the controller architecture. Therefore, the round-robin reconfiguration scheme currently used will be modified to provide a more evenly distributed workload after processor failure. We will also experiment with simulations of other types of processor failures besides the ALU failures currently simulated.

Using the joint failure simulation code partially installed, we will perform experiments involving processor reconfiguration after joint failure. 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 first be added.

RELATED WORK

The speedup advantages of a multiprocessor architecture have been examined in several robot control architectures (0, 0, 0). The performance improvement is achieved using parallel algorithms in various parts of the control code, such as the forward and inverse dynamics (0, 0, 0, 0). Most of the work with parallel

robot controllers that we have analyzed has attempted to achieve speedup only. However, some work has been done with parallel architectures used to provide processor fault tolerance.

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. (0). MAFT was designed to provide high performance and reliability for a wide range of real-time applications.

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. In the MAFT architecture, each node has a 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 (0) 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. The four computers can be configured in three modes of operation: triple modular redundancy, duplex, and simplex. Once a processor is considered faulty, it is not utilized again until it has been repaired (0).

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.

CONCLUSION

As the performance requirements of robots become more demanding, the response times provided by uniprocessor controller architectures become insufficient. Therefore, multiprocessor architectures are an attractive approach to provide the desired response times. Robot reliability is another area that can benefit from the use of multiprocessor architectures. By incorporating processor failure detection and recovery capabilities, the system is made more reliable. This recovery from processor failure is only possible if there is another processor available to take on the work of the failed processor.

At the cost of adding more processors, better performance is achieved. This performance, however, is degraded when processor fault tolerance is incorporated into the architecture. Some of this performance degradation may be overcome with the addition of more processors. This is the trade-off between high-speed performance, processor fault tolerance, and system cost that must be considered in system design.

We have developed and evaluated several robot controllers. The first is a serial baseline. The second is a parallel version without processor fault tolerance, and the last two are parallel versions with varying levels of fault tolerance. The robot controller hardware is a shared memory multiprocessor architecture. We also simulated a robot for visual analysis of our robot controllers.

In order to test the failure detection and reconfiguration code, we simulated processor failures by modifying the data to be compared by the control processors. Currently, the control processors are only reconfigured due to processor failures. However, when a joint fails, the processor that was controlling that joint may be used to perform some of the tasks of a more heavily loaded processor. We are adding code to simulate this kind of failure, so that we can determine the cost and benefit of processor reconfiguration following joint failure. Joint limits have already been incorporated, and these limits can be changed dynamically.

Our results demonstrate the value of permitting trade-off between performance and processor fault tolerance. Comparison of the serial version of the controller to the parallel version without fault tolerance shows that the use of multiple processors improves the

response time, as expected. The parallel controller would ideally be n times faster than the serial controller. However, some time is lost due to synchronization in the parallel control algorithm. Our controller demonstrates good speedup given the architecture used. The simulation of the serial program on a faster processor shows the level of performance that could be achieved. Use of faster processors would result in better performance.

The performance degradation due to the addition of processor fault tolerance is evident in comparisons between the two versions of the controller without fault tolerance and their fault-tolerant versions. The difference in execution times between the two fault-tolerant versions demonstrates the trade-off between the level of fault tolerance and performance. Our results further show that processor fault tolerance can be incorporated, while still offering better performance than the uniprocessor robot control architecture. Further results will be reported in future papers.

ACKNOWLEDGEMENTS

D.L. Hamilton is supported by NASA Graduate Fellowship #NGT-70251. Research supported in part by NSF under Grant MSS-9024391, and by the Dept. of Energy under Sandia National Laboratory Contract #18-4379A. Use of Sequent Symmetry S81 provided by the Dept. of Computer Science at Rice University under NSF Grant CDA-8619393.

REFERENCES

- R.L. Andersson. Computer Architectures for Robot Control: A Comparison and a New Processor Delivering 20 Real Mflops. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1162–1167, Scottsdale, AZ, 1989.
- 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.
- 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.
- A.S. Deo and I.D. Walker. Robot Subtask Performance with Singularity Robustness using Optimal Damped Least-Squares. In *1992 IEEE International Conference on Robotics and Automation*, pages 434–441, Nice, France, 1992.
- A. Fijany and A.K. Bejczy. Parallel Algorithms and Architecture for Computation of Manipulator Forward Dynamics. In *Proceedings of the 1991 IEEE International Con-*

- ference on Robotics and Automation*, pages 1156–1162, Sacramento, CA, 1991.
- J.H. Graham. Special Computer Architectures for Robotics: Tutorial and Survey. *IEEE Transactions on Robotics and Automation*, 5(5):543–554, October 1989.
- D.L. Hamilton. Performance and Reliability of a Parallel Robot Controller. Master's thesis, Rice University, April 1992.
- D.L. Hamilton, J.K. Bennett, and I.D. Walker. Parallel Fault-Tolerant Robot Control. In *1992 SPIE Conference on Cooperative Intelligent Robotics in Space III*, Boston, MA, November 1992. To appear.
- J.Y. Han and C.Y. Wang. Modeling and Performance Evaluation of Multiprocessor Systems for Real-Time Non-linear Robot Control. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1016–1021, Scottsdale, AZ, 1989.
- R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Tham-bidurai. The MAFT Architecture for Distributed Fault Tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.
- 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.
- 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.
- 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.
- M.W. Spong and M. Vidyasagar. *Robot Dynamics and Control*. Wiley, New York, 1989.
- 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.
- 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.