ACS Implementation of A Robotic Control Algorithm with Fault Tolerant Capabilities

Shu-Yi Yu, Nirmal Saxena, and Edward J. McCluskey

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California 94305-4055

Abstract

This paper demonstrates that an adaptive computing system (ACS) is a good platform for implementing robotic control algorithms. We show that an ACS can be used to provide both good performance and high dependability. An example of an FPGA-implemented dependable control algorithm is presented. The flexibility of ACS is exploited by choosing the best precision for our application. This makes it possible to reduce the amount of required hardware and improve the performance. Results obtained from a WILDFORCE emulation platform showed that even using 0.35μm technology, the FPGA-implemented control algorithm has comparable performance with the software-implemented control algorithm in a microprocessor based on 0.25μm technology. Different voting schemes are used in conjunction with multi-threading and hardware redundancy to add fault tolerance to the robotic controller. Error-injection experiments demonstrate that robotic control algorithms with fault tolerance techniques are orders of magnitude less vulnerable to faults compared to algorithms without any fault tolerant features.

1. Introduction

Adaptive Computing Systems (ACS) augment the traditional Von Neumann processor-memory computation model by a fabric of reconfigurable hardware. One model of the ACS architecture consists of CPU, I/O, memory, and reconfigurable coprocessors. For example, a compiler [Gokhale 98] can partition and map applications into parts that run well on traditional fixed instruction processors and parts that run well on the reconfigurable coprocessor. Commercial implementations of the ACS model range from a single chip [Gokhale 98], a multi-chip board [Annapolis 99][TelSys 99][Xilinx 99], or a multiple-board system [Quickturn 99]. Figure 1 illustrates an example of an ACS using a Xilinx Virtex series FPGA as a reconfigurable component.

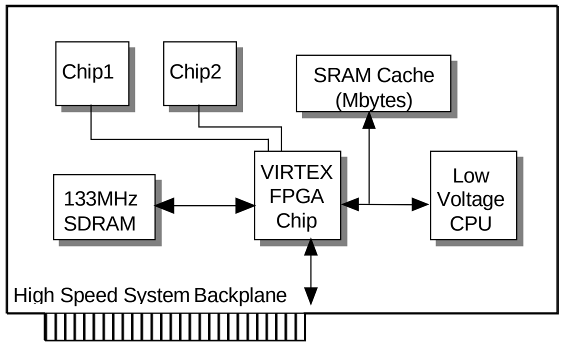


Figure 1. A Xilinx Virtex chip as a system component- an ACS implementation [Xilinx 99].

Applications that are well suited for ACS are: digital signal processing, image processing, control, bit manipulation, compression, and encryption [Saxena 00]. Most of the research and development in the ACS area has been dictated by the performance requirements of target applications. Performance is an important requirement; however, the use of ACS applications in areas like nuclear reactors, fly-by-wire systems, remote (space station) locations, and life-critical systems makes dependability also an important requirement. Dependability in an ACS can be enhanced by exploiting the ability of reconfiguration to repair hardware when faults are present. With very few exceptions [Culbertson 97][Lach 98], little work has been done in the dependable ACS area. This research is part of the ROAR[1] project [Saxena 98], whose objective is to provide dependable computing solutions in an ACS framework. The main thrust of this paper is the instrumentation of robotic control algorithms in ACS environments.

Robots are frequently used under hazardous or remote circumstances, such as in nuclear power plants [Kim 96] or in spacecraft [Wu 93]. Maintenance and repair is usually very expensive and time-consuming for these applications. A lot of research has been done

[1] ROAR stands for Reliability Obtained by Adaptive Reconfiguration.

seeking ways of providing an effective fault tolerant robotic system. Researchers have explored techniques for introducing fault tolerant capabilities into robots. For example, different control schemes are presented in [Ting 94][Yang 98]. Redundancy and voting in joint and actuator levels are suggested in [Tosunoglu 93][Toye 90][Thayer 76]. [Toye 93] and [Hamilton 92] discuss architecture level redundancy, [Hamilton 94][Visinsky 95] present a fault tolerant architecture and framework for robotics. [Toye 93] presents different voting methods in robotic application. Metrics to evaluate fault tolerance versus overhead in robotic systems have also been investigated [Hamilton 96a][Hamilton 96b].

With the exception of [Hamilton 92] and [Hamilton 94], most of the cited research work concentrate in either the control or the mechanical field. Not much work has been done in fault tolerance of the electronic hardware where control algorithms are implemented. A conventional way of controlling a robot is to run the control algorithms on general-purpose computers. Figure 2, for example, shows a mobile Puma 560 Manipulator [Khatib 95], in which a Pentium CPU is used to compute the force strategies. In this paper, we implement the control algorithm on FPGAs to show the feasibility of ACS in the robotic area in terms of high dependability and performance.

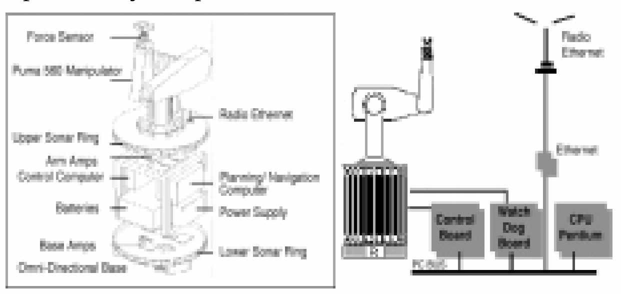


Figure 2. Puma 560 manipulator used in SAMM project [Khatib 95].

In the following sections, we present our implementation of a fault tolerant control algorithm in an ACS environment. Section 2 introduces the robotic control system. Section 3 presents the fault tolerance techniques implemented in the control system. Section 4 describes the implementation of the system in both general-purpose processors and reconfigurable hardware. Results of emulation of the control system are also shown in this section. Section 5 describes the evaluation of fault tolerance techniques and experimental results. Section 6 concludes this paper.

2. Control System

Figure 3 shows a block diagram of a general feedback-control robot system [Craig 89]. It consists of a robot unit and a control unit. The robot unit represents the mechanical part, and the control unit represents the electronic part. The interface between the robot unit and the control unit includes sensors and A/D, D/A converters. The sensors sample the current position of the mechanical part and transform it to an analog signal (voltage or current). The A/D converter converts the signal and sends it to the control unit. The desired trajectory and other control parameters are supplied by the user to the control unit. The control parameters describe the characteristics of the robot unit. The control unit collects data and calculates the needed force. The calculated value is converted to an analog signal, and that signal drives the motor to move the robot.

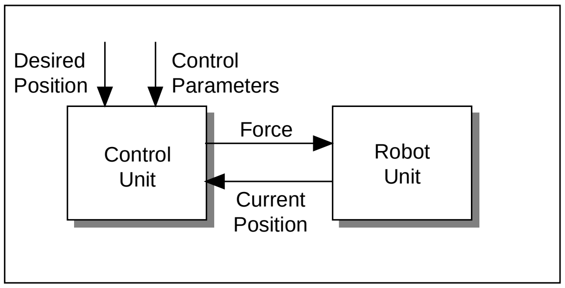


Figure 3. Block diagram of a feedback-controlled robot system.

The robot can be modeled as a second-order linear mechanical system. We apply a second-order linear control algorithm to control the robot. A detailed description of the system and the algorithm is in the appendix. The designs are emulated and implemented in two ACS test-beds: Quickturn's System Realizer [Quickturn 99] and WILDFORCE Board [Annapolis 99]. We have used the Quickturn System Realizer test-bed as a validation tool for our ACS designs. To get more accurate performance and implementation data, we have used the WILDFORCE board because its structure more closely resembles the ACS architecture model (described in Section 1). These test-beds are described in Sec. 4.1.

This paper focuses on the control unit. In the design of the control unit, we use two approaches to implement different fault tolerance techniques. In the first approach, the algorithm is implemented as a fully combinational circuit. In the second approach, the algorithm is implemented in multiple stages and hardware is reused. A detailed description of the control unit is in Sec 4.1.

The performance of the controller affects the performance of a whole robotic system. The rate of producing a corresponding control force is called the servo rate [Craig 89]. The servo rate needs to be fast enough to keep track of the inputs, to reject disturbance from the outside world, to prevent aliasing, and to suppress the natural resonance of the mechanical part. In this paper, the performance achieved by different implementations of the control unit, with or without fault tolerance, is examined and compared.

The controller needs to be designed with proper precision to meet the requirement of the robotic application and to accommodate the characteristics of the robot. The tolerable steady-state error differs among applications, and the resolution and accuracy vary among robots. Hence the needed precision of the controller varies among applications. There is a trade-off among precision, performance, and hardware overhead. To implement the control algorithm, we need to choose the precision that meets all the application requirements while taking less hardware overhead and giving the highest performance. ACS provides the flexibility of varying precision. The design can be optimized by choosing the precision of the system. Section 4.1 discusses experiments on different precision.

3. Fault Tolerance Techniques

Because of safety issues in robotic applications, fault tolerance techniques are needed to improve the dependability of the control system. To determine suitable fault tolerance techniques for our ACS robotic application, two techniques are implemented and compared. One is multi-threading, and the other is hardware redundancy.

The concept of using multi-threading has been described in [Thorton 64] and [Smith 78]. The idea of using multi-threading for fault tolerance was described in [Saxena 98]. This is illustrated in Fig. 4. Figure 4 (a) shows an algorithm executed in multiple stages in limited hardware. Some resources are under-utilized (idle) due to data dependencies and memory latency. By scheduling multiple independent threads most of the idle resources can be reclaimed. To obtain fault tolerance by multi-threading, multiple copies of the same algorithm are executed as multiple threads in the hardware. The final output is obtained by voting among the results from different threads. As shown in Fig. 4 (b), the replicated instructions of the redundant thread can be inserted into the idle stages to minimize timing overhead. In a fault-free scenario, identical outputs are produced from all threads. When faults are present, the redundant threads provide a means for fault detection and fault masking.

In hardware redundancy, the hardware is replicated into multiple copies. The input is connected to all of the modules. In case of two copies, a comparator is used to detect the errors. For three copies and more, a voter is used to determine the final output among the outputs from the modules. The replicated copies provide fault detection and possibly fault masking capabilities. Figure 4 (c) illustrates an example of duplex system (two-modular hardware redundancy).

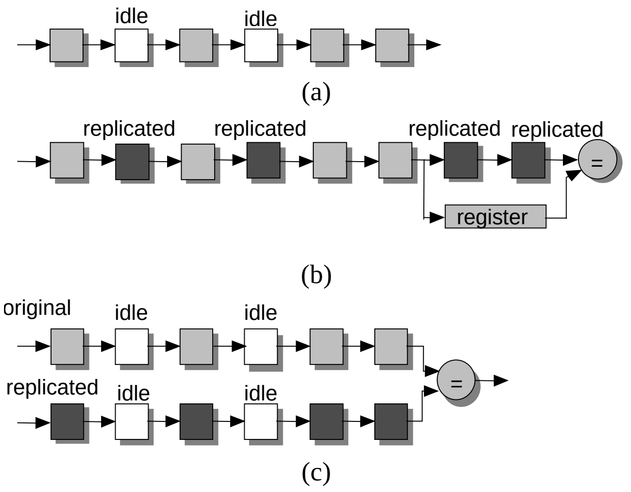


Figure 4. Multi-threading vs. hardware redundancy: (a) an algorithm in multiple stages, (b) multi-threading, and (c) hardware redundancy.

Possible voting schemes used in conjunction with fault tolerance techniques include exact (bit-by-bit) majority voting, median voting, weighted average voting, plurality voting, and others. In an NMR (N-Modular Redundancy) system, these voting schemes are indistinguishable when there is only one faulty module. These different voting schemes behave differently in the presence of multiple-module failures. Comparison among various voting schemes is reported in [Bass 97]. In this paper, we implement two voters in our design and compare their behavior. One is the majority voter, which is widely used, and the other is the median voter, which performs the best in Bass' paper. The majority voter selects the bit-wise output agreed by the majority among the modules. The median voter selects the median value of the outputs from different modules.

Error detection mechanism is added in conjunction with voters to provide error-detecting ability. Figure 5 illustrated the voter with error detection. The inputs are compared with the voted output, and if it is different, error signals will be generated.

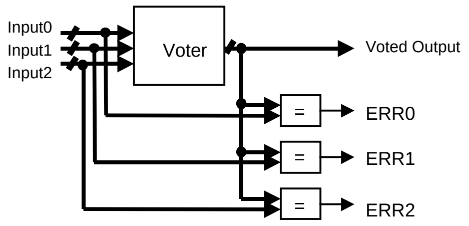


Figure 5. A voter with error detection mechanism.

To evaluate the fault tolerance techniques, we considered dependability, and overhead in terms of area and performance degradation. The area and performance are examined by implementing the system in reconfigurable hardware. The experiments are described in Sec 4. The dependability of the systems is examined through fault injection experiments that yield data on numbers of detected faults and corruption. Fault injection experiments are described in Sec. 5.

4. Implementation of the Control Algorithm

The feasibility of implementing robotic algorithms in ACS environment is demonstrated by the implementation of a second-order linear control algorithm in ACS hardware, Quickturn™ System Realizer and WILDFORCE® board. For the purposes of comparison, the algorithm is also implemented in C programs running on general-purpose processors. To investigate the area overhead and the performance degradation of designs with fault tolerant features, different fault tolerance techniques are implemented in the control system for a comparative evaluation. In addition, the controllers are designed using different numbers of bits to exploit the flexibility of ACS hardware.

Section 4.1 describes implementation in ACS hardware. Section 4.2 explains the implementation of the C program and shows performance data. Section 4.3 discusses our results.

4.1 ACS Implementation

To demonstrate the feasibility of the dependable ACS implementation of robotic algorithms, we use a commercial reconfigurable system, Quickturn's System Realizer [Quickturn 99], and a multi-chip FPGA board, the WILDFORCE board [Annapolis 99], as our platforms. The controller is initially emulated in the Quickturn's System Realizer to verify its functionality. It is then implemented on the WILDFORCE board.

Quickturn's M250 System Realizer is an emulator from Quickturn Design Systems. The System Realizer comprises of one Logic Module (LM), one Probe Module (PM), and one Control Module (CM). A LM contains Quickturn MUX chips and 80 Xilinx LCA4013 FPGAs. In each FPGA, up to 13k logic gates can be implemented. The LM performs the logic function of the design. The PM contains an integrated logic analyzer and stimulus generator. The PM applies test vectors to the design and records output responses. The CM contains an embedded CPU and hard-disk drives. It controls the System Realizer and provides network connectivity. Additional Target Interface Modules (TIMs) can be used for the connection of emulation engines and the target system for in-circuit emulation. The I/O cards in TIMs can be replaced with other devices, such as embedded memory or analog circuits.

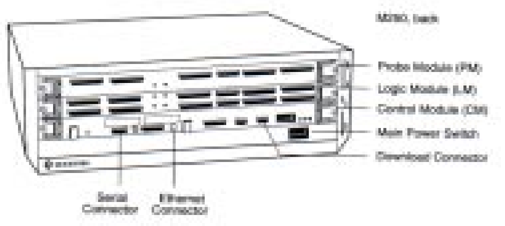
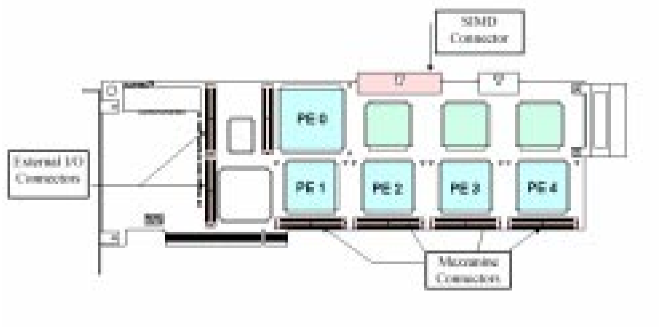
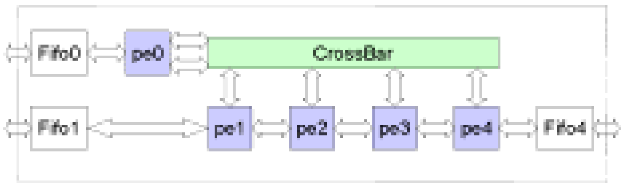


Figure 6. Quickturn's M250 System Realizer [Quickturn 99]

The WILDFORCE board is a PCI-based custom reconfigurable multi-chip system. A layout and a block diagram of the WILDFORCE board are shown in Fig. 7. The system has five processing elements: one Control/Processing Element (CPE) and four Processing Elements (PE1 through PE4.) The PEs consist of Xilinx 4036XLA FPGAs, each of which can realize up to 36k logic gates. Each PE has a 1Mb of local memory. All of the local memories are accessible to the host processor through the PCI bus. A crossbar is used to provide communication among PEs on the board. Each PE has one 36-bit connection to the crossbar, and CPE has two. The crossbar is composed of three Xilinx XC4006E FPGA devices. The PE array connects to the PCI bus through a set of bi-directional synchronous FIFOs. The maximum frequency of FIFO access is 50MHz. The board is capable of generating PE clock rates ranging from 300kHz to 55MHz.



(a)



(b)

Figure 7. The WILDFORCE board [Annapolis 99]: (a) a layout, and (b) a block diagram.

In the control unit design, we use two architectures to implement different fault tolerance techniques. To implement multi-threading, the control algorithm is divided into multiple stages, and the hardware resources are reused. The computation is replicated as different threads. The threads share the same hardware resources. To implement hardware redundancy, the control unit is designed in combinational circuit and replicated.

To find the suitable precision for our control system, the controller is designed in different precision. We evaluated controllers with the robotic characteristics, such as the response time, the steady-state error, and the time to stabilization. Among all systems that meet the application requirement, the lowest-precision design is selected to minimize the hardware overhead and to optimize the performance.

Table 1 lists the simulated response data of systems with different precision in Verilog. The precision is 32, 24, 16, and 12 bits, respectively. With the given test trajectory, the systems work well in all precision except for 12 bits. In the 12-bit system, arithmetic overflow occurs resulting in overshoot of the output waveform. As shown in table 1, the response time and the time to stabilization of systems with 32, 24, and 16-bit numbers are almost the same. The major difference among the systems is accuracy: a higher-precision system has less steady-state error. Therefore, in terms of the response of the system, as long as the requested data range does not cause arithmetic overflow in the system and the steady state error is tolerable, a low-precision system is sufficient for the application while taking less area and shorter execution time than a higher-precision design. From the simulation results, the 16-bit system was the implemented choice on the WILDFORCE board.

Table 1. Square wave response for systems with different precision.

Number of bits	32	24	16	12
Steady state error(%)	0.006	0.1	2	---
Time to stabilization (<0.1%) (cycles)	135	133	145	
Response time (cycles)	1	1	1	

The designs emulated on Quickturn's emulator are written in Verilog RTL code. The code is synthesized and compiled by Quest software. The compiled data are downloaded to the emulator. Test vectors are applied to the emulated circuit to verify the designs.

Table 2 lists the designs emulated on the Quickturn's System Realizer. The designs use multi-threading as the fault tolerance technique. For the three-thread design, both 32-bit and 16-bit numbers are implemented.

Table 2. Emulation on Quickturn's System Realizer.

Threads	1	2	3	
Number of bits	32	32	32	16
Number of FPGAs	29	29	103[2]	20
# of Stages	7	9	11	11

The designs implemented on the WILDFORCE board were written in VHDL code. The code was synthesized with Synplify. The placement and routing was done with the software provided by Xilinx.

The control system is implemented in two PEs. The control unit is in PE1, and the robot unit is in PE2. The two units are separated in two chips for fault injection purposes. The two units communicate through the bus between PE1 and PE2. Input vectors are applied from the computer through FIFO1. Output values are transmitted to the host computer through FIFO4.

Tables 3 and 4 show the emulation results on the WILDFORCE board. In the tables, the hardware area is represented by the number of Look Up Tables (LUTs) and registers used in FPGAs. In the Xilinx XC4000 series FPGAs, the logic function is provided by Configurable Logic Blocks (CLBs). A CLB consists of two four-input LUTs (F LUTs), one three-input LUT (H LUT), and two registers. The number of LUTs is the total number of mapped F LUTs and H LUTs. The data is obtained from report files of a Xilinx placement-and-route tool.

Table 3. Emulation results on WILDFORCE board – hardware redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Med	Maj
LUTs (total 3888)	404	819	1284	1257
Registers (total 2592)	115	230	345	349
Frequency (MHz)	24	26	22	24
Latency (μs)	0.042	0.038	0.045	0.042

Table 4. Emulation results on WILDFORCE board – multi-threading.

# of Threads	1	2	3	3
Voter Type	--	---	Med	Maj
LUTs (total 3888)	409	814	1248	1251
Registers (total 2592)	362	636	945	945
# of Stages	7	8	11	11
Frequency (MHz)	50	50	50	50
Latency (μs)	0.14	0.16	0.22	0.22

[2]This design was emulated in Quickturn M3000 System Realizer, which is larger than M250. Other designs shown on the table were emulated in Quickturn M250 System Realizer.

4.2 Control Algorithm on General-Purpose Processors

We also implemented the control algorithm in C programs and ran them on Ultra Sparc II and Pentium II Xeon processors. Two data types are used: single precision floating-point numbers (32 bits), and fixed-point numbers (32 bits). Redundancy is achieved by duplicating the code inside the program. Table 5 shows the latencies of the C programs on general-purpose processors. The latencies for floating-point numbers range from 0.184 to 11.704μs, while those for fixed-point numbers range from 0.033 to 0.646μs. These results are compared with the performance of the ACS implementation in the next section.

Table 5: Performance of the control algorithm on general-purpose processors.

Latency (μs)					
# of Copies	Voter Type	Ultra SPARC[3] (300MHz)		Pentium II Xeon (450MHz)	
		Float	Fixed	Float	Fixed
1	---	0.184	0.099	3.866	0.033
2	---	0.290	0.376	7.699	0.080
3	Majority	0.481	0.641	11.182	0.126
3	Median	0.462	0.646	11.704	0.126

4.3 Discussion

In this section, the results from the implemented designs on the WILDFORCE board are compared with the results obtained from general-purpose processors.

As shown in Table 3, the designs with hardware redundancy have latencies ranging from 0.038μs to 0.045μs. As shown in Table 4, all designs with multi-threading are limited by the maximum frequency of the board, which is 50MHz. The latencies range from 0.14μs to 0.22μs, which are proportional to the number of stages. The designs with hardware redundancy have shorter latencies than the multi-threaded designs because they are fully combinational.

As the area results show, the area overhead is roughly proportional to the number of modules or threads. It does not differ much between different fault tolerance techniques or different voting schemes except in the number of registers. Multi-threaded designs require more registers because hardware is reused and intermediate results need to be stored. For hardware redundancy, results are coherent with what can be inferred by observation of Fig. 4 (c), that is, resources are replicated. However, in the designs with multi-threading, even when resources are reused, the area overhead is not significantly reduced. The reason is that in the implemented algorithm, the area of the reused hardware resources is small compared to the additional registers and the interconnection resources for those registers, which are proportional to the number of threads. In systems that use large and complicated hardware to process data, multi-threading may take advantage of reusing resources and may reduce hardware overhead. In our case, the designs with hardware redundancy have better performance and less area overhead than those with multi-threading.

To evaluate the performance of the designs in ACS hardware, Fig. 8 shows the best performance from the algorithm in both general-purpose processors and reconfigurable hardware. Pentium II Xeon processor is in 0.25μm technology and Xilinx FPGA chips used in the WILDFORCE board are based on 0.35μm technology. It is shown that even when an older generation FPGAs were used, the performance obtained in reconfigurable hardware is comparable to that of the later generation general-purpose microprocessors. Note that to exploit the flexibility of ACS and to customize the design, the controller on the WILDFORCE board is designed in 16-bit fixed-point numbers.

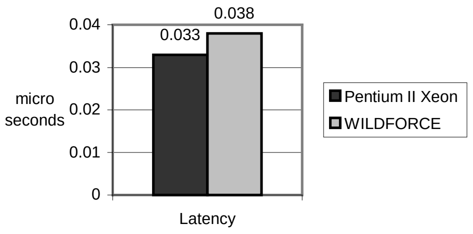


Figure 8. Performance of implementations on general-purpose processor and ACS.

5. Evaluation of Fault Tolerance Techniques

To enhance the dependability of the control system, we implemented fault tolerance techniques into the robotic controller. Evaluation is needed to determine the suitable fault tolerance technique for the implemented system in reconfigurable hardware. We consider dependability and overhead in terms of area and performance degradation for evaluation in our implementation.

The evaluation of dependability is done by injecting faults into the control unit of the system. Hence detection coverage and fault tolerance capabilities can be determined. To model stuck-at faults at combinational gates' outputs, the injected faults include stuck-at-one and stuck-at-zero faults at outputs of LUTs used by the design. All possible single stuck-at faults in each design are examined. To compare the effectiveness of the voters under the presence of multiple faults, double stuck-at faults are injected in designs with majority voters and

[3] The listed results may be pessimistic due to the inaccessibility of 64-bit compiler.

median voters. For each examined design, five thousand stuck-at fault pairs are chosen randomly. Output vectors of fault-injected designs are compared with those of fault-free designs. An output is considered corrupted if its value is different from the output of fault-free systems. Tables 6 and 7 list the results of single stuck-at fault injection experiments. Table 8 lists the results of double stuck-at fault injection experiments.

Table 6. Single stuck-at fault injection in designs with multi-threading.

# of Threads	1	2	3	3
Voter Type	---	---	Med	Maj
Injected Faults	936	1843	2803	2830
Corrupted w/o Detection	**806**	**18**	**39**	**7**
Corrupted w/ Detection	0	788	121	46
Detected w/o Corruption	0	785	2284	2404
No Detection & Corruption	130	252	359	373

Table 7. Single stuck-at fault injection in designs with hardware redundancy.

# of Modules	1	2	3	3
Voter Type	---	---	Med	Maj
Injected Faults	952	1932	2932	2812
Corrupted w/o Detection	**825**	**1**	**4**	**1**
Corrupted w/ Detection	0	835	102	32
Detected w/o Corruption	0	857	2442	2428
No Detection & Corruption	127	239	384	351

Table 8. Double stuck-at fault injection in designs with different voters (5000 fault pairs are randomly chosen).

Fault Tolerance Technique	Multi-Threading (3 threads)		Hardware Redundancy (3 copies)	
Voter Type	Med	Maj	Med	Maj
Corrupted w/o Detection	**24**	**7**	**0**	**0**
Corrupted w/ Detection	1754	1437	1656	1796
Detected w/o Corruption	3139	3464	3250	3125
No Detection & Corruption	83	92	94	79

Tables 6 and 7 show that in the single-threaded and single-modular designs, which do not have any fault tolerance or error-detecting scheme, almost all of the injected faults result in corruption. For the designs with fault tolerance techniques, the undetected corruption caused by single faults only occur when the fault is injected in the comparator (error detector), I/O logic, or the common logic part, such as the scheduler. Therefore the undetected corruption is less likely to appear.

The two-threaded and two-modular designs only have error detecting but not fault masking mechanism. In these designs, the number of detected corruption is roughly the same as the number of detection without corruption. The reason is because the output of the designs is from one of the two identical threads or modules, and the probability of fault occurrence in the output thread or module is roughly 50%. The reliability can be improved if diversity is used between modules or threads [Mitra 99].

From the single stuck-at fault results, the three-threaded and three-modular designs, which have fault-masking mechanism, have much lower numbers of corruption than other designs. The results of both single and double stuck-at faults show that the designs with a majority voter produce smaller numbers of undetected corruption than the designs with a median voter. It means that the median voter is more vulnerable to stuck-at faults than the majority voter, and is less able to detect corruption. The reason is because a stuck-at fault may affect several outputs of the median voter at the same time due to the error propagation through the carry chain. On the other hand, in a majority voter, voting is done bit-by-bit. Therefore, a stuck-at fault will affect, in the worst case, one bit of the outputs of the majority voter.

The numbers for undetected corruption in designs with hardware redundancy are much smaller than those of designs with multi-threading. It is because multi-threaded designs have more hardware that is commonly shared by different threads, such as schedulers and control logic. Therefore, faults are more likely to affect resources that are shared by several threads, and consequently, an error detection mechanism will not detect them. On the contrary, the designs with hardware redundancy replicate the whole module, hence a single fault can affect only one module, and corruption occurs only if the fault is in the voter.

Recalling the results of implementation in Sec. 4, the designs with hardware redundancy have better performance and less area overhead than those with multi-threading. From the fault injection experiments in this section, we can conclude that in our implementation, hardware redundancy provides higher dependability and has lower overhead. Therefore, hardware redundancy is more adequate to implement our robotic application than multi-threading.

6. Conclusion

We have demonstrated that an ACS is a good platform for implementing robotic control algorithms. As a case study, a robotics control algorithm has been implemented on reconfigurable hardware. Results obtained from hardware emulation on the WILDFORCE platform have demonstrated that the performance of running the algorithm on reconfigurable hardware is comparable with that on a general-purpose processor. The ability of implementing systems with different precision to customize the design has demonstrated the advantage of an adaptive computing system. In conjunction with different voting schemes, fault tolerance techniques have been used in implementing the algorithm. Fault injection experiments have shown that hardware redundancy is more suitable for the implemented control system.

7. Acknowledgements

The authors would like to thank Dr. Santiago Fernandez-Gomez, Philip Shirvani, Subhasish Mitra, Chao-Wen Tseng, and Siegrid Munda for their useful feedback and suggestions. We would also like to thank Mike Butts and Quickturn Design Systems for their technical support. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

References

[Annapolis 99] Annapolis Micro Systems Inc., www.annapmicro.com, 1999.

[Bass 97] Bass, J. M., G. Latif-Shabgahi, and S. Bennett, "Experimental Comparison of Voting Algorithms in Cases of Disagreement," *Proceedings. 23rd Euromicro Conference: New Frontiers of Information Technology*, pp. 516-23, 1997.

[Craig 89] Craig, J. J., *Introduction to Robotics: Mechanics and Control*, Addison-Wesley Publishing Company, 2nd edition, 1989.

[Culbertson 97] Culbertson, W. B., R. Amerson, R. J. Carter, P. Kuekes, and G. Snider, "Defect Tolerance on the Teramac Custom Computer, " *Proc. IEEE Symp. FCCM '97*, pp. 116-123, Apr. 1997.

[Gokhale 98] Gokhale, M. and J. M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", *Proc. IEEE Symp. FCCM'98*, Apr. 1998.

[Hamilton 92] Hamilton, D. L., J. K. Bennett, and I. D. Walker, "Parallel Fault-Tolerant Robot Control," *Proceedings of SPIE*, Vol. 1829, pp. 251-61, 1992.

[Hamilton 94] Hamilton, D. L., M. L. Visinsky, J. K. Bennett, J. R. Cavallaro, and I. D. Walker, "Fault Tolerant Algorithms and Architectures for Robotics", *Proceedings of MELECON '94 Mediterranean Electrotechnical Conference*, Vol. 3, pp. 1034-6, 1994.

[Hamilton 96a] Hamilton, D. L., J. R. Cavallaro, and I. D. Walker, "Risk and Fault Tolerance Analysis for Robotics and Manufacturing," *Proceedings of 8th Mediterranean Electrotechnical conference on Industrial Applications in Power Systems, Computer Science and Telecommunications*, Vol. 1, pp. 250-5, 1996.

[Hamilton 96b] Hamilton, D. L., I. D. Walker, and J. K Bennett, "Fault Tolerance Versus Performance Metrics for Robot Systems", *Proceedings of IEEE International Conference on Robotics and Automation*, Vol. 4, pp. 3073-80, 1996.

[Khatib 95] Khatib, O., K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, A. Casal, and A. Baader, "Force Strategies for Cooperative Tasks in Multiple Mobile Manipulation Systems", *Robotics Research 7, The Seventh International Symposium*, G. Giralt and G. Hirzinger, eds., pp. 333-342, Springer 1996.

[Kim 96] Kim, J. H., J. K. Lee, H. S. Eom, and J. W. Choe, "An Underwater Mobile Robotic System for Reactor Vessel Inspection in Nuclear Power Plants," *Proc. of the Sixth International Symposium on Robotics and Manufacturing*, pp. 351-6, 1996.

[Lach 98] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs, " *Proc. ACM/SIGDA Int'l. Symp. on FPGAs*, pp. 105-115, Feb 1998.

[Mitra 99] Mitra, S., N.R. Saxena, and E.J. McCluskey, "Design Diversity for Redundant Systems," *29th International Symposium on Fault-Tolerant Computing (FTCS-29)* Fast Abstracts, Madison, WI, pp. 33-34, June 15-18, 1999.

[Quickturn 99] Quickturn Design Systems (now part of Cadence Design), www.quickturn.com or www.cadence.com, 1999.

[Saxena 98] Saxena, N. R. and E.J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y. Yu, and E. J. McCluskey, "Dependable Computing and Online Testing in Adaptive and Configurable Systems," *IEEE Design and Test*, pp. 29-41, 2000.

[Smith 78] Smith, B. J., "A pipeline, Shared Resource MIMD Computer", *Proc. Intl. Conf. on Parallel Processing*, pp. 6-8, 1978.

[TelSys 99] TSI TelSys Inc., www.tsi-telsys.com, 1999.

[Thayer 76] Thayer, W. J., "Redundant Electrohydraulic Servoactuators," *Technical Bulletin 127*, MOOG INC. Controls Division, 1976.

[Thorton 64] Thorton, J. E., "Parallel operation in the Control Data 660", *AFIPS Conf. Proc., Fall Joint Computer Conf.*, Vol. 26, pp. 33-40, 1964.

[Ting 94] Ting, Y., S. Tosunoglu, and B. Fernandez, "Control Algorithms for Fault-Tolerant Robots," *Proceedings of 1994 IEEE International Conference on Robotics and Automation*, Vol. 2, pp. 910-15, 1994.

[Tosunoglu 93] Tosunoglu, S., "Fault Tolerance for Modular Robots," *Proceedings of IECON*, Vol. 3, pp. 1910-14, 1993.

[Toye 90] Toye, G., *Management of Non-homogeneous Functional Modular Redundancy for Fault Tolerant Programmable Electro-mechanical Systems*, Ph.D. thesis, Stanford University 1990.

[Toye 93] Toye, G. and L. J. Leifer, "Helenic Fault Tolerance for Robots," *Computers & Electrical Engineering*, Vol. 20, pp. 479-97, 1993.

[Visinsky 95] Visinsky, M. L., J. R. Cavallaro, and I. D. Walker, "A Dynamic Fault Tolerance Framework for Remote Robots", *IEEE Transactions on Robotics and Automation*, Vol. 11, No. 4, pp. 477-490, August 1995.

[Wong 96] Wong, K. K., E. C. Tan, and A. Wahab, "A VLSI Median Voter for Fault Tolerant Signal Processing Applications," *3rd International Conference on Signal Processing Proceedings*, Vol. 2, pp. 1574-7, 1996

[Wu 93] Wu, E. C., J. C. Hwang, and J. T. Chladek, "Fault Tolerant Joint Development for the Space Shuttle Remote Manipulator System: Analysis and Experiment," *IEEE Trans. On Robotics and Manufacturing—Recent Trends in Research, Education, and Applications*, Vol. 9, No. 5, pp. 675-80, 1993.

[Yang 98] Yang, J.-M. and J.-H. Kim, "Fault-Tolerant Locomotion of the Hexapod Robot," *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, Vol. 28, No. 1, pp. 109-116, February 1998.

[Xilinx 99] Xilinx, Inc, www.xilinx.com, 1999.

Appendix A: Second-Order Linear Control Algorithm

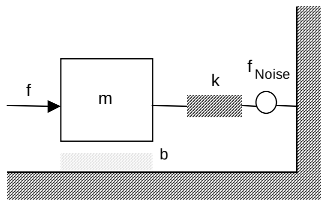


Figure 9. A second-order linear mechanic system.

Consider a second-order linear mechanic system. As shown in Fig. 9, an object of mass *m* is connected to the wall by a spring with coefficient *k*. Assume the frictional force is proportional to the object's velocity, and the friction coefficient is *b*. The position of the object is described by the variable *x*. Assume there exists a noise force f_{Noise}. The origin (x=0) is defined as the resting position of the object. Then the equation of motion is

(eq 1) $$f_{Noise} + m\ddot{x} + b\dot{x} + kx = f$$

We want the object to move along the desired trajectory, $x_d(t)$. To control the motion of the object, we want to apply a force $f(t)$ to the object parallel to x axis. The control law is

(eq 2) $$f=m(\ddot{x}_d + K_d\dot{e} + K_p e + K_i \int edt) + b\dot{x} + kx$$

where

$$e = (x - x_d)$$

By equating f of equation 1 and 2, and differentiating both sides, the result is obtained in the form of a third order differential equation in terms of e:

(eq 3) $$\dddot{e} + K_d\ddot{e} + K_p\dot{e} + K_i e = \frac{\dot{f}_{Noise}}{m}$$

From equation 3, by assigning different values to the control gains K_p, K_i, and K_d, we can make $e(t)$ approach zero quickly. That is, $x(t)$ can approach $x_d(t)$ quickly if we choose proper values for K_p, K_i, and K_d.

A.1 Robot Block

In our design, the behavior of the robot block is emulated using hardware. We are going to solve x in the equation of motion (eq 1) to obtain the current position of the robot.

In a discrete-time system, differentiation and integration are performed as

(eq 4) $$\int x(t)dt = \sum x(t)\Delta t = \sum x(n\Delta t)\Delta t = \sum x[n]\Delta t$$

(eq 5) $$\dot{x}(t) = \frac{x(t) - x(t-\Delta t)}{\Delta t} = \frac{x(n\Delta t) - x(n\Delta t - \Delta t)}{\Delta t} = \frac{x[n] - x[n-1]}{\Delta t}$$

Then the original equation of motion (eq 1) becomes

(eq 6) $$f_{Noise}[n] + m\left(\frac{x[n] - 2x[n-1] + x[n-2]}{(\Delta t)^2}\right) + b\left(\frac{x[n] - x[n-1]}{\Delta t}\right) + kx[n] = f[n]$$

We can combine the Δt part into the coefficients. Therefore,

(eq 7) $$f_{Noise}[n] + m'(x[n] - 2x[n-1] + x[n-2]) + b'(x[n] - x[n-1]) + kx[n] = f[n]$$

where

$$m' = \frac{m}{(\Delta t)^2}$$

$$b' = \frac{b}{\Delta t}$$

Solving equation 7 in discrete time,

(eq 8) $$x[n] = \frac{m'(2x[n-1] - x[n-2]) + b'x[n-1]}{m' + b' + k} + \frac{f[n] - f_{Noise}[n]}{m' + b' + k}$$

The robot block will compute the current position from the above equation. The force $f[n]$ is the output of the control block.

A.2 Control Block

In the control block, we are going to compute the needed force f from the control law:

(eq 2) $$f=m(\ddot{x}_d + K_d\dot{e} + K_p e + K_i \int edt) + b\dot{x} + kx$$

Similar to the robot block, we still use subtraction and addition to replace differentiation and integration in our discrete-time system. Therefore, the equation becomes

(eq 9) $$f[n] = m'(x_d[n] - 2x_d[n-1] + x_d[n-2]) + K_d'(e[n] - e[n-1]) + K_p e[n] + K_i'\sum e[n] + b'(x[n] - x[n-1]) + kx[n]$$

where

$$m' = \frac{m}{(\Delta t)^2}$$

$$b' = \frac{b}{\Delta t}$$

$$K_d' = \frac{K_d}{\Delta t}$$

$$K_i' = K_i\Delta t$$

In the control block, the inputs are the control gains K_p, K_i', and K_d'– whose values are decided by the user– and the coefficients *m'*, *b'*, and *k*– which describe the motion of the second-order linear system. In practical, *m*, *b*, and *k* are obtained by measuring the characteristics of the robot; in addition, the sampling interval Δt is decided by the system. Throughout this paper, we have used *m'*=40, *b'*=10, *k'*=10, K_p=0.4, K_i'=0.064, and K_d'=0.4.