Evolutionary Fault Recovery in a Virtex FPGA Using a Representation That Incorporates Routing

Jason Lohn[1], Greg Larchev[1], and Ronald DeMara[2]

[1] Computational Sciences Division, NASA Ames Research Center,
Mail Stop 269-1, Moffett Field, CA 94035-1000, USA
email: {jlohn, glarchev}@email.arc.nasa.gov
[2] School of Electrical Engineering and Computer Science,
University of Central Florida, Orlando, FL 32816-2450
email: demara@mail.ucf.edu

Abstract. Most evolutionary approaches to fault recovery in FPGAs focus on evolving alternative logic configurations as opposed to evolving the intra-cell routing. Since the majority of transistors in a typical FPGA are dedicated to interconnect, nearly 80% according to one estimate, evolutionary fault-recovery systems should benefit by accommodating routing. In this paper, we propose an evolutionary fault-recovery system employing a genetic representation that takes into account both logic and routing configurations. Experiments were run using a software model of the Xilinx Virtex FPGA. We report that using four Virtex combinational logic blocks, we were able to evolve a 100% accurate quadrature decoder finite state machine in the presence of a stuck-at-zero fault. Evolutionary experiments with the hardware in the loop have begun and we discuss the preliminary results.

1 Introduction

Numerous advantages of Field Programmable Gate Arrays (FPGAs) in space-borne electronics have been identified in recent research publications [3, 15] and manufacturers' literature [1, 16]. Benefits include reconfiguration capability to support multiple missions, the ability to correct latent design errors after launch, and the potential to accommodate on-chip and off-chip failures. Ground Support Equipment (GSE) based FPGA applications primarily employ reprogrammable devices as a means of amortizing development costs over multiple missions. In GSE-enabled applications such as Reusable Launch Vehicles (RLVs), FPGAs are configured or replaced between missions rather than being reprogrammed during flight. For applications such as RLVs, comparatively short mission durations and low levels of ionizing radiation are involved. Hence for many ground reconfigurable applications, conventional Triple Modular Redundancy (TMR) techniques often provide sufficient fault handling coverage.

On the other hand, in-mission reconfigurable FPGAs are advantageous for deep space probes, satellites, and extraterrestrial rovers. In these applications, the radiation exposures, mission durations, and repair complexities are significantly greater. The need for adequate fault coverage during these missions has become further intensified by the increasing number of FPGAs being deployed. For instance, NASA's Stardust probe contains over 100 FPGA devices. Although the Stardust's FPGAs are based on a non-reprogrammable antifuse-based technology, a more recent space-qualified SRAM-based technology has become commercially available.

In SRAM-based devices, the number of programming cycles is unlimited. Hence new techniques become feasible for active recovery through reconfiguration of a compromised FPGA. The approach developed here concentrates on autonomous reconfiguration of SRAM-based devices while in-flight. The experiments conducted involve Xilinx's SRAM-based Virtex parts from the same device family as the space-qualified QPRO radiation-hardened series.

Permanent Single-Event Latchup (SEL) failures may impact CLBs and/or programmable interconnections within the FPGA itself. They may also involve other supporting devices that the FPGA interfaces with or processes data from. These failure modes also suggest that the ability to derive an alternative FPGA configuration in-situ would be beneficial. Likewise, SEL exposures exist with regards to the data processing path within the FPGA that is not involved with the device's programmable configuration. In the above cases, the FPGA configuration derived at design time will no longer provide the required functionality for the damaged part. Traditionally, redundant spares have been utilized to replace the damaged device.

Autonomous repair provides an alternative to device redundancy as a means of restoring lost capability. While redundant spares exist only in limited quantities, evolutionary recovery methods attempt to facilitate repair through reuse of damaged parts. Hence the potential benefits are two-fold. First, one or more failures might be accommodated by reconfiguring the failed part without incurring the increased weight, size, or power traditionally associated with providing redundant spares. Second, the characteristics of the failure need not be precisely diagnosed in order to be repaired. Here the repair is performed in-situ via intrinsic evaluation of the device's remaining functionality. This implies that any residual functionality, including the electrical characteristics of both the damaged device and its interaction with any supporting devices, is taken into account when realizing the repair. After isolating the fault to a size that is manageable for the evolutionary algorithm, alternate solutions are refined though iterative selection. This can be carried out without detailed knowledge of the underlying failure mechanism itself.

The approach developed here attempts to regain lost functionality due to a fault by evolving a new configuration on the defective FPGA. We assume a dual-redundant FPGA system whereby the faulty FPGA undergoes evolution to recover its functionality while the redundant FPGA maintains proper functionality during evolution on the faulty FPGA. Thus after a fault is detected, redundancy is lost for a short period of time and then restored. Application functionality is maintained throughout this process under the assumption that only one of the FPGAs fails. Our results show that the evolutionary methods are able to fully recover from a simulated stuck-at-zero fault in the input of a state machine implementing a quadrature decoder. Several research challenges remain and they are also discussed.

2 Related Work

Recently, various evolutionary algorithm approaches have been proposed for fault-recovery of FPGAs. Some previous work applies evolutionary algorithms prior to the occurrence of the fault while other approaches attempt to repair the fault after its occurrence. Some techniques involve intrinsic evolution using the failed part itself. Others rely on extrinsic evolution of an abstracted model of the devices.

Three examples of recent work that apply evolutionary algorithms to realize fault-tolerant designs include [11], [4], and [13]. In [11], Miller examined properties of messy gates whereby evolved logic functions inherently contain redundant terms as their functional boundaries change and overlap. In [4], Canham and Tyrrell compare the fault tolerance of oscillators evolved by including a range of fault conditions within the fitness measure during the evolutionary process. A population-based approach scores evolved designs using a fitness function corresponding to desired operation based on the absence of faults. When evolution is complete, an additional pass evaluates the ability of the evolved individuals to tolerate a range of faults, and the most fault-tolerant individuals are retained. In [13], the evolution of designs containing redundant capabilities without the designer having to explicitly specify the redundant parts themselves was investigated. To achieve this, a range of fault cases was introduced throughout the evolution process. This allowed individuals to exploit whatever component behaviors exist, even behaviors known to be faulty.

An evolutionary fault-recovery approach is described by Vigander [14]. He develops a genetic algorithm to restore functionality after random faults are injected into a 4-bit by 4-bit multiplier using standard genetic operators. He simulated the repair of the prior-designed multiplier that consisted of feed-forward interconnection of hypothetical FPGA cells capable of 8 different logic functions. He used as his fitness function the number of correct input-output mappings from the 256 possible input combinations that could be applied to the multiplier. He demonstrated that while it is not exceedingly difficult to derive a solution that can produce a nearly correct repair, completely correct repairs present a challenging problem. To remedy this, he demonstrated that a voting system with as few as three alternatively evolved repaired circuits was capable of producing a majority output that was completely correct.

3 Representation and Operators

Several goals were taken into account while designing the representation scheme. Amenability to recombination is of course a primary concern. After that, our priorities were to let the GA work in the largest, most flexible design space as possible: we wanted to allow all possible LUT configurations and allow the maximum number of CLB interconnections given the constraints of hardware routing support. We also wanted to disallow illegal configurations and to minimize non-coding alleles (introns).

Bitstring representations are a natural choice for FPGA applications, and many times the raw configuration string can be used as the representation. In our case, we chose a bitstring representation mainly out of convenience in programming. Since we knew that only a handful of CLBs would be evolved, our bitstrings would be at most 1000 bits long. We acknowledge that this approach would likely suffer as more CLBs were utilized and the corresponding bitstring enlarged to thousands of bits.

The representation is shown in Figure 1. This scheme is comprised of multiple 128-bit fields, one for each CLB. Within each CLB field are a number of sub-fields that specify each of the LUT bits and remote connections. There are 16 bits that specify the contents of each LUT. Each LUT has four inputs, and since each of these inputs can be connected to other LUT outputs, the remote CLB/LUT requires addressing bits. Since our system will be comprised of four CLBs, we need only two bits to specify the remote CLB, and another two bits to specify the particular LUT within the CLB. This pattern of sub-fields continues for each LUT until all the LUTs in the CLB are accounted for. An illustration of the CLBs, LUTs and sample routing is shown in Figure 2.

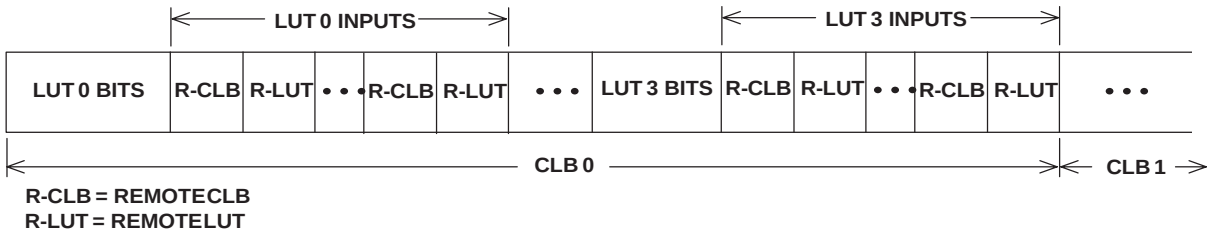


Fig. 1. Genetic representation used showing logic fields and routing fields.

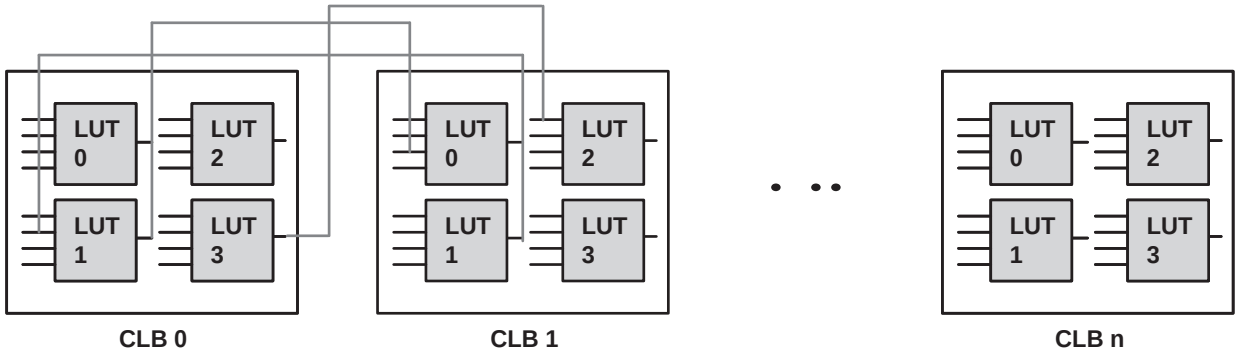


Fig. 2. Example of routing among CLBs.

The operators employed were crossover and mutation. Two-point crossover was implemented using cut points allowed between bits. Mutation was applied on individual bits.

4 Fault Recovery Of Quadrature Decoder

The quadrature decoder [2] was selected as an initial case study for testing and refinement of our evolutionary recovery strategy. It represents a NASA application of manageable size that is appropriate for tuning of the GA. Quadrature decoders provide a means of counting objects passed back and forth through two beams of light, or alternatively determining the angular displacement and direction of rotation of an encoder wheel turning about its axis. A quadrature decoder that determines the direction of rotation of a shaft is shown in Figure 3.

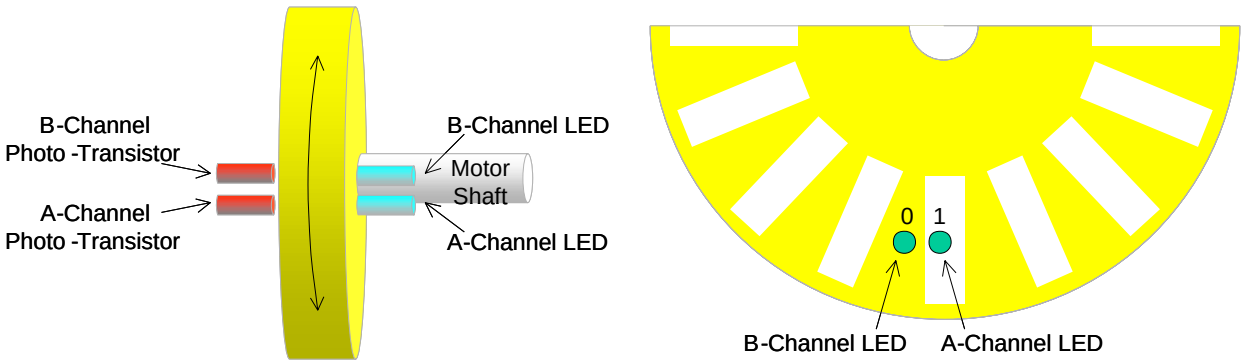
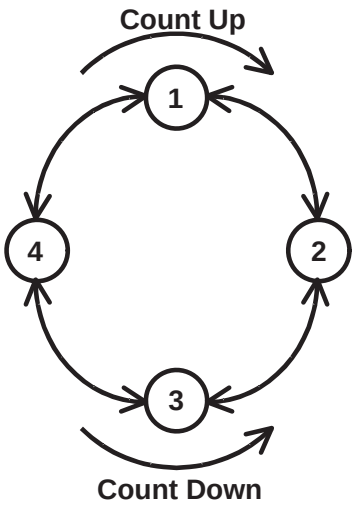


Fig. 3. Rotating shaft application for a quadrature decoder.

The concept of operation for the quadrature decoder is that the objects, or opaque arcs on the rotating wheel, to be counted will first obscure and then move past the two light beams in succession. The order in which the beams are cleared can be used to ascertain the direction of rotation. The use of two beams acts to preclude false counts due to jitter or bounce resulting from multiple phantom reads. For example, to have a valid increment in the rotational count, both beams must be cleared in succession.

To implement the encoder, it is possible to employ a state machine that keeps track of the beam activity. The state machine accepts two single-bit inputs which are asserted only when the corresponding sensor is obscured. When a change of the inputs occurs, the state machine transitions to its next internal state. The state machine outputs a single bit to indicate if the wheel is continuing to rotate in the same direction or if it has changed direction of rotation. The finite state machine for the quadrature decoder is shown in Figure 4.



Ch A	Ch B	State
1	0	1
1	1	2
0	1	3
0	0	4

Fig. 4. Quadrature decoder finite state machine.

5 Experimental Setup and Results

The software system used is depicted in Figure 5. The entire system is implemented in software. The GA software is ECJ, a Java-based evolutionary computation and genetic programming system by Sean Luke of George Mason University. ECJ is augmented by our code for tasks like decoding individuals and calculating fitness. The GA sits on top of Xilinx Corporation's JBits software [5, 8], a set of Java classes which provide

an Application Programming Interface to access the Xilinx FPGA bitstream. Xilinx's Virtex DS software, which simulates the operation of Virtex devices, is used to test candidate solutions. Borland's JBuilder Java environment is used for development and to run the system, though Sun Microsystem's Java virtual machine is used beneath JBuilder.

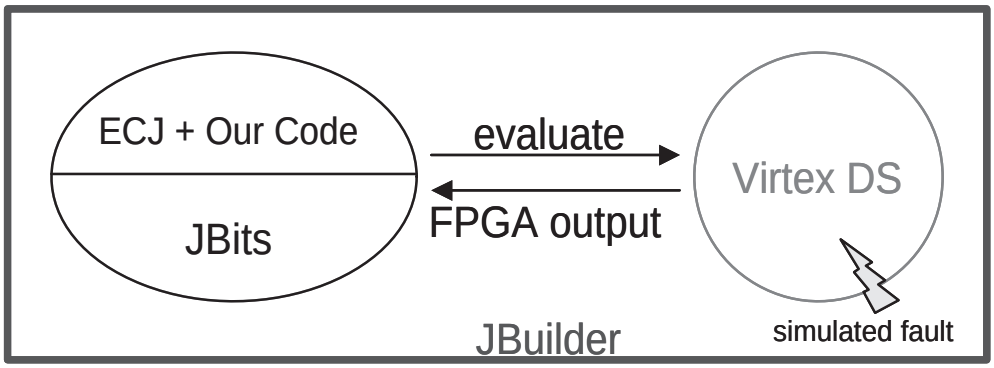


Fig. 5. Software system.

To evaluate the fitness of an individual, an input stream of 500 bit pairs is used. These inputs attempt to fully exercise the evolving finite state machines. The output stream consists of 510 bits sampled across all four CLBs. Ten bits are added to allow for delays in the evolved FSMs. This gives ten output stream windows of length 500, with each output stream shifted by 1-bit from the next. Sampling across all the CLBs allows the GA to maximum flexibility in building the FSM. Thus, fitness is expressed as:

$$F = \max_{i=1,4;j=0,9}(\mathrm{CLB}_i^j)$$

where CLB_i^j represents the number of correct output bits from the ith CLB shifted by j clock ticks. The fitness is simply the highest number of correct output bits seen across all of the CLBs and across the ten output windows. The best score is 500, and the worst score is 0.

The genetic algorithm was set up as shown in Table 1. Small population sizes were necessary since a memory leak was present in one of the modules.

Number of generations	1000
Population size	40
Tournament Size	4
Elitist Individuals	2
Gen 0 Seeding	20 individuals
Crossover rate	0.8
Mutation rate	0.002 per bit

Table 1. GA parameters.

Approximately 10 experimental runs were conducted using smaller input bitstreams of 100 bit pairs. These were found to evolve finite state machines that were tuned to the test cases, but not robust when interrogated with out of sample input test streams. Two runs were conducted using 500 bit pairs and one these runs was able to evolve a 100% accurate quadrature decoder finite state machine in the presence of an induced fault. The best evolved configuration was found in generation 623 and is shown in Figure 6. Two of the 16 LUTs went unused which is not surprising given that the FSM can be implemented with about 10 LUTs. The GA exploits the induced fault to its advantage because if you remove the fault in the evolved

solution, it no longer functions correctly – it achieves an accuracy of only 93.8%. Also, note that the input LUTs had mostly zeros in their tables. This is because we fix most of those bits to zero in the genome since they do not affect the LUT's function. However, the "corner" bits of each of those input LUTs are involved in processing the input, and therefore, are evolved.

The GA performance curve for this run is shown in Figure 7. The run ramps up quickly showing that useful search is underway, however, the average fitness is stagnant for about 300 generations, which is not encouraging. The runs are quite slow to execute on a 2 GHz Pentium 4 PC. Runtimes were about 45 hours since each evaluation takes approximately 6 seconds.

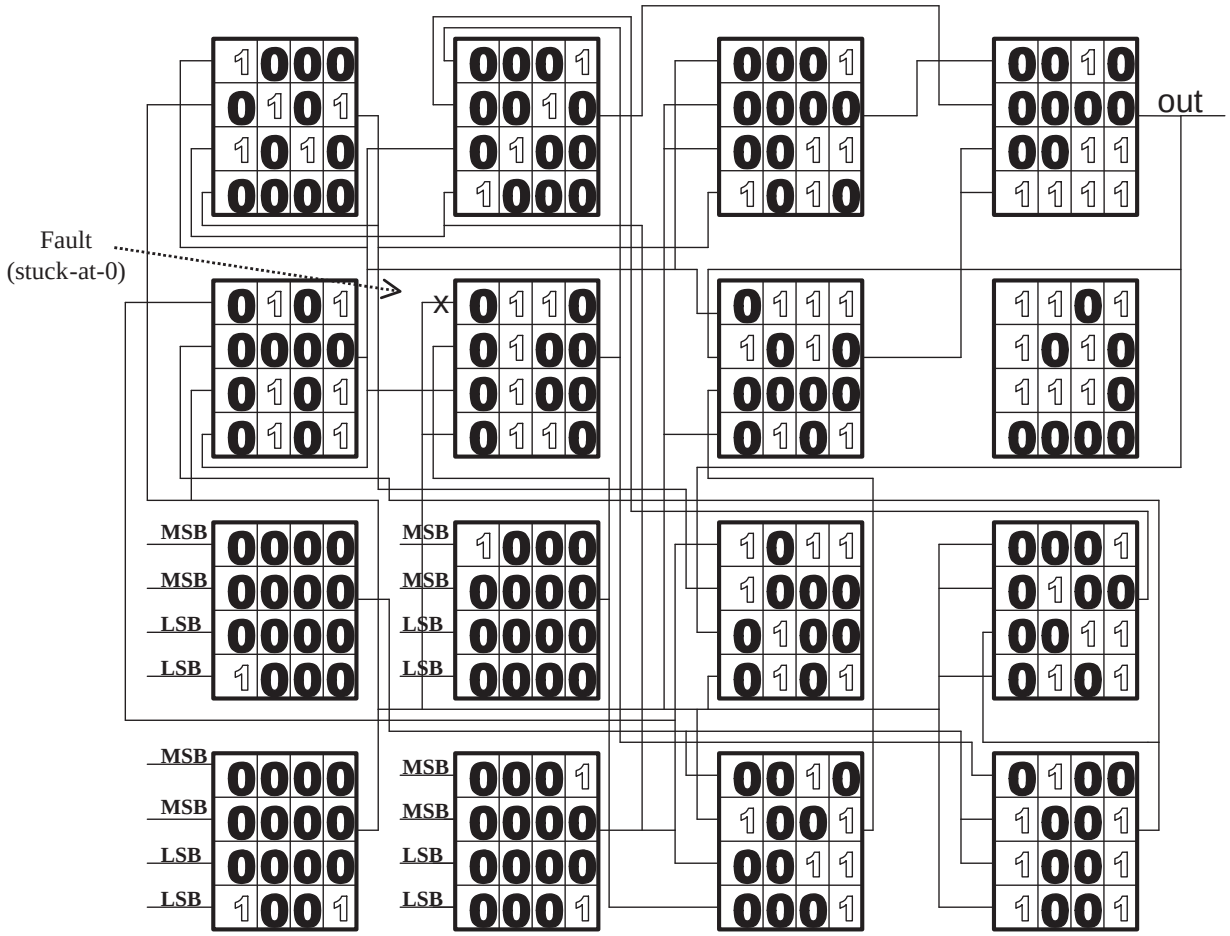


Fig. 6. Evolved configuration showing routing, LUT contents, and simulated fault. Inputs are on the lines labeled MSB and LSB, referring to the least/most significant bit of the input. Wires that are shown crossing perpendicularly (eg, +) are unconnected – only wires that have ⊤ junctions are connected.

At the time of this writing, we have started running similar experiments using the actual hardware FPGA in the evolutionary loop. The initial runs look promising. We have been able to evolve perfect quadrature decoders in the presence of 20 injected stuck-at faults in less than 5 minutes on a board clocked at 1 MHz. Even more encouraging is that we may be able to complete an entire evolutionary run in less than a minute. Our current system spends 90% of its time in software routing operations which we believe can be drastically reduced. On the hardware side, we have the ability to clock our board at much higher clock frequencies.

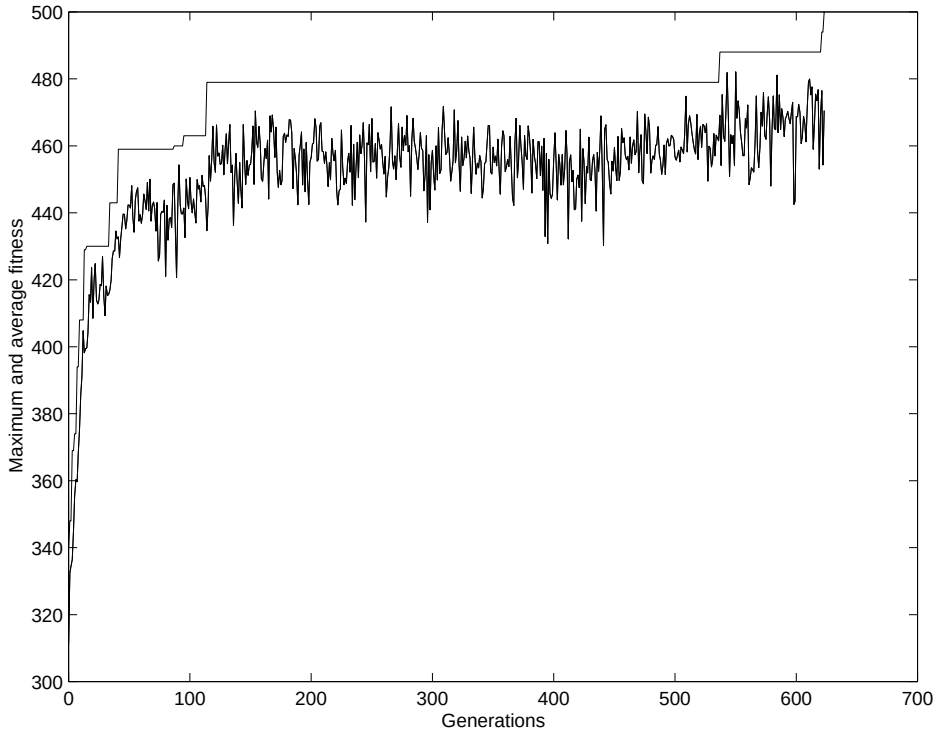


Fig. 7. GA performance curve. The top curve is the best individual's fitness at each generation and the bottom curve is the average fitness.

6 Discussion

Evolutionary systems for fault recovery on FPGAs may be an important tool in the quest for ever-higher levels of fault tolerance in NASA missions and other applications. We have demonstrated a system that is able to evolve a realistic spacecraft control function in the presence of a fault. Using a software simulation of an FPGA, we constructed a genetic representation that included both logic and routing information, and ran a genetic algorithm to evolve a quadrature decoder. As is typical in evolutionary algorithm applications, the evolved solution exploits its resources in unexpected ways. In our case, the algorithm made use of the fault itself in constructing its solution. If there is economy to be gained by exploiting damaged resources, that is certainly a benefit largely unique to evolutionary search.

Potential advantages of this approach are handling a wider range of errors, and relaxing the requirement of fault location/isolation. An autonomous fault recovery system would be possible if the evolution could be done at sub-second speeds. Future work includes investigation of scalability to more complex logic functions and systems that have multiple induced faults. Speeding up the evaluation cycle by doing evolution directly in hardware is our next line of research.

7 Acknowledgments

The authors would like to thank David Gwaltney of NASA Marshall Space Flight Center for suggesting the quad decoder application, and Delon Levi of Xilinx, Inc. for many helpful discussions. The research described in this paper was performed at NASA Ames Research Center, and was sponsored by NASA's Computing, Information, Communications, and Technology Program.

References

1. Actel Corporation, "Actel FPGAs Make Significant Contribution To Global Space Exploration," Press Release, August 30, 1999. available at: http://www.actel.com/company/press/1999pr/SpaceContribution.html

2. Agilent Technologies, Inc., Quadrature Decoder/Counter Interface ICs, Data Sheet HCTL-2020PLC.
3. N. W. Bergmann and P. R. Sutton, "A High-Performance Computing Module for a Low Earth Orbit Satellite using Reconfigurable Logic," in Proceedings of Military and Aerospace Applications of Programmable Devices and Technologies Conference, September 15-16, 1998, Greenbelt, MD.
4. R. O. Canham and A. M. Tyrrell, "Evolved Fault Tolerance in Evolvable Hardware," in Proceedings of IEEE Congress on Evolutionary Computation, 2002, Honolulu, HI.
5. S. Guccione, D. Levi, P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).
6. P. Haddow and G. Tufte, "Bridging the Genotype-Phenotype Mapping for Digital FPGAs," The Third NASA/Dod Workshop on Evolvable Hardware, pp. 109-115
7. D. Keymeulen, A. Stoica, R. Zebulum, "Fault-Tolerant Evolvable Hardware using Field Programmable Transistor Arrays," IEEE Transactions on Reliability, Special Issue on Fault-Tolerant VLSI Systems, Vol. 49, No. 3, September 2000, pp. 305-316.
8. D. Levi and S. Guccione, "GeneticFPGA: Evolving Stable Circuits on Mainstream FPGAs," In Adrian Stoica, Didier Keymeulen, and Jason Lohn, editors, Proceedings of the First NASA/DOD Workshop on Evolvable Hardware, pp. 12-17, IEEE Computer Society Press, Los Alamitos, CA, July 1999.
9. J.D. Lohn, G.L. Haith, S.P. Colombano, D. Stassinopoulos, "A Comparison of Dynamic Fitness Schedules for Evolutionary Design of Amplifiers," in Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, IEEE Computer Society Press, 1999, pp. 87-92.
10. D.C. Mayer, R. B. Katz, J. V. Osborn, J. M. Soden, "Report of the Odyssey FPGA Independent Assessment Team," NASA/JPL, 2001.
11. J. F. Miller and M. Hartmann, "Evolving messy gates for fault tolerance: some preliminary findings," in Proceedings of the Third NASA/DoD Workshop on Evolvable Hardware, July 12-14, 2001, Long Beach, CA.
12. M. Tahoori, S. Mitra, S. Toutounchi, E. McCluskey, "Fault Grading FPGA Interconnect Test Configuration," in Proceedings of Intl Test Conference, 2002.
13. A. Thompson, "Evolving Fault Tolerant Systems," in Proceedings of 1st IEE/IEEE Intl Conference on Genetic Algorithms in Engineering Systems, IEE Conf. Pub. No 414, pp 524-529, TBD Date, TBD Place.
14. S. Vigander, Evolutionary Fault Repair of Electronics in Space Applications, Dissertation, Norwegian University of Science and Technology, Trondheim, Norway, February 28, 2001.
15. E. B. Wells and S. M. Loo, "On the Use of Distributed Reconfigurable Hardware in Launch Control Avionics," in Proceedings of Digital Avionics Systems Conference, TBD day/month, 2001, TBD location.
16. Xilinx Inc., "Xilinx Radiation Hardened Virtex FPGAs Shipping To JPL Mars Mission And Other Space Programs," Press Release, May 15, 2001.