https://ntrs.nasa.gov/search.jsp?R=20060021469 2019-08-29T21:59:27+00:00Z

Source of Acquisition
NASA Ames Research Center

Evolutionary Based Techniques for Fault Tolerant Field Programmable Gate Arrays

Gregory V. Larchev
University Affiliated Research Center, UC Santa Cruz
NASA Ames Research Center
Moffett Field, CA
glarchev@mail.arc.nasa.gov

Jason D. Lohn
NASA Ames Research Center
Moffett Field, CA
jlohn@email.arc.nasa.gov

Abstract

The use of SRAM-based Field Programmable Gate Arrays (FPGAs) is becoming more and more prevalent in space applications. Commercial-grade FPGAs are potentially susceptible to permanently debilitating Single-Event Latchups (SELs). Repair methods based on Evolutionary Algorithms may be applied to FPGA circuits to enable successful fault recovery. This paper presents the experimental results of applying such methods to repair four commonly used circuits (quadrature decoder, 3-by-3-bit multiplier, 3-by-3-bit adder, 4-to-7 decoder) into which a number of simulated faults has been introduced. The results suggest that evolutionary repair techniques can improve the process of fault recovery when used instead of, or as a supplement to Triple Modular Redundancy (TMR), which is currently the predominant method for mitigating FPGA faults.

1. Introduction

FPGAs have a number of advantages which make them particularly suitable for space applications. These advantages have been noted in both recent research publications [3], [13] and manufacturers' literature [1], [14]. The benefits of FPGAs include reconfiguration capability to support multiple missions, the potential to accommodate on-chip and off-chip failures, and the ability to correct latent design errors after launch.

For some FPGA applications such as Reusable Launch Vehicles (RLVs), comparatively short mission durations and low levels of ionizing radiation are involved. In these cases, conventional 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, prompting the need for adequate fault coverage. Since the number of programming cycles in SRAM-based devices is unlimited, new techniques become feasible for active recovery through reconfiguration of a compromised FPGA.

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.

Autonomous repair can either provide alternative to or supplement redundancy as a means of restoring lost capability. Evolutionary recovery methods attempt to facilitate repair through reuse of damaged parts. The fault repair mechanisms discussed in this paper can improve system reliability regardless of whether redundancy is utilized or not. If a particular circuit has been shown to respond well to evolutionary repair, then evolutionary algorithms can be relied on as a primary source of fault tolerance. This allows the engineers to avoid the increased size, weight, and power consumption traditionally associated with providing redundant spares. In cases when evolutionary

algorithms have difficulties producing fully functional repairs, it is still possible to use these methods alongside traditional redundancy techniques. By repairing each individual triplet of a triple-redundant system, it is possible to improve the performance of each triplet by a large enough margin so that the majority output is 100% correct (even if each individual output is not). Another advantage of the genetic algorithm based methods for the fault repair is that the characteristics of the failure need not be precisely diagnosed in order to restore the lost functionality.

Section 2 of this paper will discuss prior work in the area of FPGA fault tolerance. Section 3 will describe the type and number of faults we are looking to address. Sections 4, 5, 6 and 7 will present the results of applying evolutionary techniques to repair faults in 4 different types of circuits. Section 8 will conclude the paper and discuss future applicable work.

2. Background

Various evolutionary approaches have been previously proposed for FPGA fault recovery. While some apply evolutionary algorithms prior to the occurrence of the fault, others attempt to repair the fault after its incidence. Three recent examples that apply evolutionary algorithms to realize fault-tolerant designs include [5], [4], and [8]. In [5], 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 compared the fault tolerance of oscillators evolved by including a range of fault conditions within the fitness measure during the evolutionary process. In [8], the evolution of designs containing redundant capabilities without the designer having to explicitly specify the redundant parts themselves was investigated.

The second approach involves restoring the functionality of the original circuit after introducing the fault or faults. Some examples of this approach will be presented in the later sections of this paper.

3. Fault description

The overall goal of the project was to examine the performance of the evolutionary fault-repair mechanism for various circuits when a certain percentage of circuit's transistors is affected by faults. Ideally, a number of single-event latchups (SELs) would be replicated in the circuit. Unfortunately, exact simulation of radiation-induced faults is difficult. First, there are few studies available on what type of faults occurs most frequently when a Virtex FPGA is subjected to cosmic radiation. Second, the proprietary nature of the Xilinx Virtex configuration file format makes it difficult to control the behavior of the individual transistors on the FPGA. In our case, the most convenient method of fault simulation is "hard-wiring" the individual LUT values and/or connections by setting the corresponding bits inside our chromosome to 0 or 1. However, simply fixing a certain percentage of chromosomal bits fails to take into account the actual distribution of logic and routing transistors inside the FPGA. It is estimated that on a Virtex FPGA 80% or more of all transistors are dedicated to routing. Thus, a larger number of simulated faults should affect routing rather than logic. We decided to implement a condition where at least 10% of all the LUTs in the circuit produce a constant output (either a 0 or a 1). This setup simulates shorting the output of a LUT to either power or ground. To complement the simulated routing faults, a smaller number of logic faults was also introduced. The final simulation "hard-wired" between 1% and 2% of all the LUT bits to either 0 or 1, thus simulating either a stuck-at-0 or a stuck-at-1 condition inside the LUT.

4. Quadrature decoder

The quadrature decoder was selected as an initial case study for testing and refinement of our evolutionary recovery strategy. 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 (Figure 1).

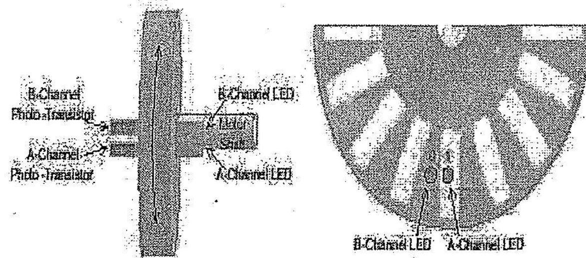

Figure 1. Quadrature decoder

The quadrature decoder circuit is a 4-state state machine, with 2 1-bit inputs and 1 1-bit output. It is implemented using 4 CLBs (or 16 LUTs) on a Virtex FPGA. In order to test the correctness of the circuit, a test vector of 700 bit pairs is fed in. A circuit that

produces all 700 output bits correctly is presumed to be fully functional.

While the total of 16 LUTs are used to implement the quadrature decoder, the circuit inputs are fed into the first 4 LUTs (LUTs 0 through 3). LUTs 4 through 15 can have each of their inputs connected to the output of any other LUT in the circuit. All the outputs are registered, and feedback is allowed. The total length of the chromosome representing the quadrature decoder is 400 bits (Figure 2).

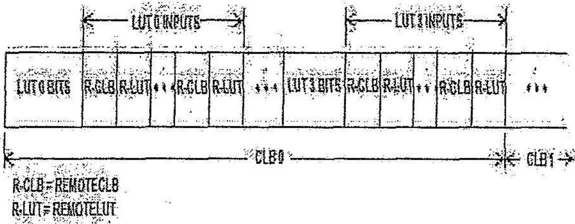

Figure 2. Format of the Quadrature Decoder chromosome

Our first experiment was to evolve a quadrature decoder from scratch (i.e. with all initial circuit configurations randomly generated) on a faultless substrate. 5 runs were performed, with each run limited to 4000 generations (Table 1). The following evolutionary parameters were used for the runs: Population size – 40, Crossover type – two-point, Crossover probability – 80%, Mutation probability – 0.2% per bit, Tournament size – 3, Elitism – 2.

Table 1. Results of quadrature decoder evolution from scratch

	Initial fitness	Initial fitness %	Final fitness	Final fitness %	Number of generations	Indivs. Per generation	Mutation probability
Run 1	483	69	700	100	777	40	0.002
Run 2	468	66.86	700	100	321	40	0.002
Run 3	461	65.86	700	100	3552	40	0.002
Run 4	490	70	683	97.57	3999	40	0.002
Run 5	458	65.43	676	96.57	3999	40	0.002
Avg	472	67.43	691.8	98.83			
Median	468	66.86	700	100			

Several evolutionary runs were performed with just 1 fault in order to test both the fault injection mechanism and the response of the evolutionary algorithm to singular faults. While not enough runs have been performed to obtain statistically significant results, the general observation is that the algorithm has little trouble recovering from just 1 fault of any type.

The fault injection approach for introducing multiple faults was described in Section 3. There are 16 LUTs used in the evolution of the circuit. 2 of those LUTs would be affected by simulated routing faults, making them output either a constant 0 or a 1. Such setup roughly simulates a scenario in which 12.5% of the circuit area is damaged by the routing faults. In addition, 5 logic bits (out of 256 total or ~2%) would be subject to a stuck-at fault. The location and the type of faults (stuck-at-0 or stuck-at-1) were determined randomly for each run (for both logic and routing). Each run took place for at least 10K generations; all runs either successfully completed or were terminated before they reached 20K. The first 3 runs were performed with a population of 60 individuals; the later 7 runs had a population of 100 individuals.

The generation 0 of each run was seeded with 20 "hand-designed" quadrature decoder individuals, each of them fully functional when no faults were present. The rest of the "generation 0" individuals were randomly created. The only differences between the runs were the location and the type of faults.

There were 10 runs performed. Out of the 10, 6 runs have produced a complete repair (resulting in a 60% repair rate). For the runs which achieved complete repair, average number of evaluations required was 66053 (or 660 generations with a population of 100). The average starting fitness was 76.7%, and the average ending fitness was 99.5%. Average improvement in fitness per 1000 evaluations was 2.12%. For the runs where complete repair was achieved, this improvement was 3.52%. For the runs where complete repair was not achieved, the improvement was 0.02% (Table 2).

5. 3-by-3-bit Multiplier

The 3-by-3-bit multiplier was the first combinational circuit tested with our algorithm. Our motivation for evolving this circuit stems from its importance in Digital Signal Processing (DSP), where multipliers are used to construct Finite Impulse Response (FIR) filters.

Table 2. Results of quadrature decoder evolution with faults

	Starting best fitness	Starting best fitness %	Ending best fitness	Ending best fitness %	Improvement	Improv %	Number of gens	Number of evals. per gen	Number of evals.	% Improvement per 1000 evaluations
Run 1	666	95.1429	700	100	34	4.857143	37	60	2220	2.18790219
Run 2	462	66	700	100	238	34	375	60	22500	1.51111111
Run 3	568	81.1429	675	96.429	107	15.28571	10814	60	648840	0.02355853
Run 4	503	71.8571	700	100	197	28.14286	3489	100	348900	0.08066167
Run 5	646	92.2857	700	100	54	7.714286	10	100	1000	7.71428571
Run 6	445	63.5714	696	99.429	251	35.85714	18256	100	1825600	0.01964129
Run 7	472	67.4286	699	99.857	227	32.42857	13581	100	1358100	0.0238779
Run 8	654	93.4286	700	100	46	6.571429	8	100	800	8.21428571
Run 9	496	70.8571	700	100	204	29.14286	209	100	20900	1.39439508
Run 10	458	65.4286	694	99.143	236	33.71429	12334	100	1233400	0.02733443
Average	537	76.7143	696.4	99.486	159.4					2.11970536
									Average for complete	3.51710691
Median	499.5		700		200.5					
									Average for incomplete	0.02360304

Other groups in the evolvable hardware community have also recognized the importance of both FIR filters and their components. A hardware architecture necessary to efficiently evolve FIR filter coefficients was laid out and simulated in [12]. In [9], Thomson evolved Verilog netlists of several FIR Primitive Operator Filters (FIR filters which are constructed without the use of multipliers). In [11], Vassilev used evolutionary algorithms to produce circuit configurations for multipliers of various sizes (the largest one being 4-by-4-bit) which use the minimal possible number of logic gates. Torresen [10] evolved a 5-by-5-bit multiplier in simulation, by using data partitioning to decompose the problem into smaller "training sets" and evolving each output bit individually.

The 3-bit multiplier is a combinational circuit. It has 6 inputs (3 bits for each of the 2 multiplicands) and 6 outputs (for the 6-bit product). Since the circuit is fully combinational, all the LUT outputs are unregistered. Therefore, feedback is not allowed, in order to prevent metastable conditions within the circuit. Several stages are used; the inputs into a LUT at any given stage can only come from the outputs of LUTs from the earlier stages. The initial template used to evolve the multiplier is shown in Figure 3.

The circuit can use up to 12 CLBs, or 48 LUTs. The gene for each LUT is 40 bits long. Out of those 40 bits, 16 bits implement the logic of the LUT, while 24 bits specify the connections of the LUT inputs. The algorithm has a provision to enforce the rule that inputs into any given LUT may only come from the LUTs in previous stages. Aside from the number of bits devoted to routing, and the method to enforce the stage rule (which does not affect the chromosome itself, only the way it is interpreted), the overall chromosomal structure for the multiplier is the same as that for the quadrature decoder (Figure 2). The complete chromosome is 1536 bits long. A multiplier circuit has to be able to produce 384 bits correctly in order to be 100% functional. Therefore, the fitness of each individual in the population ranges from 0 to 384.

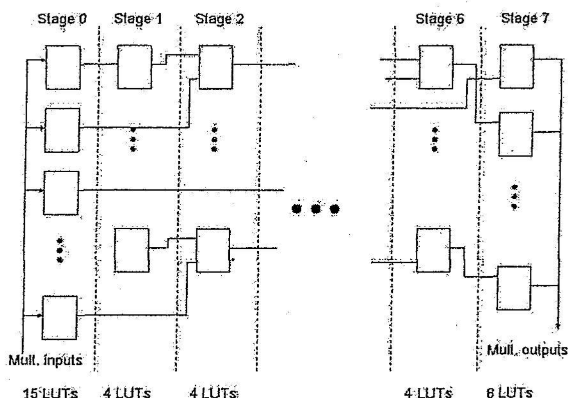

Figure 3. Initial template for multiplier evolution

Before testing the fault-repair capabilities of our evolutionary algorithm on the 3-bit multiplier, we wanted to evolve one from scratch on a faultless substrate. The evolved circuit was later used to seed the fault-containing runs (Figure 4). The evolution took 114248 generations with a population of 200 individuals per generation. Other evolutionary parameters were as following: crossover type - two-point, crossover probability - 80%, mutation

probability - 0.1% per bit, tournament size - 3, number of elite individuals - 2.

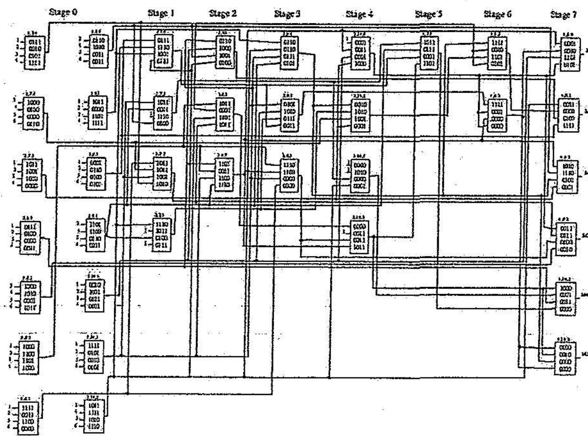

Figure 4. 3-by-3-bit multiplier evolved from scratch

In order to satisfy the conditions proposed in Section 3, 5 LUTs (10.4% of the total circuit area) and 12 additional LUT logic bits (1.6% of the total number) were affected by simulated stuck-at faults. 17 runs were conducted, each run varying in length and/or initial seed numbers for the randomly determined functions (Table 3).

Of all the runs performed, none has achieved 100% repair. The average improvement in correctness over the course of the run was 12.5% (from 83.3% to 95.8%). Notably, most of the improvement occurs early in the evolutionary process. The average improvement after 1000 generations was 10.4% (from 83.3% to 93.7%).

6. 3-by-3 bit adder

Along with multipliers, adders are important building blocks of digital circuits. Adders are even more essential to the FIR filter design than multipliers (since it is possible to implement an FIR filter without the use of multipliers, but not without the use of adders).

Because of their usefulness, simplicity, and scalability, adders have been frequently utilized as sample problems for testing various evolutionary algorithms. In [6], Miller demonstrated several 2-bit adders evolved in simulation from 2-input logic gates and 2-input MUXes. In [2], Bentley continued Miller's work by applying constraints to the use of some FPGA resources, while allowing his genetic algorithm to make more extensive use of others. And in [7], Shanthi explored the evolution of fault-tolerant 2-bit adders, implemented by utilizing empty resources available on the FPGA.

Table 3. Results of multiplier evolution in the presence of faults

Run #	Initial fitness	Initial fitness %	Final fitness	Final fitness %	Improv %	Number of gens	Individs. Per gen	Evolution seed	Fault seed	Mutation probab. (0.1% by default)	Fit after 1000 gens	Fit. After 1000 gen %	Improv after 1000 gen %
1	332	86.46	365	95.05	8.594	13256	100	4371	4371		359	93.49	7.031
2	332	86.46	362	94.27	7.813	2709	100	4372	4371		361	94.01	7.552
3	332	86.46	364	94.79	8.333	1928	100	4373	4371		362	94.27	7.813
4	332	86.46	366	95.31	8.854	9892	100	4374	4371		360	93.75	7.292
5	332	86.46	365	95.05	8.594	1942	100	4375	4371		365	95.05	8.594
6	320	83.33	379	98.7	15.36	10883	100	4371	4372		371	96.61	13.28
7	320	83.33	378	98.44	15.1	26561	100	4372	4372		370	96.35	13.02
8	320	83.33	377	98.18	14.84	12006	100	4373	4372		363	94.53	11.2
9	320	83.33	380	98.96	15.63	21529	100	4374	4372		373	97.14	13.8
10	320	83.33	374	97.4	14.06	12361	100	4375	4372		366	95.31	11.98
11	314	81.77	361	94.01	12.24	11477	100	4371	4373	M: 0.2%	348	90.63	8.854
12	314	81.77	364	94.79	13.02	16940	100	4371	4373		348	90.63	8.854
13	314	81.77	361	94.01	12.24	11354	100	4372	4373		351	91.41	9.635
14	314	81.77	361	94.01	12.24	10536	100	4373	4373		348	90.63	8.854
15	314	81.77	359	93.49	11.72	17235	100	4374	4373		351	91.41	9.635
16	287	74.74	360	93.75	19.01	22425	100	4371	4380		353	91.93	17.19
17	320	83.33	380	98.96	15.63	222056	100	4377	4372		369	96.09	12.76
Avg	319.8	83.29	368	95.83	12.55						360	93.72	10.43

The 3-by-3-bit adder is a combinational circuit which has 6 inputs (3 for each number to be added) and 4 outputs (the sum can be up to 4 bits long.) The empty template for evolving an adder is exactly the same as that for evolving a multiplier (Figure 3). The adder, however, is a simpler circuit to evolve: it is comprised of fewer gates, and the outputs have 4 bits (not 6). Since there are 64 possible input-output combinations, and each output is 4 bits long, the fitness of any adder circuit can range from 0 to 256.

Like the multiplier experiment, our first goal was to evolve an adder from scratch on a faultless substrate. Again, the evolved circuit was later used in the runs into which the faults were introduced. Out of the 6 runs performed, 1 run has evolved a 100% functional adder in 3487 generations (Figure 5, Table 4).

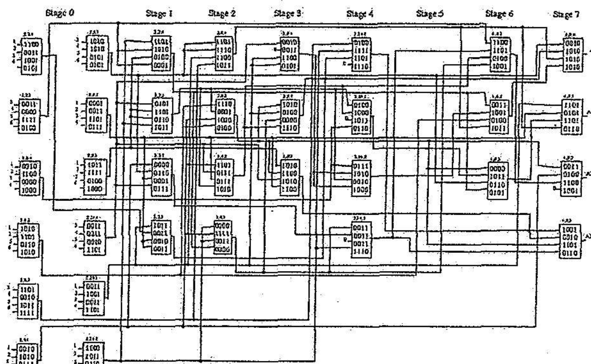

Figure 5. 3-by-3-bit adder evolved from scratch

Table 4. Results of 3-by-3-bit adder evolution from scratch

	Initial fitness	Initial fitness %	Final fitness	Final fitness %	Number of generations	Indivs. Per generation
Run 1	150	58.59	251	98.05	10530	200
Run 2	152	59.38	252	98.44	27159	100
Run 3	148	57.81	253	98.83	18276	200
Run 4	150	58.59	250	97.66	10918	100
Run 5	146	57.03	256	100	3487	100
Run 6	150	58.59	253	98.83	9750	100
Avg	149.33	58.33	252.5	98.63		
Median	150	58.59	252.5	98.63		

The evolutionary parameters for the successful run were the following: Crossover type: two-point, Crossover probability: 80% per individual, Mutation probability: 0.1% per bit, Tournament size: 3, Elitism: 2.

The same number of faults was injected into the 3-bit adder as into the 3-bit multiplier (5 LUT outputs and 12 LUT logic bits were subject to a stuck-at fault.) 10 runs were performed in order to attempt to recover the lost functionality. Out of the 10 runs, 2 have evolved a 100% correct circuit, resulting in a 20% repair rate (Table 5). All the runs were terminated between 10K and 17K generations. The evolutionary parameters were exactly the same as those used when the adder was evolved from scratch.

7. 4-to-7 decoder

The decoder circuit has 4 inputs and 7 outputs; it is used to control the individual segments of the 7-segment LED display. The design is fairly simple; it requires fewer gates to implement than either a 3-bit adder or a 3-bit multiplier. The inputs are the bits for a number between 0 and 15; the outputs indicate whether a particular segment should be turned on or off. Usually 7-segment displays can only show numbers between 0 and 9; however, our circuit incorporates numbers 10 through 15 as well.

The template for the evolution of the circuit is similar to that for the 3-bit multiplier and the 3-bit adder. There are 16 possible input combinations, each resulting in a 7-bit long output. Therefore, the fitness of each circuit can range from 0 to 112.

The 4-to-7 decoder is a smaller circuit than either the 3-bit multiplier or the 3-bit adder, with fewer gates required to implement it. In addition, the maximum fitness of the decoder is smaller than that of the multiplier or the adder. Hence, the decoder is considerably easier to evolve. Only one evolutionary run was conducted to evolve the decoder from scratch. The best random individual in generation 0 had a fitness of 68 (60.7%), and a perfect individual was achieved in 119 generations. The evolutionary parameters were the same as those used for the faultless evolution of the 3-bit adder.

The number and the type of faults introduced into the 4-to-7 decoder are the same as those for the multiplier and the adder (5 LUT outputs and 12 LUT logic bits subject to a stuck-at fault). Just like in previous experiments, the generation 0 of each run contains 20 previously-evolved decoder individuals (while the rest of the population is randomly generated). 10 different runs were performed (the difference between the runs was in the location and the type of faults.) 9 runs have produced a fully functional decoder (90% repair rate). The 10th run was stopped after approximately 10K generations (Table 6). The evolutionary parameters were the same as those used for the faultless run.

Table 5. Results of 3-by-3-bit adder evolution in the presence of faults

	Initial fitness	Initial fitness %	Final Fitness	Final fitness %	Improv. %	Num. of gens	Fitness after 1000 gens	Fitness after 1000 gens %	Improv. After 1000 gens %	% improv per 10000 evals	% improv per 10000 evals for the first 1000 gens
Run 1	180	70.31	224	87.5	17.188	15358	217	84.766	14.453	0.111912	1.4453125
Run 2	166	64.84	222	86.719	21.875	11557	215	83.984	19.141	0.189279	1.9140625
Run 3	191	74.61	253	98.828	24.219	10618	245	95.703	21.094	0.228091	2.109375
Run 4	161	62.89	223	87.109	24.219	16385	222	86.719	23.828	0.14781	2.3828125
Run 5	192	75	254	99.219	24.219	12141	247	96.484	21.484	0.199479	2.1484375
Run 6	197	76.95	251	98.047	21.094	11995	216	84.375	7.4219	0.175855	0.7421875
Run 7	195	76.17	224	87.5	11.328	11741	222	86.719	10.547	0.096483	1.0546875
Run 8	184	71.88	253	98.828	26.953	10274	241	94.141	22.266	0.262343	2.2265625
Run 9	211	82.42	256	100	17.578	16379	250	97.656	15.234	0.107321	1.5234375
Run 10	202	78.91	256	100	21.094	1786	251	98.047	19.141	1.181061	1.9140625
Avg	187.9	73.4	241.6	94.375	20.977		232.6	90.859	17.461	0.269964	1.74609375
Median	191.5	74.8	252	98.438	21.484		231.5	90.43	19.141	0.182567	1.9140625

Table 6. Results of 4-to-7 decoder evolution in the presence of faults

	Initial fitness	Initial fitness %	Final Fitness	Final fitness %	Improv. %	Num. of gens	Improv. % per 1000 evals
Run 1	92	82.14286	112	100	17.85714	134	1.33262
Run 2	90	80.35714	112	100	19.64286	111	1.76963
Run 3	83	74.10714	112	100	25.89286	451	0.57412
Run 4	94	83.92857	112	100	16.07143	462	0.34787
Run 5	80	71.42857	112	100	28.57143	405	0.70547
Run 6	91	81.25	112	100	18.75	5456	0.03437
Run 7	81	72.32143	103	91.964286	19.64286	10347	0.01898
Run 8	89	79.46429	112	100	20.53571	1102	0.18635
Run 9	88	78.57143	112	100	21.42857	214	1.00134
Run 10	84	75	112	100	25	753	0.33201
Avg	87.2	77.85714	111.1	99.196429	21.33929	1944	0.63027
Avg excluding run 7						1010	0.6982
Median	88.5	79.01786	112	100	20.08929	456.5	0.46099

8. Conclusion and future work

Several observations can be made about the results of the performed evolutionary runs. First, every single run produced an improvement in circuit performance. Most of the improvement took place early in the process. Second, some circuits respond more readily than others to evolutionary fault repair. This responsiveness is usually correlated with the size and the complexity of the circuit. The repair time and probability also depend on the number and location of faults.

Based on the observed results, multiple fault-repair strategies for the actual space-bound circuits can be proposed. For each mission, the fault-susceptible circuits can be analyzed prior to the launch. Each of those circuits would be subjected to a number of faults it might be expected to experience over the course of the mission. After introducing the faults into the circuit, an evolutionary fault-repair algorithm would be applied. The experimental results would indicate the probability that the functionality of the circuit could be fully restored in the allotted amount of time. Here, the availability requirements of the circuit (i.e., maximum allowed downtime) would need to be taken into account as well. Based on the testing results, there are two possibilities for incorporating the evolutionary algorithm into the mission. If it is determined that the genetic algorithm is likely to fully recover the functionality of the damaged circuit, then the evolutionary methods can be relied on as the only source of fault tolerance. In this case, size and complexity savings can be realized from eschewing redundant modules. The pre-launch testing might also determine that, for a specific circuit, the genetic algorithm would be unlikely to restore full functionality with the anticipated number of faults in the allowed amount of time. In this case, it would still be possible to utilize evolutionary fault repair alongside the TMR. It has been observed that evolutionary fault repair is successful in restoring the functionality of the tested circuits to above 80% or 90% in virtually every case. Such repair might be sufficient if TMR is also used; because of the majority vote, the overall system might produce fully correct output even if each individual triplet does not.

Just like TMR, evolutionary fault repair system would add size and complexity to the spacecraft electronics. However, the increase in circuitry would be constant relative to the overall number of electronic components (unlike TMR, where the increase is linear relative to the overall circuit area). A single GA-specific processor could potentially perform evolutionary repair for every single FPGA-based

circuit aboard the spacecraft (more than one processor might be required in case the original processor would ever need to undergo repair itself, but the amount of overhead is still constant.)

The main benefit of our algorithm is the fact that it tests evolved solutions on the physical FPGA, as opposed to software simulation. This enables the algorithm to take into account the physical features of the device (faults would fall under that category), and relaxes the requirement of fault location and isolation. Certainly every fault-mitigation approach needs to utilize physical FPGA at some stage of development. Currently our algorithm recovers the circuit functionality from the simulated faults. A logical continuation of this work would include testing the algorithm with physical faults (possibly by subjecting the FPGA to radiation from a particle accelerator).

There are other potential areas for future work. Plans call for utilizing more advanced, generative genetic algorithms in order to increase the performance of the evolutionary search. Other goals include testing the algorithm on the actual mission-ready circuits.

9. 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] Peter J. Bentley, Timothy G.W. Gordon, Jungwon Kim, and Sanjeev Kumar, "New Trends in Evolutionary Computation." In Proceedings of CEC 2001, the Congress on Evolutionary Computation, Seoul, Korea, May 27-30, 2001 IEEE Press. RN/00/68

[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] 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.

[6] J. F. Miller, P. Thomson, and T. Fogarty, "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study", chapter 6, in Genetic Algorithms and Evolution Stategies in Engineering and Computer Science: Recent Advancements and Industrial Applications, published by Wiley, 1997 (November).

[7] A.P.Shanthi, Balaji Vijayan, Manivel Rajendran, Senthilkumar Veluswami and Ranjani Parthasarathi, "Automatic GA Based Evolution of Fault Tolerant Digital Circuits", In Lipo Wang, et al., editor, 4th Asia – Pacific Conference on Simulated Evolution and Learning (SEAL 02) vol. 2, Nov. 2002,pp. 845-849.

[8] A. Thompson, "Evolving Fault Tolerant Systems," in Proceedings of 1st IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems, IEE Conf. Pub. No 414, pp 524-529.

[9] R.Thomson and T.Arslan, "Evolvable Hardware for the Generation of Sequential Filter Circuits," in Proceedings of The 2002 NASA/DOD Conference on Evolvable Hardware, July 15-18, 2002, Alexandria, VA.

[10] J.Torresen, "Evolving Multiplier Circuits by Training Set and Training Vector Partitioning," in Proceedings of Evolvable Systems: From Biology to Hardware, 5th International Conference, ICES 2003, March 2003, Trondheim, Norway.

[11] V.K.Vassilev, D.Job and J.F.Miller, "Towards the Automatic Design of More Efficient Digital Circuits," in Proceedings of The Second NASA/DOD Workshop on Evolvable Hardware, July 13-15, 2000, Palo Alto, CA.

[12] K.A.Vinger and J.Torresen, "Implementing Evolution of FIR-Filters Efficiently in an FPGA," in Proceedings of The 2003 NASA/DOD Conference on Evolvable Hardware, July 9-11, 2003, Chicago, IL.

[13] 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, October 14-18, 2001, Daytona Beach, FL.

[14] Xilinx Inc., "Xilinx Radiation Hardened Virtex FPGAs Shipping To JPL Mars Mission And Other Space Programs," Press Release, May 15, 2001.