

Evolutionary Strategies and Intrinsic Fault Tolerance

A.M. Tyrrell, G. Hollingworth and S.L. Smith
Bio-inspired Architectures Lab, Department of Electronics,
University of York YO10 5DD, UK
Andy.Tyrrell@bioinspired.com

Abstract

Redundancy is a critical component to the design of fault tolerant systems; both hardware and software. This paper explores the possibilities of using evolutionary techniques to first produce a processing system that will perform a required function, and then consider its applicability for producing useful redundancy that can be made use of in the presence of faults, ie is it fault tolerant? Results obtained using Evolutionary Strategies to automatically create redundancy as part of the “design” process are given. The experiments are undertaken on a Virtex FPGA with intrinsic evolution taking place. The results show that not only does the evolutionary process produce useful redundancy, it is also possible to reconfigure the system in real-time on the Virtex device.

1 Introduction

Fault tolerance is a technique used in the design and implementation of dependable computing systems. With the increase in complexity of systems complete fault coverage at the testing phase of the design cycle is very difficult to achieve, if not impossible. In addition, environmental effects such as Radio Frequency and Electromagnetic interference and misuse by users can mean that faults will occur in systems. These faults will cause errors and if left untreated, could cause system failure. The role of fault tolerance is to deal with errors, caused by faults, before they lead to failure.

Fault-tolerance is increasingly a crucial part of system designs. More and more applications are using programmable systems for their operation. Many of these applications have part or all of their function classified as critical in one form or another. Since the testing of systems fully is generally unrealistic, critical functions must be protected “on-line”. This is often achieved by using fault-tolerance to cope with errors produced during the operation of the system. The traditional techniques are based around simple static redundancy (such as N-version systems, eg [1]). These are expensive in terms of equipment, time and design

costs. An alternative approach is to use a form of reconfiguration to “bypass” the faulty item, which may be a hardware process, a software process, or a combination of the two.

Fault tolerance in a processing system implies the mapping of a logical array onto a non-faulty physical array. When faults arise, a mechanism must be provided for reconfiguring the physical system such that the logical array can still be represented by the remaining non-faulty processing elements. All reconfiguring mechanisms can be considered to be based on one of two types of redundancy: time redundancy or hardware redundancy [2].

In time redundancy the tasks performed by faulty processors are distributed among its “neighbours”. When reconfiguration occurs, processors dedicate some time to performing their own tasks and some to performing the faulty processors’ functions, resulting in some degradation of the system's performance. In addition, the algorithm being executed must be sufficiently flexible to allow a simple and flexible division of tasks in real-time.

In hardware redundancy physical spare processing elements and links are used to replace the faulty ones. For this process reconfiguring algorithms must optimise the use of spares. In the ideal case, a processing system with N spares must be able to tolerate N faulty processors. However, in practice, limitations on the interconnection capabilities of each cell prevents this goal from being achieved.

The majority of hardware redundancy reconfiguration techniques rely on complex algorithms to re-assign physical resources to the elements of the logical array. In most cases these algorithms are executed by a central controller which also performs diagnosis functions and accomplishes the reconfiguration of the physical system [3,4]. This approach has been demonstrated to be effective, but its centralised nature makes it prone to collapse if the processor in control fails. These mechanisms also rely on the designer making *a priori* decisions on reconfiguration strategies and data/code movement, which are prone to error and may in practice be less than ideal. Furthermore, the timing of signals involved in the global control are often prohibitively

long and therefore, unsuitable for real-time fault tolerance.

An alternative approach is to distribute the diagnosis and reconfiguration algorithms among all the processing elements in the system [5,6]. In this way no central agent is necessary and consequently the reliability and time-response of the system should improve. However, this decentralised approach has tended to increase the complexity of the reconfiguration algorithm and the amount of communications within the network. In addition, considerable work is required on producing redundancy.

Traditionally, fault tolerance has been added explicitly to system designs by including redundant hardware and/or software which will "take over" when an error has been detected. A novel alternative approach would be to design the system in such a way that the redundancy was incorporated implicitly into the hardware and/or software during the design phase [7,8]. This should provide a more holistic approach to the design process. We already know that Genetic Algorithms and Programming can adapt and optimise their behaviour and structure to perform a specific task, but the aim is that they should "learn" to deal with faults within their operation space. This implicit redundancy would make the system response invariant to the occurrence of faults.

The Virtex devices now allow us to perform real-time partial reconfiguration on hardware [9]. This paper explores the possibilities of using evolutionary techniques to first produce a processing system that will perform a required function, and then consider its applicability for producing useful redundancy that can be made use of in the presence of faults, ie is it fault tolerant?

The results shown in this paper are based around a simple example, an evolved oscillator [10]. All evolution was performed on a Virtex 1000.

2 Evolved Fault Tolerance

Faults can be viewed in a system similar to changes in the environment for evolving biological entities. The individuals compete in the new environment to produce a species which is capable of living and reproducing with the new changes. This method of adapting to faults requires a change in the way most evolutionists view evolutionary algorithms.

In general evolutionary algorithms are viewed as an optimising process, where the algorithm is halted when some maximal/average fitness is reached. For evolutionary strategies to be used to provide fault tolerance, the algorithm should be viewed as a continuous real-time adaptation system, constantly adapting to changes in the environment. These can come from a number of sources, for example:

- hardware failures leading to faults;
- requirements/specification changes;
- environmental changes (eg EM interference).

When such a change occurs, the fitness of the individuals will change correspondingly. As the evolution progresses, the individuals will adapt to the new changes until they regain their fitness.

Critical to any fault tolerant system is useful redundancy, whether in the form of virtual hardware/software or real hardware/software and possibly design. One of the features of an evolutionary system is its reliance on having a diverse population from which to evolve from. The hypothesis tested in this paper concerns the applicability of these diverse members to provide useful redundancy in the presence of faults, ie provide inherent fault tolerance [11,12]. In [11], the authors considered Populational Fault Tolerance, where they effected faults into a number of transistor circuits. In this work high degrees of genetic convergence was achieved and the authors speculated on what might be providing redundancy within a given population, and hence allowing fault tolerant properties to exist. The work reported in this paper encourages genetic convergence not to occur and hence considers a related but different set of evolved circuits.

3 JBits

The work reported in this paper involved the intrinsic evolution on Virtex devices. This made considerable use of JBits and a brief overview of JBits is now given. JBits [13] is a set of Java classes which provide an Application Program Interface (API) into the Xilinx Virtex FPGA family bitstream. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. It provides the capability of designing and dynamically modifying circuits in Xilinx Virtex series FPGA devices. The programming model used by JBits is a two dimensional array of Configurable Logic Blocks (CLBs). Each CLB is referenced by a row and column index, and all configurable resources in the selected CLB may be set or probed.

[10] showed an example application programmed using JBits, the requirement was to change the LookUp Table (LUT) entry so that the function of the gate is changed from an OR gate to an AND gate (a simple example). To perform this application a baseline bitstream is created which contains the required input and outputs to the LUT. The LUT is forced to be in a specific CLB on the FPGA so that it can later be located. Once this has been done the Java

Input Vector				OR	AND
G3	G2	G1	G0		
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	0	0	1	0
1	0	0	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

Table 1: Truth Table for AND/OR gates

program as shown in [10] can be written. The program is written to first load in the baseline bitstream, this bitstream is then modified to change the function of the cell (in this case change the OR gate into an AND gate), the LUT entries for the AND and OR gates are calculated by taking the truth table and writing the outputs into a single 16bit (4 nibble) value:

```
OR  1111 1111 1111 1110    0xfffe
AND 1000 0000 0000 0000    0x8000
```

Once the LUT has been changed (within JBits), the bitstream is then retrieved and sent to the Virtex Chip.

Since the JBits interface can modify the bitstream in this way it is still possible to create a 'bad' configuration bitstream; for this reason it is important to make sure that the circuit which is downloaded to the chip is always a valid design. This is done by only ever modifying a valid design to implement the evolvable hardware. This is a method similar to that demonstrated in Delon Levi's work on GeneticFPGA in [14] where the XC4000 series devices were programmed to implement evolvable hardware. The major obstacle in that implementation, the speed with which the devices could be programmed, is no longer a problem here.

The hardware interface is the XHWIF (Xilinx Hardware InterFace). This is a Java interface which uses native methods to implement the platform dependent parts of the interface. It is also possible to run an XHWIF server, which receives configuration

instructions from a remote computer system and configures the local hardware system. This enables the evolvable hardware to be evaluated in parallel on multiple boards. This interface is also part of Xilinx's new Internet Reconfigurable Logic (IRL) methodology, where the computer houses a number of Xilinx parts which can easily be reconfigured through the Java interface.

4 Evolutionary Strategies and Evolvable Hardware (EHW)

The DNA sequence (the genotype) carried by all living things is used, through a process known as embryological development, to build the phenotype (the body), which is a sequence of chemical interactions within each cell that distinguishes it from other cells and describes its action. The body is subject to environmental pressures where its fitness for reproduction is assessed; fitter individuals have a higher rate of reproduction. This means that genes within the DNA that code specific 'good' traits (traits which describe better reproduction abilities) will have a higher probability of existing in future populations. Genetic Algorithms were first explored by John Holland [15]; he showed that it is possible to evolve a set of binary strings which describe a system's characteristics and that a measure of fitness can be applied to the system's output(s). The binary strings are analogous to the DNA, and the implementation to the phenotype.

In artificial evolution, a binary string usually describes the system. This is achieved directly or indirectly through some form of embryological development. The system is then assessed within the environment to determine its fitness relative to the other individuals (eg does one circuit have a better filter characteristic than another?). This measure can then be used to weight the probability of deleting a system from the population, so that through many generations good genes survive and bad genes die out. The chromosomes of different members of a population can have segments exchanged or *crossed over*, and part of a chromosome can be randomly *mutated*. Both of these operations will create a new population to be considered for fitness (and hence, survival). The circuits produced will gradually move towards an optimal condition where the mutations and crossovers have no improved effect on the performance of the circuit. In the majority of situations this evolutionary process is performed in simulation, with only the final fittest circuit being implemented in hardware.

A further extension to the domain of evolutionary techniques came with the creation of Field Programmable Gate Array (FPGA) [16] devices and Programmable Logic Devices (PLD), since these can be programmed using a binary string or by coding a binary logic tree. The electronic circuits can then be evaluated either electronically, to compare their outputs with the required output (intrinsic EHW) [17, 18], or in simulation (extrinsic EHW) to create some measure of the fitness [19].

For simplicity a $16*(1+1)$ Evolutionary Strategy (ES) was used for the majority of experiments reported in this paper. The $(1+1)$ ES simply works by creating a parent, mutating this parent to create a child and comparing the fitness of the parent and the child. The best individual becomes the parent for the next generation. In the $16*(1+1)$ ES this process was replicated 16 times, with little interaction between the $16*(1+1)$ ES, allowing increased diversity.

Further selection pressure is created through the overwriting of consistently bad $(1+1)$ ES's by another random $(1+1)$ ES. The environment is similar to a number of species evolving, where each $(1+1)$ ES is a species (even though there is only ever one individual!). Each species competes for its fitness with fifteen other species; if a species' fitness drops due to a change in the environment (which could be a literal change in the environment or a change in the genotype due to a mutation for example), the species will be deleted (xenocide) and replaced with one of the other species, the choice as to which one is used for replacement is done randomly. Although the number of species would seem to decrease with this method, it in fact does not appear to be the case, since once the

new species has been in existence for a number of generations it will be significantly different from the other species due to mutations.

All experiments were performed on a Xilinx Virtex FPGA system, which allowed partial reconfiguration to be performed using JBits [13]. The application was to evolve a simple 3KHz oscillator on this device [10]. A fitness level of 200 was set to be an acceptable answer. The hardware configuration is shown in figure 1. Rather than checking the actual frequency of the signal by polling the output value and checking to see when it is high and when it is low, the fitness is evaluated by including a counter which counts the number of positive edges within a certain time period. This proved to be a more accurate method of frequency measurement.

The results for three different mutation rates are shown in figures 2-4. The fault model used to test the fault tolerance of the system was one that considered only stuck-at-faults (stuck-at-one - SA1 and stuck-at-zero - SA0) and short circuits. These faults were emulated and inserted into the bitstream of the device. The position and fault type were specified by the setting of particular LUTs within a device. Further faults were introduced into the circuit until, in most cases, the system failed.

5 Results

An important measure when implementing the fault tolerant hardware is the rate of mutation implemented in the $16*(1+1)$ ES. The measure by which the rate of mutation is decided is the rate of change of the genotype, and in a related way the disruption created by the mutation operator. These two measures are orthogonal to each other; the higher the mutation the faster the genotypes can respond to change (faults) but the higher the mutation the more disruption caused to the genotype. With low mutation rates the system becomes less responsive but is more stable once a good fitness has been reached. Three mutation rates are shown in figures 2-4, high mutation level (5 mutations/individual/generation), medium level (2 mutation/individual/generation) and low level (1 mutation/individual/generation).

The most striking feature observed from figures 2-4 is that when faults are injected into the best individual there is another individual in the population that can be identified as the new best with a fitness sufficiently high to be acceptable for use, until a sufficient number of faults are injected and the whole population fails. There does appear to be sufficient redundancy produced by the $16*(1+1)$ ES to provide fault tolerance for these type of faults. A more detailed consideration of the results highlights the different characteristics of the three mutation rates. With low mutation rates the

“jitter” in the fitness is low and when faults occur, which require the circuit to evolve around the fault, the regaining of fitness levels is relatively slow. When mutation rates are high, the “jitter” in the best individual is much more pronounced, but with this mutation rate the fitness of the best individual climbs much more rapidly than with a low mutation rate. The medium mutation rate appears to give the best overall response in terms of both these measures. The “jitter” in the fitness is less pronounced than with a high mutation rate, but when a fault occurs the individuals regain fitness more quickly than for low mutation rates.

Another possible measure of diversity would be to actually look at the “circuit” layout of individuals in a population. The problem is how to identify which matrix elements in the FPGA are providing useful functionality and which are not. An experiment was devised to help provide this information. The genotype is tested by first inserting a Stuck At 0 fault (SA0) and retesting the fitness, inserting a Stuck At 1 fault (SA1) and retesting. Finally, the faults are removed and testing takes place again, just to be sure that the genotype still works even without faults. If the results for the best individual have a fitness which was reduced by at least half, the individual is considered dead and that particular element in the overall matrix is considered critical to the functionality. There is an extra bit (which is not used in the genotype/phenotype map) that describes whether that individual matrix element is used by the circuit. This bit and the corresponding matrices can then be used to build a map of the useful/useless elements within the matrix in the current best solution.

Figures 5, 6, 7 and 8 show the best fitness genotype of the 16*(1+1)ES. The output bit of the circuit is the top right corner, and the results show just how varied the different runs of the 16*(1+1)ES are. Now, as indicated earlier, these figures cannot accurately tell us how much of the circuit is being used because there may be redundant parts to the circuit which mean that although they are being used, the fitness does not decrease to half the maximum value when a fault is inserted. Instead, what can be noticed from these diagrams are the parts of the circuit which are definitely required, ie figures 7 and 8 are more likely to be damaged through faults than figures 5 and 6.

6 Conclusions

The initial results presented in this paper indicate that when individuals within a population are prevented from having genetic convergence the population as a whole does present characteristics that might be very useful in safety critical environments.

The work presented here assumes that an initial evolutionary stage has taken place before the system goes “on-line”. This alleviates to some extent the problems of on-line evolution and the fact that many of the mutants are less than fit. This allows for the on-line system to consist on mostly “useful” individuals. In addition, the evolutionary strategy used means that these are relatively unrelated individuals enhancing the probability of useful diversity. Evolution, once on-line, takes place on all individuals apart from the current “best”. This reduces the effect of low fitness outputs caused by mutation.

It is not the aim of this work to try and test out all possible fault conditions within a particular design, in practice this would never be possible. What is being tested here is the hypothesis that within an evolved population of individuals, there is a high probability that a fault which “kills” one member of the population will not significantly effect the performance of another, useful, member of the population.

It seems clear from the initial experiments that evolutionary strategies can be used to produce redundant versions of systems which increase the fault tolerance of an application. Further work is required to investigate other evolutionary strategies such as Genetic Algorithms, but the results presented here contribute to this exciting and rapidly expanding subject.

Acknowledgements

This work is supported by an EPSRC studentship in the UK and by Xilinx Inc.

References

- [1] Avizienis, A. and Kelly, J.P.J.: 'Fault Tolerance by Design Diversity: Concepts and Experiments', IEEE Computer, August 1984, 17 (8), pp 67-80.
- [2] Chean, M. and Fortes, J. (1990) 'A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays', IEEE Computer, pp 55-69, January.
- [3] Dutt, S. and Mahapatra, N. (1997) 'Node-covering, Error-correcting Codes and Multiprocessors with Very High Average Fault Tolerance', IEEE Transactions on Computers 46 (9), pp 997-1014.
- [4] Fortes, J. and Raghavendra, C. (1985) 'Gracefully Degradable Processor Arrays', IEEE Transactions on Computers, 34 (11), pp 1033-1043.
- [5] D. Mange, M. Sipper, A. Stauffer, G. Tempesti. Towards Robust Integrated Circuits: The Embryonics Approach. *Proceedings of the IEEE*, 88 (4), April 2000, pp. 516-541.
- [6] Ortega, C., Mange, D., Smith, S.L. and Tyrrell, A.M. Embryonics: A Bio-Inspired Cellular Architecture with

Fault-Tolerant Properties, *Journal of Genetic Programming and Evolvable Machines*, 1 (3), July 2000, pp 187-215.

[7] A. Thompson. Evolving Fault Tolerant Systems. Proc. 1st IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA'95, IEE Conf. Publication No. 414, 1995, pp 524-529.

[8] A. Thompson. Evolutionary Techniques for Fault Tolerance. Proc. UKACC Int. Conf. on Control (CONTROL'96), IEE Conference Publication No. 427, 1996, pp 693-698.

[9] Hollingworth, G.S., Smith, S.L. and Tyrrell, A.M. 'The Intrinsic Evolution of Virtex Devices through Internet Reconfigurable Logic', in *Lecture Notes in Computer Science*, Springer-Verlag, April 2000, pp 72-79.

[10] Hollingworth, G.S., Smith, S.L. and Tyrrell, A.M. 'Safe Intrinsic Evolution of Virtex Devices', 2nd NASA/DoD Workshop on Evolvable Hardware, July 2000.

[11] P. Layzell and A. Thompson. Understanding Inherent Qualities of Evolved Circuits: Evolutionary history as a predictor of fault tolerance. Proc. 3rd Int. Conf. on Evolvable Systems (ICES 2000): From biology to hardware, Springer-Verlag, LNCS 1801 Ed. J. Miller and A. Thompson and P. Thomson and T. Fogarty, 2000, pp 133-144.

[12] D. Keymeulen and A. Stoica and R. Zebulum. Fault-tolerant evolvable hardware using field-programmable transistor arrays. *IEEE Transactions on Reliability*, 49 (3), 2000.

[13] Xilinx. Jbits documentation, 1999. Published in JBits 2.0.1 documentation.

[14] D. Levi and S. Guccione. Geneticfpga: Evolving stable circuits on mainstream fpga devices. In *The First NASA/DoD Workshop on Evolvable Hardware*. IEEE Computer Society, 1999.

[15] Holland, J.H.: 'Adaptation in Natural and Artificial Systems', University of Michigan Press, 1975.

[16] Xilinx Inc.: 'XC6200 Field Programmable Gate Array Data Book', <http://www.xilinx.com/partinfo/6200.pdf>, 1995.

[17] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at functional level. In *International conference on Evolutionary Computation: The 4th Conference on Parallel Problem Solving from Nature*, pages 62-71, 1996.

[18] Yao, X. and Higuchi, T.: 'Promises and Challenges of Evolvable Hardware', *International Conference on Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science 1062, Springer Verlag, 1996, pp 55-78.

[19] J. F. Miller. Evolution of digital filters using a gate array model. In Poli et al., editor, *Evolutionary Image Analysis, Signal Processing and Telecommunications*, volume 1596 of LNCS, pages 17-30. Springer, 1999.

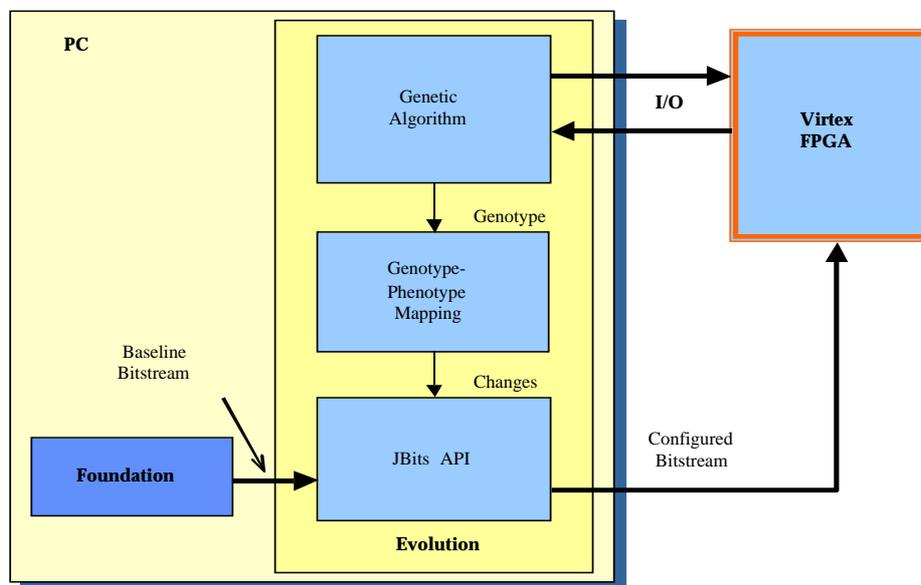


Figure 1: Virtex hardware set up

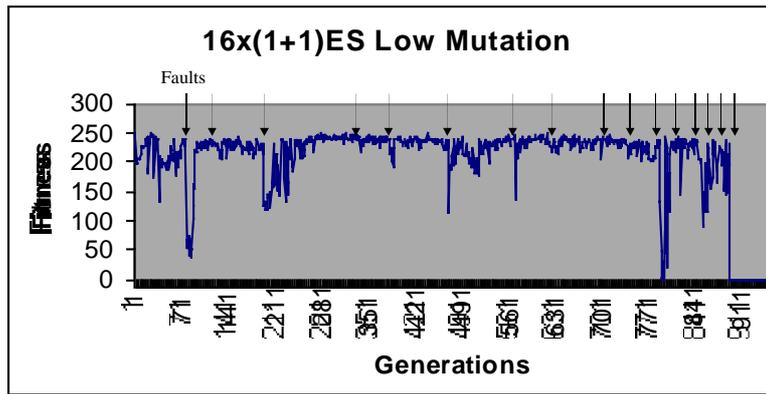


Figure 2: Fitness of the best individual when subjected to faults, with a low mutation rate

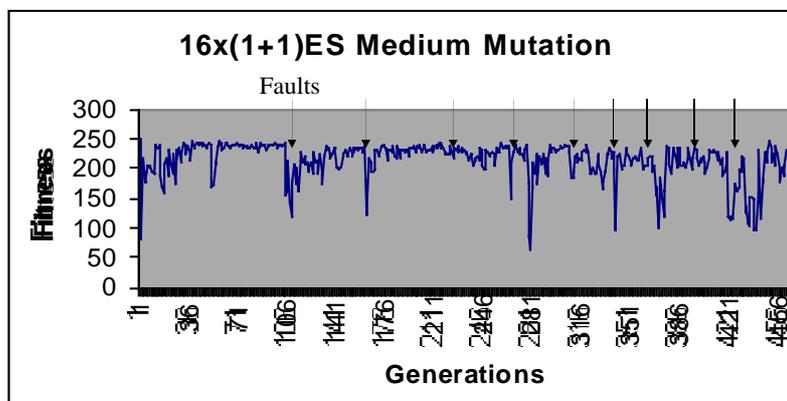


Figure 3: Fitness of the best individual when subjected to faults, with a medium mutation rate

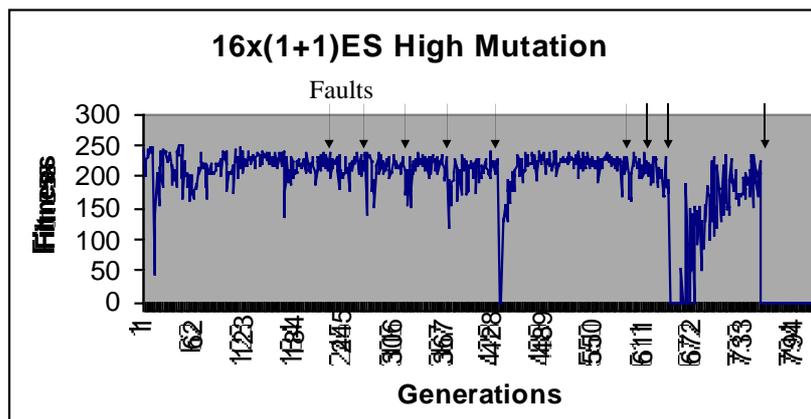


Figure 4: Fitness of the best individual when subjected to faults, with a high mutation rate

QuickTime™ and a
Photo - JPEG decompressor
are needed to see this picture.

Figure 5: Run of the 16 – 1+1 (4)

QuickTime™ and a
Photo - JPEG decompressor
are needed to see this picture.

Figure 6: Run of the 16 – 1+1 (3)

QuickTime™ and a
Photo - JPEG decompressor
are needed to see this picture.

Figure 7: Run of the 16 – 1+1 (2)

QuickTime™ and a
Photo - JPEG decompressor
are needed to see this picture.

Figure 8: Run of the 16 – 1+1 (1)