On the Practicality of Using Intrinsic Reconfiguration for Fault Recovery

Garrison W. Greenwood
Department of Electrical & Computer Engineering
Portland State University
Portland, OR 97207

Keywords: evolvable hardware, fault recovery, intrinsic evolution, reconfiguration

Abstract

Evolvable hardware combines the powerful search capability of evolutionary algorithms with the flexibility of reprogrammable devices, thereby providing a natural framework for reconfiguration. This framework has generated an interest in using evolvable hardware for fault-tolerant systems because reconfiguration can effectively deal with hardware faults whenever it is impossible to provide spares. But systems cannot tolerate faults indefinitely, which means reconfiguration does have a deadline. The focus of previous evolvable hardware research relating to fault-tolerance has been primarily restricted to restoring functionality, with no real consideration of time constraints. In this paper we are concerned with evolvable hardware performing reconfiguration under deadline constraints. In particular, we investigate reconfigurable hardware that undergoes intrinsic evolution. We show that fault recovery done by intrinsic reconfiguration has some restrictions, which designers cannot ignore.

1 Introduction

Reliable systems can be depended on to provide continual service. Unfortunately, faults are inevitable, which leads to disruptions in service. One way of increasing a system's availability is to make it fault-tolerant—i.e., capable of detecting and recovering from hardware faults. Exchanging a faulty component with an operating spare is the most widely used method for hardware fault recovery [1], but this method is not always practical. For instance, in some systems mission essential equipment occupies all of the available space leaving no room for spare hardware. Another case is when system failure results from an unanticipated change in the operational environment. For example, high radiation environments can induce effects in metal-oxide-semiconductor devices [2] and high temperatures can lead to failures in voltage converters [3]. Simply replacing the original system with an identical copy accomplishes nothing if the environmental conditions haven't changed.

Reconfiguring a faulty system eliminates the need for redundant hardware. Reconfiguration methodically changes the failed system's architecture until a desired functionality is restored. There is no guarantee full functionality can be restored in all cases. Nevertheless, reconfiguration is a viable fault recovery technique—especially when providing redundant hardware becomes impractical.

Our work presented in this paper concentrates on those fault tolerant systems that cannot use redundant hardware for fault recovery and are therefore forced to use reconfiguration. Spacecraft or unmanned air vehicles are good examples; both have little room for spare hardware and they must operate in potentially hostile environments.

Evolvable hardware (EHW) has emerged as a powerful method for doing original hardware design—which naturally suggests it could be equally useful for doing hardware reconfiguration. The idea behind EHW is to combine the biologically-inspired search methods of evolutionary algorithms with the flexibility of reconfigurable hardware. The evolutionary algorithm searches throughout the space of all possible configurations looking for the one that performs the best. Every configuration the evolutionary algorithm finds must be evaluated and there are two accepted methods: *extrinsic evolution* where the evaluation is done in software, and *intrinsic evolution* where the evaluation is done on a hardware implementation. In many instances intrinsic evolution is necessary because the only real way to evaluate a configuration is to implement it and have it actually operate in the physical environment. *Intrinsic reconfiguration* refers to any search process that uses intrinsic evolution to find new hardware configurations.

Three types of devices are suitable reconfigurable architectures: the field programmable gate array (FPGA) for digital applications; the field programmable analog array (FPAA) for analog applications; and the field programmable transistor array (FPTA) that can be used in either analog or digital applications. These devices represent different levels of granularity. Both the FPGA and FPAA provide configurable blocks of circuitry along with programmable routing resources. Conversely, the FPTA consists of an array of MOSFET transistors interconnected via programmable switches. A small number of capacitors are included on-chip, but resistors are synthesized using the MOSFET transistors. FPGAs and FPAAs are available as commercial off-the-shelf (COTS) devices (e.g., see [4] and [5]); the FPTA was fabricated for NASA's Jet Propulsion Laboratory and is currently only available for research studies [6].

Some recent work has shown EHW can be quite effective for reconfiguring existing hardware to overcome faults [7, 8, 9]. The ability of evolutionary algorithms to find good reconfigurations is not at issue in this paper. We instead are concerned with the issue of time. Most EHW-based studies rely on device simulators rather than physical hardware. This simulator use means the time to download a configuration, the time to program the device, and the time to conduct a fitness evaluation on the reconfigured hardware has largely been ignored—even though they dramatically affect the evolutionary algorithm's running time.

Systems cannot operate indefinitely with faults. It should be obvious that timing constraints must be considered whenever evolutionary fault recovery is used. Why then do researchers ignore them? Greenwood, et. al [10] were the first to suggest EHW-based reconfiguration has a time limit. Researchers almost exclusively focus on using EHW only to restore functionality. The evolutionary algorithm running time is usually reported, some-

times as a maximum, or sometimes as an average taken over many runs. Average running time doesn't guarantee compliance with a hard recovery deadline. Even the maximum running time is worthless because there is no guarantee some future trial run won't take too long.

Designers must contemplate a variety of factors before choosing a fault recovery method. This paper describes the advantages and limitations of intrinsic reconfiguration so designers can make an informed choice. In addition, several recommendations about the best ways to use intrinsic reconfiguration for fault recovery are included. No new methods to speed up the evolution are discussed. The reader should understand intrinsic reconfiguration properties are not application dependent and are not restricted to any particular type of evolutionary algorithm. Those properties are the central theme of this paper. Optimization issues are handled more appropriately within the context of a specific problem.

The paper is organized as follows. Section 2 compares extrinsic and intrinsic reconfiguration and explains why the latter is usually the only option for fault recovery. Section 3 gives a brief introduction to RTS and shows why fault-tolerant systems are real-time. Section 4 derives a formula for estimating the intrinsic reconfiguration time and gives some examples to show reconfiguration time can be surprisingly long. Section 5 presents guidelines for future development of fault recovery systems that employ intrinsic reconfiguration. Finally, some practical issues involving intrinsic reconfiguration are given in Section 6.

2 Why Intrinsic Reconfiguration?

We are concentrating on autonomous fault tolerant systems—i.e., systems that can detect, isolate and ultimately recover from a fault without human intervention. Whenever EHW is used for fault recovery, the evolution can be done either intrinsically with physical hardware or extrinsically with hardware simulations. Is one method better than the other? We believe only the intrinsic method makes any sense for two reasons:

(a) *lack of in-situ computing resources*

Most EHW research is conducted in laboratories using Pentium PCs or UNIX workstations—resources not usually installed on a deep-space probe heading towards the planet Neptune! A more likely computing environment for reconfiguration operations would be a single low-end microprocessor. Such a processor is relatively slow and can only address small amounts of memory, which affects the simulation runtime and the accuracy of the hardware models. Lack of space also precludes using pipelined hardware or multiple processors that could speed up the simulation. Conversely, intrinsic reconfiguration doesn't use hardware models, so there are no modeling inaccuracies. Low-end microprocessors are quite capable of running evolutionary algorithms used for EHW. In fact, compact genetic algorithms even run entirely on a single VLSI chip [11].

(b) *inherent inaccuracies in hardware simulations*

A serious problem for spacecraft is collision with orbital debris [12]. Unforeseen failures like these are hard to represent accurately in a simulation, and even then only if the precise nature of the failure is known. Unfortunately, getting accurate failure

information is nearly impossible with truly autonomous systems, especially when the failure was caused by exogenous events such as collisions. Evolving a new hardware configuration with an incomplete or inaccurate simulation is futile. On the other hand, intrinsic reconfiguration never has to make simplifying assumptions because the evolving hardware is directly interfaced to the damaged system. Hence, there is no doubt about how much functionality is restored in the reconfigured system.

The above discussion should not imply intrinsic reconfiguration has no disadvantages. Fault recovery is inherently a real time process—an idea largely ignored by the evolvable hardware community. We will shortly show that intrinsic reconfiguration can take surprisingly long. In fact, under certain circumstances it may take so long that it cannot be used for fault recovery! Our objective is to first show designers the limitations of intrinsic reconfiguration and second to provide the necessary tools for evaluating its potential as a fault recovery method. However, before discussing those issues the reader must understand what makes a system a real-time system.

3 Real-time systems

This section provides a brief introduction to real-time systems. Interested readers are referred to some of the excellent books on this topic for further information (e.g., see [13]). We begin with a formal definition of a real-time system.

Definition: (real-time system)

Any system that is both logically and temporally correct

Logical correctness means the system satisfies all functional specifications. Temporal correctness means the system is guaranteed to perform these functions within explicit timeframes. Fault-tolerant systems qualify as real-time systems because fault detection and fault recovery inherently have deadlines. That is, the fault must be detected within a certain period of time after it occurs, and the fault must be corrected within a certain period of time after it is detected. Fault recovery may also have an expected start time.

The notion of real-time is often interpreted to mean really fast. This interpretation is not correct. Real-time does not necessarily mean fast—and fast does not necessarily mean real-time. Suppose a document must be sent from Chicago to London, and two delivery systems are available: surface mail with a guaranteed 3-day delivery time or e-mail with a guaranteed 5 minute delivery time. The e-mail delivery is orders of magnitude faster than surface mail, but that does not necessarily mean it qualifies as a real-time delivery system. It is the required delivery deadline that ultimately establishes whether the real-time system definition has been met. For example, both systems are real-time systems if the deadline is six days because both are logically and temporally correct. However, neither one is a real-time system if the deadline is three minutes because neither one is temporally correct.

Real-time systems are classified as hard or soft. Hard systems have catastrophic consequences if the temporal requirements are not met—up to and including complete system

destruction. In fact, if the hard system is safety-critical, failure could lead to injury or even death. Conversely, soft systems only have degraded performance if the temporal requirements aren't met. The classification of a fault-tolerant system, in particular, depends on the nature of the faults and the consequences for failing to detect and correct them in a timely manner. Suppose a fault results in an over-temperature condition. If the system hardware can survive this condition for up to five minutes, then fault recovery must be completed within five minutes to prevent further damage. This would be a hard fault-tolerant system. On the other hand, if the fault only causes a minor loss of some sensor data, fault recovery could take considerably longer without dire consequences. This would be a soft fault-tolerant system.

4 Quantifying Reconfiguration Time

In this section we derive a formula for estimating the intrinsic reconfiguration time and describe how the formula is used. The derivation assumes the hardware environment consists of a single processor—i.e., a microprocessor or microcontroller—interfaced to a single reconfigurable device. The processor is responsible for executing the evolutionary algorithm, converting configurations into a proper file format, downloading the configuration to the device, and running fitness tests to evaluate the configuration. The processor may or may not be dedicated to reconfiguration. That is, the processor may be engaged with other tasks whenever reconfiguration is not being performed.

4.1 Formula Derivation

The evolutionary algorithm manipulates a genome that encodes all of the information needed to create a hardware configuration. During each generation λ offspring are created. However, with intrinsic evolution an offspring's fitness evaluation cannot begin until the configuration information is physically downloaded and the device is programmed. Let t_p denote the time it takes to do this. Since λ downloads take place every generation, the download time per generation lasts λt_p seconds. If t_f denotes the duration of the fitness test per individual, then λt_f is the fitness evaluation time per generation.

The t_p value depends on the type of reconfigurable device—i.e., an FPGA, FPAA or FPTA—whereas t_f is application dependent because the test duration is determined by the scope of the fitness test. Every offspring consumes $t_p + t_f$ time units. It therefore follows that an evolutionary algorithm running for k generations, while producing λ offspring per generation, has an intrinsic reconfiguration time of

$$T_r(k, \lambda) \;=\; k\lambda(t_p + t_f) \tag{1}$$

4.2 Some Examples

Table 1 shows the programming time (t_p) for several reconfigurable devices. This programming time cannot be ignored, despite being rather small in most FPGAs, because EHW

Device	Type	Size	Mfg	t_p (ms)	Ref.	Notes
ispPAC10	FPAA	4	Lattice Semiconductor	100	[14]	
AN220E04	FPAA	4	Anadigm	3.8	[15]	1, 2
XC3020A	FPGA	64	Xilinx	1.5	[16]	2
Virtex XCV50	FPGA	1728	Xilinx	7	[4]	3
XC4085XL	FPGA	3136	Xilinx	192	[16]	2
APEX II EP2A70	FPGA	6720	Altera	12.5	[17]	4
JPL's FPTA2	FPTA	64	fabricated by MOSIS	0.008	[18]	5

(1) All 18 banks are reloaded with 256 bytes/bank
(2) Serial transfer with 10 MHz clock
(3) Byte-wide transfer with 10 MHz clock
(4) Byte-wide transfers with 66 MHz clock
(5) Byte-wide transfers with 160 MHz clock

Table 1: Programming times for various popular reconfigurable devices. All are COTS devices except for the FPTA. The units for size are configurable logic blocks for FPGAs, modules for FPAAs, and cells for FPTAs. The references indicate where the t_p value is documented.

algorithms frequently have populations sizes in the hundreds and they run for thousands of generations.

Example 1:

An EP2A70 FPGA is intrinsically evolved by a generational GA. The population size is 200 and 1000 generations are processed. From Table 1, $t_p = 12.5$ ms. Then the time spent just reprogramming the FPGA is $\lambda \cdot k \cdot t_p$ or about 42 minutes.

t_f is responsible for the rather long search times often encountered in analog applications. For instance, suppose a proportional-derivative controller is implemented in an FPAA. A controller's fitness is found by applying a step input to the control system and then measuring its settling time. The fitness evaluation lasts at least as long as the settling time does, which can be somewhat lengthy. Indeed, settling times of two minutes are not unheard of [19]. Under these circumstances, it wouldn't take a very large population size nor a large number of generations to make an intrinsic reconfiguration run for hours or even days before finishing.

Example 2:

An AN220E04 FPAA is used to compensate for aging effects in a control system responsible for positioning a satellite's communications antenna. The reconfiguration search is done by a generational GA run for 500 generations with a population size of 100. The system's step response is measured to determine if the compensation is correct. This step response test takes $t_f = 625$ milliseconds to conduct. Hence, $\lambda = 100$, $k = 500$ and $t_p = 3.8$ ms. Substituting into Eq. (1) yields $T_r(500, 100) \approx 8.7$ hours.

5 Discussion

Reconfiguration times are meaningless until they are put into context. For instance, take Example 2 from the previous section. Suppose brief communication sessions with the satellite are scheduled at 10 hour intervals. A session may be skipped, but skipping two sessions in a row is not permitted. If a fault is detected just prior to a scheduled session, and if the error results in missing the session, then the fault recovery deadline is 10 hours. This is the worst case scenario[1]. An almost 9 hour reconfiguration time may seem quite long, but in this case it is perfectly acceptable because $T_r < 10$. On the other hand, it would not be acceptable if communication sessions were scheduled at 6 hour intervals.

The only way to determine if there is a problem is to compare the reconfiguration time against the fault recovery deadline. This latter quantity is system dependent. No problem exists so long as the reconfiguration time is less than the recovery deadline.

This time comparison adds a new perspective on intrinsic evolution and, at the same time, imposes a new requirement. Reconfiguration becomes a real-time process whenever it is used as a fault recovery method. Consequently, it is no longer sufficient to just talk about how an evolutionary algorithm was able to restore a circuit's functionality. These statements may show logical correctness, but without comparing the reconfiguration time against a deadline there is no proof of temporal correctness. Just reporting an algorithm's running time doesn't say anything about temporal correctness either. The key point is expressed by the following first principle:

> *No fault recovery method can legitimately proclaim efficacy until it is proven to be both logically and temporally correct.*

The validity of this principle is easy to see. If the recovery method isn't logically correct, then the problem can't be fixed. If it isn't temporally correct, then the problem can't be fixed soon enough to prevent other things from going wrong. Without proving logical and temporal correctness, there is no basis for claiming a fault recovery method is effective.

It is easy to prove if a fault recovery method is logically correct—try it and see if it fixes the problem. Proving temporal correctness, however, is more complicated because it really depends on conducting a thorough failure modes and effects analysis (FMEA) or fault tree analysis (FTA). This analysis should identify all potential faults and their effects on system performance [20]. One outcome of the failure analysis is the recovery deadlines. Temporal correctness is proven if a logically correct recovery is guaranteed to finish prior to the recovery deadline. (Unfortunately, no EHW-based fault recovery method is temporally correct. See Section 6 for futher details.)

No attempt was made in this study to promote one reprogrammable device over another. In fact, the devices we chose to study were picked merely to show the broad range of what is available. There is, however, one caveat. Some reprogrammable devices, such as the ispPAC10, are E^2PROM based. Hence, there is a limit to the number of times the device can be reconfigured. RAM based devices should be used if the population size and/or the number of generations is large.

[1] Missing one session is permitted. If the fault is detected just after a scheduled session, the fault recovery deadline would be 20 hours.

Up to this point only the programming and fitness evaluation times has been considered. However, the time an evolutionary algorithm takes to do all of the other tasks done each generation, such as conducting binary tournaments or performing n-point crossover, is also important. The genome associated with a particular reprogrammable device is always the same, so the overhead associated with running the evolutionary algorithm is roughly constant each generation regardless of the application. Let t_{oh} denote the constant overhead value. Then a more accurate formula for the reconfiguration time is

$$T_r(k, \lambda, t_{oh}) \;=\; k\lambda(t_p + t_f) + kt_{oh} \tag{2}$$

The number of generations (k) and the evolutionary algorithm overhead time (t_{oh}) can dramatically affect reconfiguration time. k depends on the thoroughness of the search and how easily the evolutionary algorithm escapes local optima. The overhead time depends on a variety of factors including how parents are chosen, the complexity of the reproduction operators, and so on. Additional time must be added to account for the time to convert each genotype into the proper file format needed for the download.

Greenwood, et. al [10] suggested evolutionary algorithms designed for reconfiguration searches perform best if they have high selection pressure and if they emphasize mutation for reproduction. In principle, any type of evolutionary algorithm could be used for a reconfiguration search, but from a practical standpoint genetic programming algorithms should be avoided. Genetic programming algorithms designed for EHW problems are put on large multiprocessor systems to abridge their long running time [21, 22]. This becomes especially problematic for fault-tolerant systems because, if there isn't enough room for redundant hardware, then there isn't enough room for a large multiprocessor system either. It seems unlikely a full-fledged genetic programming search, run on a single processor, could finish quickly enough to meet a fault recovery deadline of only a few hours.

Finally, some devices can be partially reconfigured—i.e., configuration changes can be restricted to only small portions of the device. The idea is to identify the particular region in the bitstream which requires reconfiguration. A header is then wrapped around this bitstream data to identify the address at which to start reconfiguration [23]. Partial reconfiguration reduces t_p. This reduction may be important in situations where t_p and t_f are of the same order of magnitude.

6 Putting Theory Into Practice

We assume the system is fully designed and it meets all functional specifications. The objective is now to find out how the system can fail and the resultant effects. This evaluation can be done in several ways and the interested reader can find more comprehensive information elsewhere (e.g., [24] contains an excellent introduction).

Systems are typically evaluated with one of the following methods:

- *failure modes and effects analysis (FMEA)*

  A bottom-up method that looks at each component's failure modes and how those failures affect system performance.

- *fault tree analysis (FTA)*

  A top-down method that assumes a specific failure has happened and then analyzes each subsystem to look for the cause.

- *failure modes and effects testing (FMET)*

  Failures are injected into the system to observe their effects.

A designer would use either a FMEA or a FTA to identify the system's failure modes and their effects. For each effect the analysis will show what actions (if any) would prevent the effect from occurring, and if so, how much fault recovery time is allowed.

In principle a fault recovery should not be used unless it is both logically and temporally correct. Logical correctness is proven with a FMET—i.e., inject the fault and see if intrinsic reconfiguration prevents the effect from occurring. However, proving temporal correctness is an entirely different story.

The reality is temporal correctness cannot be formally proven for intrinsic reconfiguration because it uses an evolutionary algorithm. Since these algorithms are stochastic, independent runs are not guaranteed to produce a repeatable outcome. Hence, there is no way to guarantee a fixed number of generations will always find a desired circuit configuration.

This limitation does not, however, mean intrinsic reconfiguration cannot be used for fault recovery. It is also a reality that no system is perfect, always capable of performing when called upon. Systems do fail, so all recovery methods—whether or not they use evolutionary algorithms—have a finite probability of not executing. Consequently, the primary concern is to reduce the likelihood of failure. All a designer can do is minimize the failure risk to an acceptable level. There are ways to minimize risk when using intrinsic reconfiguration, but that depends on whether the fault is anticipated or unanticipated.

6.1 Minimizing Risk With Anticipated Faults

Anticipated faults are those faults identified while conducting an FMEA or FTA. These faults are precisely known and their effects are observed. Moreover, the fault recovery time is known because the analysis tells the time delay between the fault and its effect.

It was stated above that an FMET will verify logical correctness. But the first thing—before even conducting the FMET—is to see if intrinsic reconfiguration has any chance of meeting a recovery deadline. Here simulation can be useful[2]. Once the fault is identified a designer can envision the type of circuitry needed for fault correction. The designer could construct an evolutionary algorithm to generate candidate hardware configurations and use the simulator to evaluate them. A test run will show how many generations it takes to produce the necessary circuit configuration. Eq. (2) will estimate the physical hardware reconfiguration time which can then be compared against the fault recovery deadline.

The FMET should be conducted if and only if the reconfiguration time is less than the recovery deadline. The best way to minimize any risk is to allow sufficient slack time. In

[2]Some COTS device vendors provide simulators for their products. Custom FPAAs and FPTAs and some mixed-signal devices can use PSpice for simulation.

other words, the intrinsic reconfiguration should end well short of the recovery deadline. The definition of "well short" of course depends on the system and the consequences of missing the recovery deadline. In safety-critical systems—i.e. systems where failure can lead to total system destruction, physical injury or even death—the recovery deadline should be several times longer than the reconfiguration time. That requirement can be relaxed for systems that aren't safety-critical, but reducing the reconfiguration time will always decrease the risk. *This means the designer must know what constitutes an acceptable level of risk before specifying the maximum reconfiguration time.*

Another way of reducing risk uses concepts taken from on-board preventive maintenance programs [25]. The main idea is to do fault recovery in stages. In other words, during each stage, which is a fixed time period, intrinsic reconfiguration is only expected to find an somewhat improved solution. The evolutionary algorithm uses the final population from the previous stage as the initial population in the current stage. The (still faulty) system has degraded performance, but it remains online between stages. This process continues until full recovery is achieved.

No fault recovery procedure can remove all risk. This means other actions may be necessary. See MIL-STD-882D for some additional risk mitigation measures [26].

6.2 Minimizing Risk With Unanticipated Faults

Unanticipated faults are caused by events beyond a designer's control. Two sources of unanticipated faults are persistent, unexpected environmental changes or damage resulting from exogenous events such as collision with orbital debris. In many instances the exact change in the environment or the extent of damage cannot be accurately determined. Consequently, unanticipated faults frequently have unpredictable effects. Without knowing the effect, it is impossible to state the recovery time; temporal correctness is no longer relevant. However, logical correctness may still be important but that depends on the source of the unanticipated fault.

6.2.1 Changed Environment

Component characteristics can change when the operational environment changes [2, 3]. A new hardware configuration might exploit those changed characteristics to restore any lost functions. For example, Stoica et al. [6] successfully reconfigured a FPTA to restore functions lost due to temperature increases. Logical correctness always holds for these type of problems because the recovery objective is to completely restore all original system functions.

6.2.2 Exogenous Events

Exogenous events create havoc with an operating system because there is no way to predict what faults will be induced; there is likewise no way to predict the ultimate effects. Consequently, logical correctness may be unattainable because there is no guarantee the available fault recovery method can deal with an unspecified failure mode.

Under these circumstances the corrective action switches from restoring system functions to guaranteeing system survival—which contradicts the normal interpretation of logical cor-

rectness in a fault tolerant system. Intrinsic reconfiguration would initially just search for a stable system configuration that stops any fault migration. For example, compensators can keep a system stabilizable even when a sensor or actuator completely fails [27]. Any other recovery actions may have to be done after the damaged system is taken offline.

We conclude this section with the following two remarks:

1. Deterministic recovery methods are designed to handle only a very specific set of faults. They therefore have almost no ability to deal with unanticipated faults—especially faults caused by exogenous events. On the other hand intrinsic reconfiguration can evolve novel circuitry that a designer never considered. In this realm we believe intrinsic reconfiguration has the most potential as a fault recovery method.

2. Temporal correctness is ignored under certain circumstances. Does this mean the first principle of fault recovery is invalid? Not at all. *The first principle always holds when recovery is from an anticipated fault.* The reason is an anticipated fault can be thoroughly analyzed to determine its effect and recovery time; temporal correctness requirements are therefore clearly defined. Conversely, unanticipated faults were never analyzed, so any discussions about recovery time are meaningless. Temporal correctness cannot be defined in these cases.

7 Conclusions

EHW-based reconfiguration is a viable method of performing fault recovery in systems without redundant hardware. Fault-tolerant systems are real-time systems. Consequently, any attempts to intrinsically evolve a new hardware configuration must consider the device programming time, the fitness evaluation time, and the evolutionary algorithm overhead time because they all contribute to the reconfiguration time. Eq. (2) gives a formula that estimates the intrinsic reconfiguration time.

It has been shown neither a large population size nor thousands of generations are necessary to have reconfiguration searches with surprisingly long finishing times. However, a long search time by itself is not enough to reject reconfiguration as a fault recovery method. Intrinsic reconfiguration can be used for fault recovery so long as it finishes before the mandatory recovery deadline.

Finally, a first principle of EHW-based reconfiguration was given. All intrinsic reconfiguration fault recovery methods are stochastic processes—which means they can never satisfy the first principle because temporal correctness cannot be satisfied. This does not mean intrinsic reconfiguration can't be used be fault recovery. The designer just needs to be aware there is some risk involved. Intrinsic reconfiguration can be used so long as the designer believes the risk is acceptable.

References

[1] A. Avizienis. Towards systematic design of fault-tolerant systems. *IEEE Comput.*, 30(4):51–58, 1997.

[2] H. Hughes and J. Benedetto. Radiation effects and hardening of MOS technology: devices and circuits. *IEEE Trans. Nuclear Sci.*, 50(3):500–521, 2003.

[3] M. Wismer. Steady-state operation of a high-voltage multiresonant converter in a high-temperature environment. *IEEE Trans. Power Elec.*, 18(3):740–748, 2003.

[4] Virtex 2.5V field programmable gate array product specification DS003-1 (V 2.5). Xilinx Inc., April 2, 2001.

[5] AN220E04 datasheet—dynamically reconfigurable FPAA. Anadigm Inc., 2002.

[6] A. Stoica, D. Keymeulen, R. Zebulum, A. Thakoor, T. Daud, G. Klimeck, Y. Jin, R. Tawel, and V. Duong. Evolution of analog circuits on field programmable transistor arrays. In Jason Lohn et. al, editor, *The Second NASA/DoD workshop on Evolvable Hardware*, pages 99–108, 2000.

[7] D. Keymeulen, R. Zebulum, Y. Jin, and A. Stoica. Fault-tolerant evolvable hardware using field-programmable transistor arrays. *IEEE Trans. Reliab.*, 49(3):305–316, 2000.

[8] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Embryonics: a new methodology for designing field programmable gate arrays with self-repair and self-replicating properties. *Proc. of the IEEE*, 88(4):516–541, 2000.

[9] L. Sekanina and V. Drabek. Relation between fault tolerance and reconfiguration in cellular systems. *Proc. 6th IEEE on-line testing workshop*, pages 25–30, 2000.

[10] G. Greenwood, E. Ramsden, and S. Ahmed. An empirical comparison of evolutionary algorithms for evolvable hardware with maximum time-to-reconfigure requirements. In J. Lohn et. al, editor, *Proc. 2003 NASA/DOD Conf. on Evol. Hdwe*, pages 59–66, 2003.

[11] H. Gallagher and S. Vigraham. A modified compact genetic algorithm for the intrinsic evolution of continuous time recurrent neural networks. In W. Langdon et.al, editor, *Proc. GECCO 2002*, pages 163–170, 2002.

[12] C. Belk, J. Robinson, M. Alexander, W. Cooke, and S. Pavelitz. *Meteoroids and orbital debris: effects on spacecraft.* NASA Reference Publication 1408, 1997.

[13] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley-Longmain, 3rd edition, 2001.

[14] ispPAC10 in-system programmable analog circuit datasheet. Lattice Semiconductor Corp., 2000.

[15] AN220E04 user's manual UM020800-U002g. Anadigm Inc., 2002.

[16] Dynamic reconfiguration application note XAPP093. Xilinx Inc., November 10, 1997.

[17] APEX II programmable logic device family data sheet. Altera Inc., August 2003, Ver 3.0.

[18] R. Zebulum, Y. Jin, and A. Stoica. JPL, private communication, 2003.

[19] NGST yardstick mission. *NGST Monograph No. 1*, Next Generation Space Telescope Project Study Office, Goddard Space Flight Center, 1999.

[20] Facility system safety guidebook. *NASA-STD-8719.7*, January 1998.

[21] M. Streeter, M. Keane, and J. Koza. Routine duplication of post-2000 patented inventions by means of genetic programming. In J. Foster et.al, editor, *Genetic Programming: 5th Euro. Conf., EuroGP 2002*, pages 26–36, 2002.

[22] M. Keane, J. Koza, and M. Streeter. Automatic synthesis using genetic programming of an improved general-purpose controller for industrially representative plants. In A. Stoica et. al, editor, *The 2002 NASA/DoD Conference on Evolvable Hardware*, pages 113–122, 2002.

[23] G. Hollingworth, S. Smith, and A. Tyrrell. The intrinsic evolution of Virtex devices through internet reconfigurable logic. *Proc. 3rd Int'l Conf. ICES 2000*, LNCS 1801 (Springer, Berlin), J. Miller et. al (Eds.):72–79, 2000.

[24] W. Dunn. *Practical design of safety-critical computer systems.* Reliability Press, 2002.

[25] A. Tai, L. Alkalai, and S. Chau. On-board preventive maintenance: a design-oriented analytic study for long-life applications. *Performance Eval.*, 35(3-4):215–232, 1999.

[26] MIL-STD-882D. *Standard Practice for System Safety.* Department of Defense, 10 February 2000.

[27] J. Stoustrup and V. Blondel. Fault tolerant control: a simultaneous stabilization result. *IEEE Trans. Auto. Control*, 49(2):305–310, 2004.