SELF-HEALING MEMORY HARDWARE ARCHITECTURE ON FIELD PROGRAMMABLE GATE ARRAY

Nhu Truong[1], Anthony de Souza-Daw[1], Robert Ross[2], Thang Manh Hoang[3], and Tien Dzung Nguyen[3]

[1] School of Electrical & Computer Engineering, RMIT International University Vietnam, Ho Chi Minh City, Vietnam, e-mail: thanhnhu1711@gmail.com, anthony.desouza-daw@rmit.edu.vn
[2] Department of Computer Science & Engineering, La Trobe University, Melbourne, Victoria, Australia, e-mail: rross@latrobe.edu.au
[3] School of Electronics and Telecommunications, Hanoi University of Science and Technology, Hanoi, Vietnam, Tel : +84 43692242, e-mail: thang.hoangmanh@hust.edu.vn

Received Date: February 20, 2014

Abstract

Hardware Fault-Tolerance is the set of techniques to remain operational after a fault by design. Programmable Logic Devices are good platforms to implement Hardware Fault-Tolerant techniques by utilizing abundant resources and facilitating self healing operations. In this paper we propose a hardware fault-tolerant architecture to duplicate components in order to replace faulty ones. The proposed architecture is markedly different from other works that mostly focuses on reconfiguring and evolving logic units rather than our evolvable memory units. The self-reparation process for a memory failure is the reallocation and synchronization of memory content. The internal flip-flops form an abundant reconfigurable resource and are reconfigured to work as newly created memory. The proposed architecture has been downloaded and tested on a real FPGA development board and has satisfied all of its pre-defined specifications.

Keywords: Evolvable hardware, Fault-tolerant hardware, Memory synchronization channel, Partial reconfiguration, Self-healing hardware

Introduction

Electronic hardware faults are physical defects which occur in parts of a system and which may produce unexpected outputs from affected systems. Such errors can be transferred to system outputs which can propagate throughout the system causing system failure (Anderson, T.; Lee, P. A., 1992). Traditionally, mission critical applications (e.g. aerospace, defense and medical) use complex hardware structures with multiple levels of redundancy to prevent faults from causing system failure (Dingman, C. P.; Marshall, J., 1995; Jafari, T.; Dabiri, F.; Brisk,P.; Sarrafzadeh, M., 2005; Moser, L.; Melliar-Smith, M., 2003). It is therefore advantageous to maintain circuit behaviour even when internal faults occur—significantly improving reliability. So far, several approaches and techniques of Hardware Fault-Tolerant have been proposed to deal with the potential hardware defects occurring in critical systems. Traditional hardware fault tolerant design does not autonomously fix the faulty components (Anderson, T.; Lee, P. A., 1992), which means any corrupted redundant modules will continue to produce corrupted outputs. Hence, when enough redundant modules have been corrupted system failure will occur.

In practice, a hardware fault tolerance design is based on “hardware redundancy”. Hardware redundancy is a collection of unused components which are planned for the purpose of detection and replacement of faulty components (Anderson, T.; Lee, P. A., 1992). In normal (fault-free) conditions, these redundant parts do not contribute to the overall system behaviour and are classified as redundancies at a system level. However, when a faulty component is detected, the corresponding redundant component (a duplicate) will replace the original one (the faulty one) and continue to perform the designed function.

A concept diagram of hardware redundancy technique is shown in Figure 1. Module A is a typical implementation without hardware redundancy. Module B is designed with hardware redundancy having *n* duplicates, which means it can tolerate a maximum of *n* non-simultaneous faults.

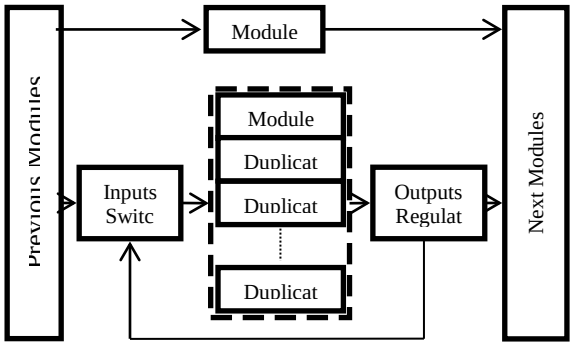

Figure 1. The concept of hardware redundancy

The number of redundant modules for each section is specified during the design stage, because of the fixed physical ASIC layout. Hardware is under-utilized while waiting for a fault to occur and hence is an inefficient but necessary allocation of resource. In practice, some parts of the system are more likely to be corrupted than others, thus such parts will use up their duplicates faster (Carter, 1986). A well-designed hardware fault tolerance system can allocate more redundant resources to the most likely-to-be-failed modules based on a failure rate prediction, thus improving its resource allocation efficiency (Carter, 1986).

Recently, traditional Hardware Fault-Tolerant techniques have experienced a conceptual change since the dawn of Evolvable Hardware, mainly in terms of the use of redundant resources. Specifically, the redundant resources are pre-allocated to each under-protected component in traditional techniques, while they are shared among all under-protected components in Evolvable Hardware. Meanwhile, the Evolvable Hardware concept provides a possibility of allowing the PLD to autonomously reconfigure itself (Evolvable hardware systems: EHW overview, 2004). Most designs in this field are using a system on chip (SoC) architecture with a central processing unit (CPU) to safely facilitate the bitstream manipulation (Hollingworth, G.; Smith, S.; Tyrrell, A., 2000). A complete design flow is also reduced to a simpler one which does not require complex calculations for synthesis, mapping, placing, and routing (Trahan, 2005; Haddow & Tufte, 2001; Zhang, Y.; Smith, S.; Tyrnell, A., 2004).

For memory fault recovery, as proposed in this paper, firstly the memory hardware needs to be reallocated and then the memory contents need to be synchronized to the current hardware system operations. The self-replication process uses partial reconfiguration which facilitates the manipulation of low-level architecture bits to change the configuration of logic layers. The proposed architecture also presents a unique memory synchronization channel, which can update the internal flip-flops of the newly created component to keep the contents aligned with the rest of the system. Thus, the system’s functional behaviour can be maintained even if physical errors occur inside a sequential component. The hardware platform chosen for design’s implementation is a firm-core virtual FPGA which claimed to have a structure

compatible with commercial FPGAs. This makes the design more flexible and realistic when considering exporting to other FPGA families rather than fixed to a particular FPGA family.

Background and Literature Review

The concept of EHW was introduced 1992 and become an intensive research topic when Daniel Mange and Tetsuya Higuchi published research proposing the concept (Higuchi, T.; Liu, Y.; Iwata, M.; Yao, X., 2006). Thereafter, hardware fault tolerance techniques have experienced a new design flow, in which the under-protected module can autonomously change its physically structure in order to perform self-healing or self-regenerating functions for its faulty parts. It is noted that, with this new approach, the overall system is uninterrupted during the change of structure. Various methods including many software algorithms and hardware architectures have been proposed to guide the manipulation process of partial bitstream such as REPLICA, BiRF, BAnMaT Light, etc.

In practice, EHW is the implementation of evolutionary computation on Programming Logic Devices (PLDs) with the ability of autonomous, self reconfiguration in order to adapt to new design criteria or archive high fault-tolerance capability (Higuchi, T.; Liu, Y.; Iwata, M.; Yao, X., 2006). However, PLDs cannot autonomously generate any function without guidance. Thus, the implementation of evolutionary computation is required in order to guide the process of evolving functionality.

The structure of EHW consists of interconnected electronic components that can dynamically reconfigure via evolutionary algorithms. The first implementations of EHW were realized using Field Programmable Gate Arrays (FPGAs), where the circuit can be modified at run-time. The basic idea is to regard the architecture bits of PLDs as a configuration image of current circuit to duplicate or evolve completely new fault-free components to replace the faulty ones (Greewood, G. W.; Hunter, D.; Ramsden, E., 2003). As shown in Figure 2, the block of M2 Faulty is replaced by M2 Good to maintain the operation.

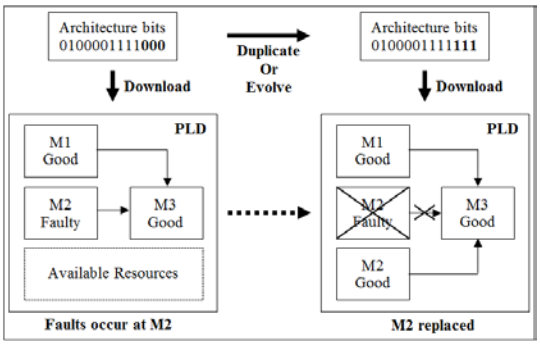

Figure 2. Hardware fault-tolerance based on EHW

It is noted that the Embryonic Project introduced by Mange et al. around 1998 is one of the first examples of research in hardware fault tolerance utilizing the EHW concept. In the pioneer works, Higuchi's (Higuchi, T.; Liu, Y.; Iwata, M.; Yao, X., 2006) aimed at evolving a completely new circuit, while Mange's manipulated the existed genomes to change the circuit physical layout (placement and routing) and to avoid the presences of physical defects inside the operating parts of the system.

By adopting certain features of cellular organization unique properties of the living world, such as self-replication and self-repair, can also be applied to artificial objects (integrated circuits) (Mange, D.; Stauffer, A.; Tempesti, G.; Vannel, F.; Badertscher, A., 2006). The application of EHW in self-repairing and self-replication resembles one of the most well-known bio-inspired hardware concepts proposed by Moshe Sipper in 1997, the Phylogeny, Ontogeny, and Epigenesis (POE) model, which attempts to mimic the natural behaviour of

living organisms in the context of electronic circuit (Sipper, M.; Sanchez, E.; Mange, D.; Tomassini, M.; Pérez-Uribe, A.; Stauffer,A., April 1997).

This work combines the Phylogeny and Ontogeny dimensions, which aims to utilize available resources on a PLD to replicate a 'healthy' component to replace a faulty one. Thus, it forms a new system with the same logic structure (same functional behaviors) but different hardware structure (different physical layout).

Bitstream Manipulation

There are a number of FPGA bitstream manipulation tools for partial reconfiguration (PR). For example, Bitstream Relocation Filter (as known as BiRF) and BAnMaT Light tool allow the relocation of a partial bitstream with minimal overhead during the download process (Ferrandi, F.; Morandi, M.; Novati, M.; Santambrogio, M. D.; Sciuto, D., 2006; Corbetta, S.; Ferrandi, F.; Morandi, M.; Novati, M.; Santambrogio, M. D.; Sciuto, D., 2007). In 2008, Note and Rannaud presented an in-depth analysis of the Xilinx bitstream format, which allows FPGA designers to manually compile and decompile bitstream without using Xilinx tools (Note, J.; Rannaud, E., 2008). Other open-source tools have been developed to facilitate the PR technique such as BitMaT (Morford, 2005), GoAhead (Beckhoff, C.; Koch, D.; Torresen, J., 2012), OpenPR (Sohanghpurwala, A. A.; Athanas, P.; Frangieh, T.; Wood, A., 2011), RapidSmith (Lavin, C.; Padilla, M.; Lamprecht, J.; Lundrigan, P.; Nelson, B.; Hutchings, B., 2011), Verilog to Routing (VTR) (Rose, J.; Luu, J.; Yu, C. W.; Densmore, O.; Goeders, J.; Somerville, A.; Kent, K. B.; Jamieson, P.; Anderson, J., 2012). These are designed to perform specific stages within the PR design flow, except for VTR. It would be useful to implement the routing algorithm extracted from one of these tools into the proposed architecture to replace the Bus Macro (BM) and improve the flexibility of designs. However, it is no coincidence that all of these algorithms are designed as software tools, because implementing an identical hardware-based function would be very costly in term of resource utilization.

Core Relocation

Dynamic core relocation is one of the most well-known applications of PR technique, in which the target module can be relocated to another location within the Partial Reconfiguration Region (PRR) while others continue operating (Kalte, H.; Lee, G.; Porrmann, M.; Ruckert, U., 2005). The relocated module will keep its IO connections through a specific BM which spans across the PRR and provides universal access to other modules within the system. However, the design cannot be dynamically reconfigured as it does not utilize an internal configuration access port (ICAP). In addition, creating a BM which spans throughout the horizontal dimension of PRR will potentially cause routing congestion issues as it also occupies parts of the routing resource on the available region for each module. Relocating any module may induce errors since it will send the BM to an unknown state temporarily during the configuration time.

Since the emergence of REPLICA project (Kalte, H.; Lee, G.; Porrmann, M.; Ruckert, U., 2005), researchers have proposed various architectures to improve the dynamical core relocation technique. For instance, Becker et al. proposed a core swapping architecture to enhance the relocation capability of partial bitstreams for FPGA run-time reconfiguration (Becker, T.; Luk, W.; Cheung, P. Y. K., 2007). A theoretical two-dimensional approach for core relocation was proposed by Morandi et al., considering the possibility of treating the PRR as a 2D plane for the relocation of rectangle objects (Morandi, M.; Novati, M.; Santambrogio, M. D.; Sciuto, D., 2008). The real architecture was then built by Rossmeissl, which allows increasing the efficiency of resource utilization by grouping modules (Rossmeissl, C.; Sreeramareddy, A.; Akoglu, A., 2009). However, Rossmeissl's work is still using the BMs spanning across the PRR.

Proposed Architecture

This research utilizes partial reconfiguration within the FPGA to replicate non-faulty components to generate redundant ones for future replacements in real time. In this research, a Virtex II chip with PR is utilized as the hardware platform.

To be able to manipulate the architecture bits, Xilinx has provided a hybrid intellectual core called the Hardware Internal Configuration Access Port (HWICAP). The HWICAP core has two interfaces: one for communicating with the user's logic implemented on the logic layer and the other for manipulating the bitstream stored on the configuration memory layer via an internal configuration access port (ICAP). The HWICAP can only configure parts of the circuit which are not directly driving its signals.

This research prolongs system life by first attempting to resynchronize corrupted memory content with the current system state and secondly replacing faulty memory modules with redundant ones. Hence, contributing to evolving hardware by evolving memory units. When the system does require reconfiguration, the system can continue producing new redundant modules whilst there is available non-defective logic in the Partial Reconfiguration Region. This research aims to design an evolvable circuit which can autonomously modify its configuration memory layer to achieve self-repair and self-replication functionality. This means it will need to perform almost the entire design flow by itself, from generating a netlist, mapping it with the current reconfigurable platform, checking the design rules, placing the actual components and routing the interconnections.

The proposed architecture utilizes hardware redundancy (a voting circuit) to prevent errors inside the under-protected module from entering the next circuit stages as shown in Figure 3. Faulty modules are removed from the system and replaced by newly created duplicates through the repair process. Thus the redundant FPGA resources are no longer fixed for each under-protected module—rather they share resources among them in order to increase the efficiency of resource utilization. Moreover, the duplicating process does not require a complete implementation of the FPGA design flow but a much simpler version—copying a partial bitstream to an alternate location or by swapping bitstreams. This technique is simple and fast to implement and maintain efficiency in comparison to traditional hardware fault tolerance methods.

HWICAP is no longer needed on a virtual FPGA since the virtual configuration memory layer has no direct connection with the actual one and can only be configured via an internal configuration access port (ICAP) (Dhanasekaran, Bagan, & Ravi, 2006). Thus, a virtual internal configuration access port (VICAP) was designed as an alternative to the HWICAP to provide access to the virtual bitstream.

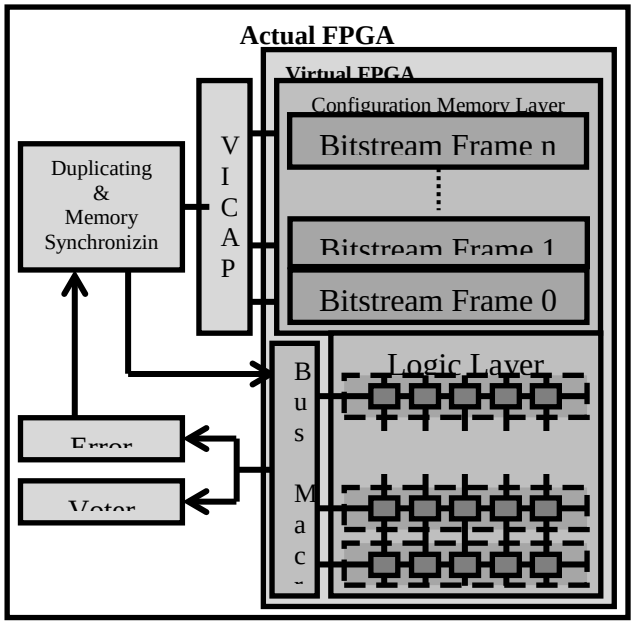

Figure 3. Proposed architecture: Overall block diagram

The actual configuration memory layer is constructed from a set of shift-registers, in which parts of the bitstream are stored. Each shift-register is considered as one configuration frame and can store the configuration information of a unique set of CLBs and SMs on the logic layer. Since most of FPGA vendors only support reading & writing at the frame level (to reduce manufacturing cost and routing cohesion) the HWICAP core does not read & write to individual registers but to frames (Partial Reconfiguration - Professor Workshop, January 2008; Xilinx, 2006). The VICAP was designed to mimic this frame based functionality.

The logic layer performs the normal operations of the FPGA. Bus Macro monitors the logic layers output and a voting algorithm or an error function determines a fault. A fault is first repaired by resynchronizing the logic layer memory components. If a fault continues then a replacement of the logic is attempted by creating new logic from the logic layer. The Bus Macro finds available resource to recreate the logic and uses it output history to perform memory synchronization of the replaced logic memory units. For example, if an error occurs inside a 3-bit counter which is currently having an output value of five (101b), and the self-repair / self-replicate process taking the next five clock cycles to be completed, then after the faulty modules have been replaced the counter outputs should be two (010b), which will actually be unknown since the newly created modules do not know their previous state or would more likely to be reset to their initial states. This output is actually fault-free in the counter level, but is seen as an error in the system level, thus will trigger another self-repair/self-replicate process and potentially creates an infinite loop if left untreated. Most hardware fault tolerance systems execute memory synchronization processes after replacing faulty modules to keep the new ones up-to-date such as Bus Cycle Level Synchronization, Memory Mirroring, Message Level Synchronization, Checkpoint Level Synchronization, and Reconciliation on Takeover (Hardware Fault Tolerance and Redundancy).

The basic principle of memory synchronization is copying the current state of known-to-be-good flip-flops to other need-to-be-synchronized ones. In order to do so, a fault-free module must act like a memory source for updating the internal flip-flops of the newly created one. Moreover, while the synchronizing process is running, the source module must continue its designed functions and update its own flip-flops to guarantee its credibility. As the proposed architecture utilizes a voting process with three candidates, a synchronization cycle having

three operating cases is set up to determine which module will act as a memory source. Figure 4 shows the synchronization cycle where each candidate is synchronised against the preceding candidate.

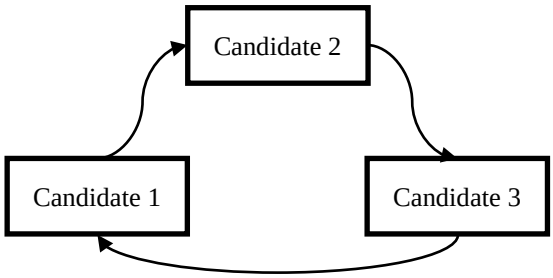

Figure 4. The circle of synchronization

The proposed self-repair function is described in Figure 5. Initially, the under-protected module remains in the 'Idle' state when operating in the fault-free condition. When faults are detected in one of its three candidates, the self-repair controller will attempt to repair the faulty candidate by synchronizing its memory several times. In case faults are caused by physical defects, which cannot be repaired by memory synchronization, the repair attempts will fail continuously, triggering the self-repair controller to give up and switch to self-replicate option instead. After the faulty candidate has been duplicated, the self-repair controller will put the system back into the idle state and continue to watch the consistency of the three candidate output signals.

The newly created candidate output signals still appear to be incorrect since its internal flip-flops are out of sync with the others, forcing the self-repair controller to send out a memory synchronization attempt to repair the newly created candidate. This time the repair attempt will succeed (assuming the newly created candidate doesn't contain physical defects), which will put the under-protected module back into its initial condition, the idle state. The whole self-repair process is executed in the background and will not corrupt the under-protected module function since it only operates on the faulty candidate. The output signals of the other two candidates will contribute to the voting process thus masking out errors on the faulty one, keeping the under-protected module output signals error-free.

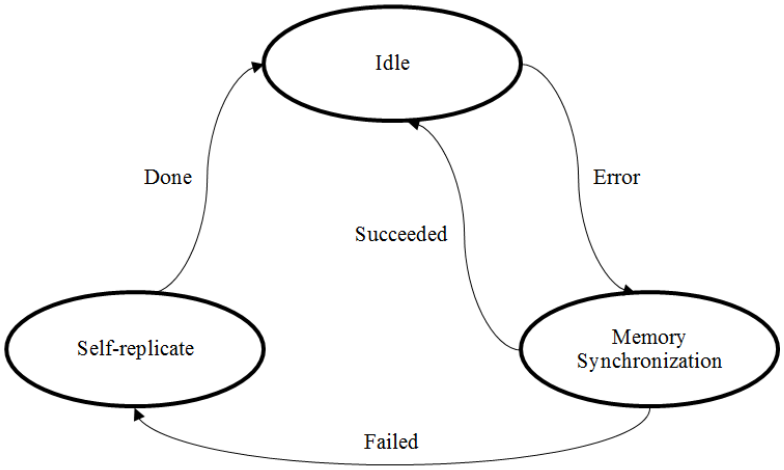

Figure 5. The self-repair function of the proposed architecture

Configuration Memory Layer

The elemental structure of the configuration memory layer is based on an array of shift-registers. There is no direct access to the individual flip-flop within each frame. The

configuration data can be shifted into each frame independently. Each bit frame stored the architecture bits of one row of CLBs and SMs. Thus, the number of architecture bits required for each CLB and SM are nCLB and nSM respectively. With m pairs of CLBs and SMs in each row of the logic layer, we can calculate the length of each bit frame using Equation 1:

$$n_{Frame} = m \times (n_{CLB} + n_{SM}) \quad (1)$$

Bus Macro

The Bus Macro connects the inputs and outputs of the logic layer to the voter and the controller module as shown in Figure 7. The real value of the Bus Macro is it can reconfigure signals from the logic layer to the voter and controller module to perform hardware fault repair of the logic layer. Hence, a newly created replacement module can be connected directly to the voter and controller modules.

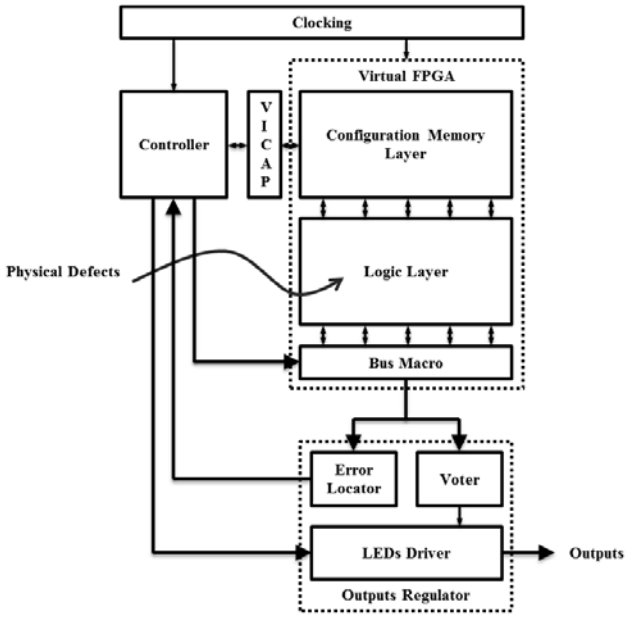

Figure 6. Proposed architecture: Detailed block diagram

Table 1 shows the components' functional description of the proposed architecture as shown in Figure 6.

Table 1. Proposed Architecture: Components' Functional Description

Component		Functional Description
Virtual FPGA	Configuration Memory Layer	An array of shift-registers storing configuration data (bitstreams). Each shift-register is considered as a segment (or frame), which stores the bitstream of a particular row of CLBs and SMs in the logic layer.
	Logic Layer	An array of CLBs and SMs (having a ratio of 1:1), whose functionality and connectivity are defined by the corresponding segments of bitstream stored in the configuration memory layer.
	Bus Macro	Provide universal accesses for three candidates' IO signals.
Outputs Regulator	Voter	Send out the dominant value of the output signals produced by the three candidates.

	Error Locator	Send out the index of the candidate whose outputs are different from the other two.
	LEDs Driver	Convert the output signals to displayable values for the LEDs and Seven-Segment Displays (SSDs). Also send out the status of the controller core for debugging.
Controller		Control the self-repair and self-replication process, direct the bitstream read/written from/to VICAP core, and manage the current available resources.
VICAP		Read/write from/to a specific segment of configuration memory layer.
Clocking		Provide two clock signals: one for system's functions and another for reconfiguration processes.

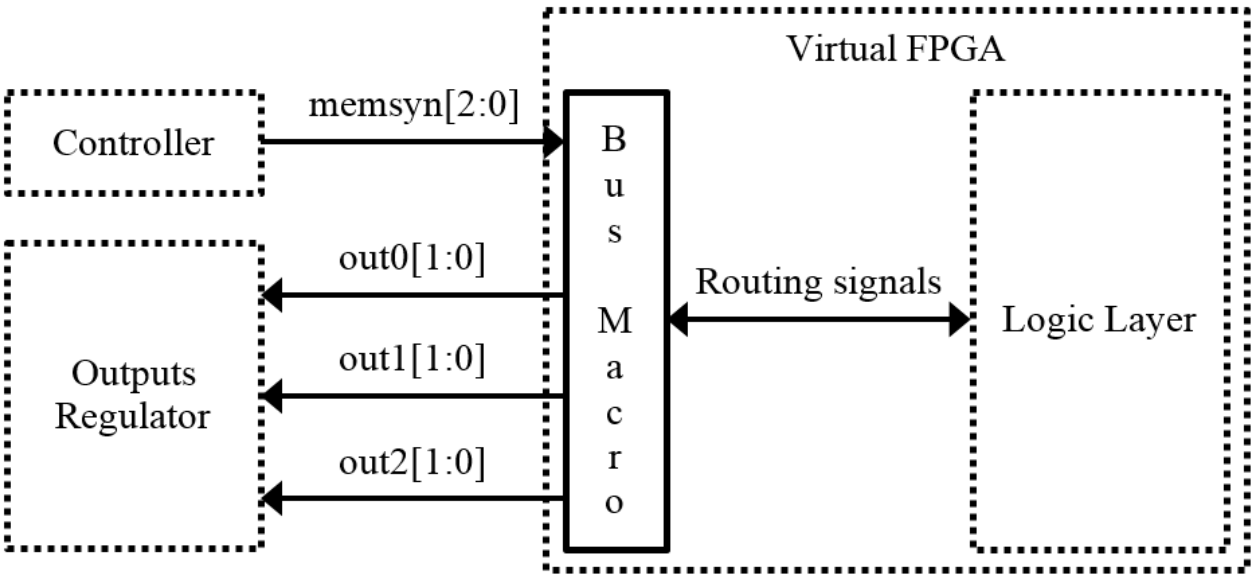

Figure 7. Bus Macro: block diagram

Memory Synchronization

The synchronization of memory occurs after an error is detected or a hardware reconfiguration process is finished. The fundamental principle of memory synchronization is creating a loop where each candidate can access the input stages of another candidate's internal flip-flops. The internal flip-flops of the candidate which is being synchronized will then be driven by the combinational circuits of another candidate, which is currently up-to-date, thus memory synchronization is completed and errors due to memory mismatches will be eliminated. The logic structure of memory synchronization is shown in Figure 8.

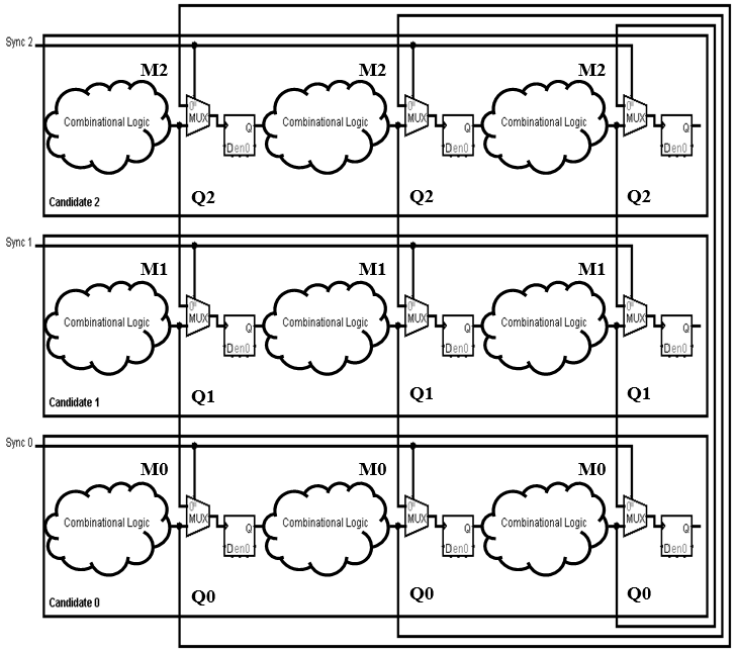

Figure 8. The logic structure of memory synchronization

Though it is rather simple, the implementation of such a structure requires an efficient design of universal connection channels since each candidate can be moved to other parts of the logic layer and loses its connection with the referenced module. Our solution is to create specific channels which spans across the logic layer to provide universal connections for each internal flip-flop as shown in Figure 9. Note that this particular channel can be left floating during the reconfiguration process without the possibility of corrupting other candidates' operation. This critical characteristic differs from the bus macro presented in REPLICA project (Kalte, H.; Lee, G.; Porrmann, M.; Ruckert, U., 2005) and enables the flip-flop memory to be reconfigured to its good-known-current-state.

Each of the four combinational logic blocks (M0, M1, M2, Free) are located within a different cell of the macroblock. The flip-flops are connected to each other via the channel (Q0, Q1, Q2, free) and the Bus Macro. After the frame is swapped (repair) the synchronization order changes to maintain the current state.

M0, M1, M2 are identical cloned combinational logic blocks. They have been implemented as counters; a simple logic and memory device. Each counter as a whole acts as a candidate to a voter algorithm to choose the output and detect logical mismatches. Logic blocks can be replaced by reconfiguring their PR block and using another combinational logic within the PR block. However, the fundamental problem is what value the flip-flop should be after reconfiguration. This is solved by connecting two flip-flops that are located in the same position and perform the same operations but in different candidates, via the memory channel. There is one channel (Q0, Q1, Q2, Q3) for every flip-flop in the candidate.

For example, Q0 connects M0 and M2, Q1 connects M0 and M1, etc. The memory channels are fixed and do not change during reconfiguration. So M0 will always be connected to Q0 and Q1. When one candidate needs to be repair, the new flip-flop selects the starting value from the other candidate flip-flop via the memory channel. If two candidates are found to be defective, the known-to-be-good flip-flop value can be found by tracing the memory channel as in Figure 9. The memory channels are connected in a cycle to avoid the last-link in the chain syndrome; which doesn't have a candidate to synchronize its flip-flop memory. Hence, system failure finally occurs when there is no more combination logic available and when all flip-flop candidates are dysfunctional.

Finite State Machine Diagram

Figure 10 shows the Controller's FSM having four states: IDLE, Memory Synchronization, Self-Replicate and System Failure. These states control the whole self-replication and memory synchronization process by monitoring the error indicator signals from the Error Locator core. If errors occur at the second candidate, the Error Locator core will generate an error indicator signal telling the Controller to duplicate the second candidate then synchronize its memory by using internal flip-flops of the first one as references.

One important assumption of the Controller is that there must be only one candidate malfunctioning at one time (otherwise the Error Locator will not detect the error's sources correctly). This is a crucial characteristic of the voting process. However, the chance of errors occurring in two or more candidates of the voting system at the same time is extremely low making this assumption acceptable.

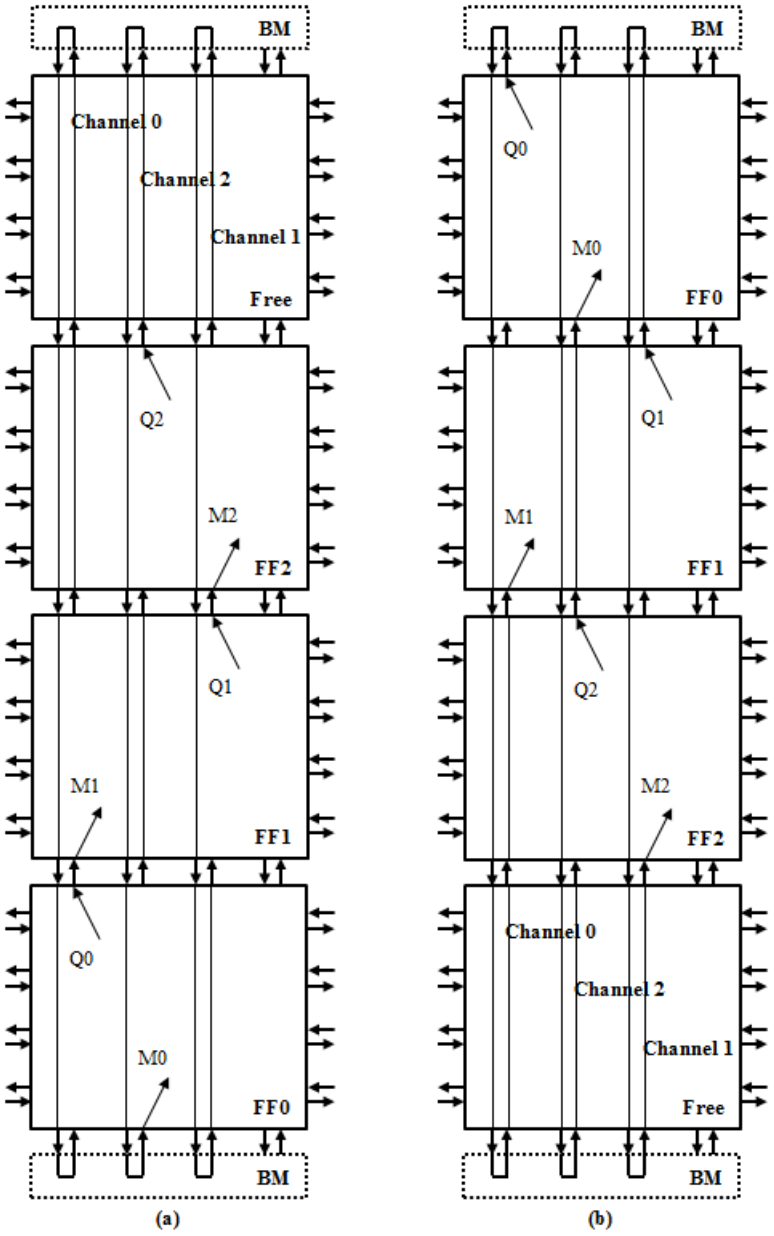

Figure 9. A synchronization cycle: synchronization order changes to maintain current state. a) Before b) after frame swapping

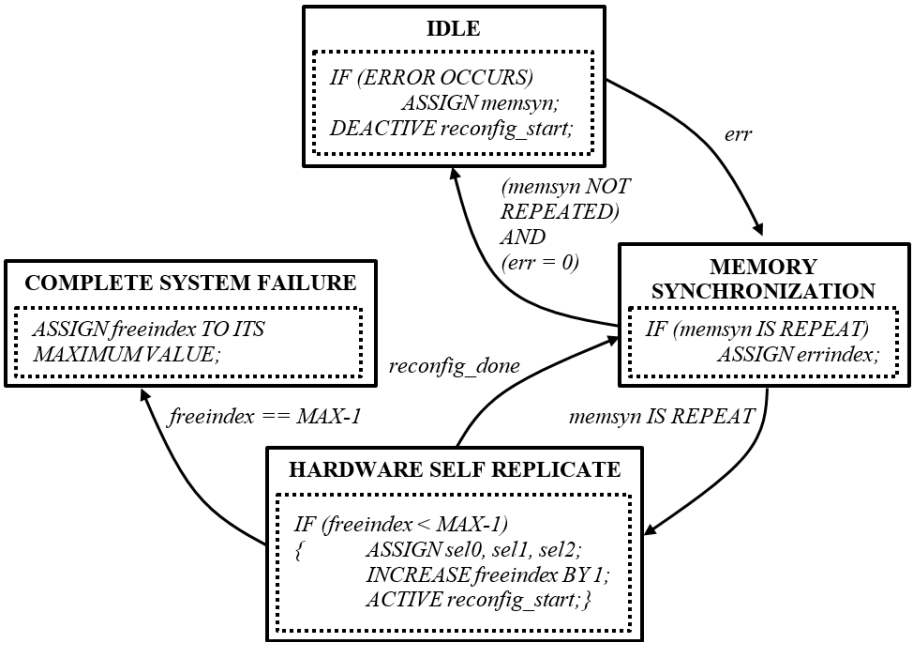

Figure 10. The controller's FSM

Results

Simulation Verification Results

The functional behaviour of Virtual FPGA were tested by simulating its elemental components from the individual to combinational level.

The testing environment for a CLB is shown in Figure 11, in which out1[1] feedbacks to in1[0] while other signals are controlled / monitored by a test-bench module. The CLB's configuration memory is loaded with a 38-bit configuration word of 0x0800000035, which defines its function as a single T-Flip-Flop. We expect the state of out1[1] to be flipped at each rising edge of clk. Signals have been omitted from Figure 11 as they are unrelated at this point.

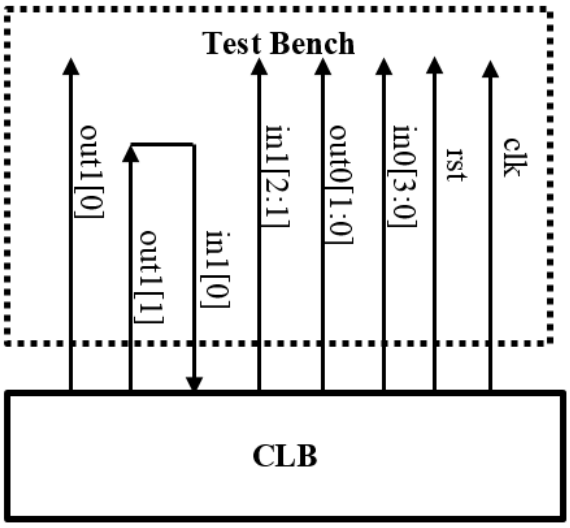

Figure 11. CLB's test bench

Figure 12 shows the CLB's IOs during behavioral simulation using Modelsim SE 6.5. The observed waveform is as expected.

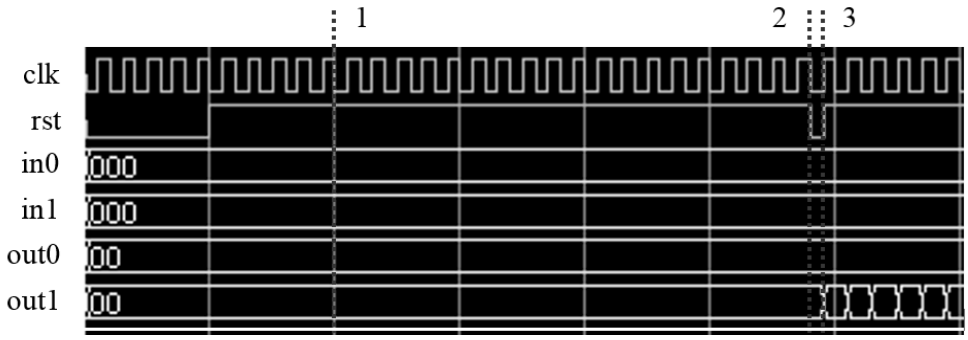

Figure 12. The CLB's waveform

In Figure 12, prog goes low, indicates that the configuration data are being shifted in. When rst goes low the contents of the internal flip-flops are cleared. When rst goes high the CLB starts to function as a flip-flop. In the previous testing environment out1[1] and in1[0] are directly connected together through the test-bench module rather than the SMs. Thus in this test, these signals will be routed through two Multiplexor based Switch Matrixes (MSMs). The MSM0 and MSM1 configuration memories are loaded with two 88-bit configuration words: 0xff8fffffffffffffffdf and 0xffffffffffffffffcfff respectively, while the CLB0 and CLB1 configuration memories are loaded with two 38-bit configuration words: 0x0800000035 and 0x0000000000 respectively. We would expect the state of the CLB0's out1[1] to be flipped at each rising edge of clk.

The observed results are precisely the same with the previous test validating the data integrity of the fault tolerant system.

On-Board Verification Results

This research utilizes a Virtex- II™ V2MB1000 Development Board as the target hardware platform for on-board verification. Table 2 lists the signal's description of testing environment. Special signals were created to generate physical defeats for testing purposes which directly control an internal switch inside each logic frame (total of six). When active, these switches will act like an open connection (physical defect) in the logic frame. LEDs give an indication of a voting mismatch (error detected between modules). 7-Segment displays indicate display resources and module outputs as described in Table 2.

Table 2 On-Board User Peripherals

On-board Component	Description
User Push Button Switch (SW5)	Global reset.
User LED	Error LED, whenever turned ON indicating errors were detected.
User 7-Segment Display (DD1)	Display outputs of the under-protected 2-bits counter.
User 7-Segment Display (DD2)	Display number of available resources.
User DIP Switch [1:6] (SW4)	Control an internal switch inside each frame (total of six) to generate open circuit defects.

Our system performance was as expected with errors generated by the switch presses being masked out by voter logic facilitating a safe, functional repair cycle. The cycle of the 2-bit counter was allowed to count and cycle through.

Once a fault was added into the system (by toggling a switch) the controller status LED illuminated indicating that a memory resynchronization operation was being performed. After a short time the system was then observed to enter the self-replicate state (as resynchronization fails to permanently fix the error). The system was observed to entering the hardware repair state followed by a relatively brief memory synchronization state before returning to the idle state with no error reported by the voter circuit. Throughout this error-inducing time the counter continued counting as normal.

This procedure of introducing errors was repeated several times with identical behavior until resources for reconfiguration were exhausted. When this occurs the system was observed to enter the system, failure state– indicating that no more fault tolerance can be provided and that the system needs to be completely replaced. When additional errors were then added the counter began to malfunction as would be expected with insufficient resources to provide redundancy in the operations.

The 2-bit counter can survive up to four physical defects out of three available frames otherwise the synchronization failed due to no known-good-state remaining, hence demonstrating the effectiveness of the proposed fault tolerant approach in hardware.

Evaluation

This research has contributed to self-healing hardware fault-tolerant design technique in four ways:

Firstly, by demonstrating a complete hardware fault-tolerant system is capable of guaranteeing the correctness of several under-protected modules until the system runs out of available resources. The static part of system which cannot be replicated is kept simple and has a reasonably smaller size in comparison with the dynamic one, thus can be protected physically during fabrication process.

Secondly, our system has the potential to maximize the efficiency of resources utilization in hardware redundancy since the redundant elements are considered as CLBs and SMs instead of fixed modules allowing them to be shared freely between different under-protected modules.

Thirdly, we demonstrate another possible way of locating the Bus Macro so that it will not span across the under-protected modules. Thus, preventing modules undergoing repair from disrupting the rest of the system even when glitches occur during partial reconfiguration process.

Finally, we introduce a new channel-based memory synchronization structure, in which the internal flip-flops are kept up-to-date through the synchronization cycle. This technique allows the proposed architecture to protect multiple sequential circuits.

Conclusions

The proposed design has demonstrated four key fault tolerant system architecture achievements, namely: detection of internal errors at runtime, resynchronization of modules to mitigate and repair temporary errors, relocation to replace faulty modules and voting circuitry to maintain data integrity. The system was successfully verified on a virtual environment and on a Virtex-II FPGA. Such fault tolerant systems result in the next generation of building real fault-tolerant applications which are capable of self-healing thus achieving the highest possible reliability.

Acknowledgement

We gratefully acknowledge RMIT Vietnam, and this research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.02-2012.27.

References

[1] C.P. Dingman, and J. Marshall, "Measuring robustness of a fault-tolerant aerospace system," In: *FTCS 95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, Washington D.C., United States, 1995.

[2] T. Jafari, F. Dabiri, P. Brisk, and M. Sarrafzadeh, "Adaptive and fault tolerant medical vest for life-critical medical monitoring," In: *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, New York, United States, 2005.

[3] L. Moser, and M. Melliar-Smith, "Demonstration of fault tolerance for corba applications," In: *ARPA Information Survivability Conference and Exposition*, Vol. 2, pp. 87, 2003.

[4] T. Anderson, and P.A. Lee, *Fault Tolerance - Principles and Practice*, Springer-Verlag, New York, United States, 1992.

[5] W.C. Carter, "Hardware fault tolerance," In: *Resilient Computing Systems: Vol. 1*, T.Anderson, ed.: John Wiley & Sons, Inc., New York, United States, pp. 11- 63, 1986.

[6] J.L. Trahan, "FPGA background," *Department of Electrical & Computer Engineering, Louisiana State University, EE7700: Course on Run-Time Reconfiguration*, 2005 [Lecture Notes].

[7] "Evolvable Hardware Systems: EHW Overview," (n.d.) 2004. [Online]. Available: http://unit.aist.go.jp/asrc/asrc-5/en/overview.html [Accessed: May, 2012]

[8] T. Higuchi, Y. Liu, M. Iwata, and X. Yao, "Introduction to evolvable hardware," In *Evolvable Hardware*, T.Higuchi, and X. Yao, eds.: Springer, New York, pp. 1-17, 2006.

[9] G.W. Greewood, D. Hunter, and E. Ramsden, "Fault recovery in linear systems via intrinsic evolution," In: *Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware*, Vol. 1, pp. 59-65, 2003.

[10] D. Mange, A. Stauffer, G. Tempesti, F. Vannel, and A. Badertscher, "Bio-inspired computing machines with artificial division and differentation," In *Evolvable Hardware*, Springer, Stanford, California, United States of America, pp. 85-98, 2006.

[11] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems," *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, pp. 83-97, April 1997.

[12] F. Ferrandi, M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," In: *International Symposium on System-on-Chip*, 2006.

[13] S. Corbetta, F. Ferrandi, M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto, "Two novel approaches to online partial bitstream," In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, IEEE Computer Society, Washington, D.C., United States, pp. 457-458, 2007.

[14] J. Note, and E. Rannaud, "From the bitstream to the netlist," In: *Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ACM, Monterey, California, United States, pp. 264-264, 2008,

[15] C.J. Morford, *BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs*, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2005.

[16] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework," In: *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 37-44, 2012.

[17] A.A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An open-source partial-reconfiguration toolkit for xilinx FPGAs," In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 228-235, 2011.

[18] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapidsmith: Do-it-yourself CAD tools for xilinx FPGAs," In: *Proceedings of the 21th International Workshop on Field-Programmable Logic and Applications (FPL'11)*, 2011.

[19] J. Rose, J. Luu, C.W. Yu, O. Densmore, J. Goeders, A. Somerville, K.B. Kent, P. Jamieson, and J. Anderson, "The VTR project: architecture and CAD for FPGAs from verilog to routing," In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, California, United States, 2012.

[20] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A bitstream manipulation filter for module relocation in partial reconfigurable systems," In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IEEE Computer Society, Washington, D.C., United States, pp. 151.2-, 2005,

[21] T. Becker, W. Luk, and P.Y.K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," In: *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, pp. 35-44, 2007,

[22] M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto, "Core allocation and relocation management for a self dynamically reconfigurable architecture," In: *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, IEEE Computer Society, pp. 286-291, Washington, D.C., United States, 2008.

[23] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu, "Partial bitstream 2-D core relocation for reconfigurable architectures," In*: Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, IEEE Computer Society, Washington, D.C., United States, pp. 98-105, 2009.

[24] "Partial Reconfiguration - Professor Workshop," *Xilinx University Program*, India, January 2008, [Workshop].

[25] G. Hollingworth, S. Smith, and A. Tyrrell, "Safe intrinsic evolution of virtex devices," In: *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware*, pp. 195-202, 2000.

[26] P. Haddow, and G. Tufte, "Bridging the genotype-phenotype mapping for digital FPGAs," In: *the Third NASA/DoD Workshop on Evolvable Hardware*, California, 2001.

[27] Y. Zhang, S. Smith, and A. Tyrnell, "Digital circuit design using intrinsic evolvable hardware," In: *6th NASA/DoD Workshop on Evolution Hardware*, Seattle, United States, 2004.

[28] D. Dhanasekaran, K.B. Bagan, and S. Ravi, "Fault tolerant system design using evolved virtual reconfigurable circuit," *IJCSNS International Journal of Computer Science and Network Security*, Vol. 6, pp. 64-72, 2006.

[29] Xilinx, *DS280: OPB HWICAP (v1.00.b), Product Specification*, Xilinx, 2006.

[30] EventHelix.com Inc., "Hardware Fault Tolerance and Redundancy," (n.d.) [Online]. Available: http://www.eventhelix.com/RealtimeMantra/HardwareFaultTolerance.htm [Accessed: 10 September 2012]

[31] G.R. Clark, "A novel function-level EHW architecture within modern FPGAs," In: *Proceedings of the Congress on Evolutionary Computation (CEC 99)*, Washington D.C., 1999.

[32] R. Lysecky, K. Miller, F. Vahid, and K. Vissers, "Firm-core virtual FPGA for just-in-time FPGA compilation," In: *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Monterey, California, United States, 2005.

[33] G. Tempesti, D. Mange, and A. SASEANtauffer, E A Sngineering Journal,elf-Repairing Vol 5 No 1 (2015)Multiplexer-B, ISSN ased2229-127X FPG p. 54 A Inspired by Biological Processes, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[34] A. Stauffer, D. Mange, G. Tempesti, and C. Teuscher, "BioWatch: A giant electronic bio-inspired watch," In: *The Third NASA/DoD Workshop on Evolvable Hardware*, 2001.

[35] P. Haddow, G. Tufte, and P.V. Remortel, "Evolvable hardware: Pumping life into dead silicon," In: *On Growth, Form and Computers*, London, Academic Press, 2003.

[36] Y. Zhang, S. Smith, and A. Tyrnell, "Digital Circuit Design using Intrinsic Evolvable Hardware," In: *6th NASA/DoD Workshop on Evolution Hardware*, Seattle, United States, 2004.

[37] N. Macias, "Ring around the PIG: A parallel GA with only local interactions coupled with a self-reconfigurable hardware platform to implement an O(1) evolutionary cycle for evolvable hardware," In: *Congress on Evolutionary Computation*, Washington D.C., 1999.

[38] G. Tufte, and P.C. Haddow, "Prototyping a GA pipeline for complete hardware evolution," In: *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware 1999*, Pasadena, California, United States, 1999.

[39] M. Murakawa, S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi, "The GRD chip: Genetic reconfiguration of DSPs for neutral network processing," *IEEE Transactions on Computers*, Vol. 48, pp. 628-639, 1999.