Discussion of nMR-based Approaches for Fault-handing in SRAM-based FPGA Devices

Francis Luna

Abstract—**Triple Modular Redundancy (TMR) is the most well known technique for fault tolerance in Field Programmable Gate Arrays. This paper will discuss some different novel approaches for the application of redundancy to FPGAs. Architectural and high-level approaches are investigated that reduce power and area overhead for both Triple and Dual Modular Redundancy. All techniques are compared to show both advantages and disadvantages of their respective approaches.**

Index Terms—**FPGA, fault tolerance, fault handling, partial reconfiguration, triple modular redundancy (TMR)**

I. Introduction

SRAM-based Field Programmable Gate Arrays (FPGA) allow for the ability to program any function into a device. The purpose of the device may change over time and require new functions to be incorporated into it. In FPGAs all logic and routing elements can be reconfigured to meet the system's needs. The FPGA allows for different configurations to be loaded into it. This is done by loading different bitstreams that program these elements to their desired need.

FPGAs then inherently have the ability to handle faults because they can load configurations that do not use the faulty component and work around the fault. This is an important feature for space applications where human intervention is infeasible [1]. However, such devices employed in space also have to consider the effect of higher radiation and extreme environmental conditions. Single Event Upsets (SEUs) are therefore a main source of concern. An SEU happens when a charged particle strikes the FPGA and flips a bit in a memory cell or permanent damage is caused to the silicon due to depleting oxide layers or other environmental effects. These SEUs can be either transient or permanent. Transient faults will eventually go away over time when that memory cell gets rewritten. Permanent faults, on the other hand, will remain in the device for the remainder of the mission and action must be taken in order to avoid mission failure. Such permanent faults are stuck-at faults where a bit or line will remain a 0 or a 1 for the rest of the device's lifecycle. These SEUs can affect any portion of the FPGA. If they affect the LUTs of the FPGA they can change the logic function of that cell. This change will remain in the system until the LUT's contents are either reloaded to realize the same function or changed. They can also cause transient pulses in the combinatorial logic path that disrupts the data and must be run through again to get the correct output.

Several techniques have been proposed and researched to mitigate the affect of these faults when they happen on an FPGA. They can be classified as either architectural or high-level techniques [10]. Architectural techniques include hardening the memory cells of the FPGA to make them less prone to faults caused by the environment. This can be seen in such commercial products as the Xilinx QPro FPGA family of devices [2]. Other architectural techniques may include the way the logic cells are placed on the FPGA fabric or altering the components of these cells to provide additional fault tolerance support. Most techniques however are high-level and implemented by the user. High-level techniques do not require any modification of the physical FPGA architecture and as such are more attractive to the user because they can be implemented on any device. However, a simple change in the architecture could provide more efficient high-level techniques.

Most all high-level techniques use some form of redundancy. This redundancy can either be in the form of multiple instances of a function running in parallel where one output is then chosen to be reciprocated, or a spare part, either hot or cold, is waiting to be enabled and carry out its function. One of the most well known high-level techniques is Triple Modular Redundancy (TMR) as seen in Figure 1.

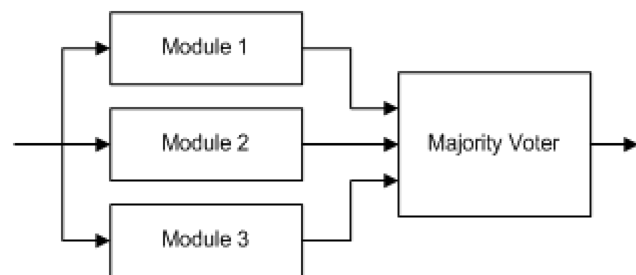

Figure 1: Basic Triple Modular Redundancy

TMR provides several benefits as well as drawbacks over other fault handling methods. Some of the benefits include extremely low detection latency, and assuming a single fault condition, it can mask the fault from propagating to the rest of the system. TMR does however come with 3x the area and power overhead. TMR can either be applied on a very fine granularity where each LUT is triplicated, or on a very coarse grain where there might be three modules doing the same function as in Figure 1. A voter is implemented to pick one of

the three outputs that are non-faulty. If one output does not match the other two, then that module is faulty and one of the other two outputs is chosen to be passed along. Further action can then be taken to correct the faulty module.

Dual Module Redundancy (DMR) or Concurrent Error Detection (CED) is similar to TMR but only duplicates the module to detect when a fault has occurred. The power and area overhead of such a method is 2x. However, unlike TMR this method only detects whether or not a fault has occurred and further action must be taken in order to recover from the fault since it cannot mask it.

In this paper both architectural and high-level nMR approaches will be discussed that provide a suitable level of fault tolerance for many applications. Vigander's work with FPGA evolution and voting [3], DeMara's work with Competitive Runtime Reconfiguration (CRR) [4], Abramovici et al.'s work with roving Self-Testing Areas (STARs) [5], Al-Haddad et al.'s work with the Reconfigurable Adaptive Redundancy System (RARS) [6], Kyriakoulakos et al.'s work with a new architecture for efficient TMR fault tolerance support [10], and Lahrach et al.'s Master-Slave TMR inspired technique [11] are the multiple works discussed in detail in this paper.

II. Vigander's Approach

A. Introduction

Vigander established very interesting results in [3]. He created his own genetic algorithm (GA) and simulator to base his experiments off of. His simulator was of an FPGA that had different constraints than that of a real FPGA. More specifically, routing restrictions were put in place to create a strictly feed-forward network. This was done in order to greatly simplify the simulator.

A 4-bit multiplier was the application he realized on his simulator for his experiments. Each multiplier configuration was tested exhaustively. All possible input combinations were compared with the output they produced. There are a total of 256 ($2^{(4+4)} = 256$) different input combinations. This was done in order to produce a fitness value. The higher the fitness value, the more functional the configuration is; it outputs more correct values given an input. The fitness value therefore is the total number of correct outputs out of 256.

Three different genetic operators were used in his genetic algorithm: crossover, mutation, and cell swapping. Crossover, he states, is when cells are inherited from one parent and then modified to create a new individual. Mutation is when a cell is randomly changed in each configuration. Cell swapping is when two cells in the configuration are swapped. This exchanges both the function and inputs of these cells but invalidates the feed-forward property when the later cell is swapped to a place earlier than it was previously. To overcome this he picks new random inputs for these cells. Elitism is also used in all his experiments in order to keep forward progress in the evolutionary process by guaranteeing that the fitness of the most fit individual is monotonically increasing.

B. Experiments

Three different groups of experiments were performed. The first group of experiments attempted to repair a single random stuck-at fault using his GA. The result was that it proved difficult for the GA to come up with a perfect repair (256 fitness value) with the faulty cell after many generations. This was further extended by providing the GA more cells to increase the search space and provide more space for the GA to avoid the faulty cell in the configuration. The result was that it had no effect for the GA finding a higher fitness individual after many generations. He also let the GA run for a long period of time to see if the fitness would improve. The result of this experiment was that no high fitness was achieved, but improvement in the fitness was shown very late in the run.

The second group of experiments consisted of accepting individuals that are not perfectly repaired. Vigander found that after many generations, different configurations fail differently from one another given the same input. He used this result to extend his experiment and created a system of three imperfect FPGAs that would then vote on each output. Important to note here that each FPGA used had a different fault so that no two fail the same way. He observed that even with individuals that do not have 100% fitness, when these individuals vote they can achieve a completely correct result. This can be seen in Figure 2, the dotted lines are the fitness of the imperfect FPGAs and the solid line is the fitness after all three have voted. This was also in the case that all three FPGAs were rendered faulty at the same time, an extremely unlikely event.

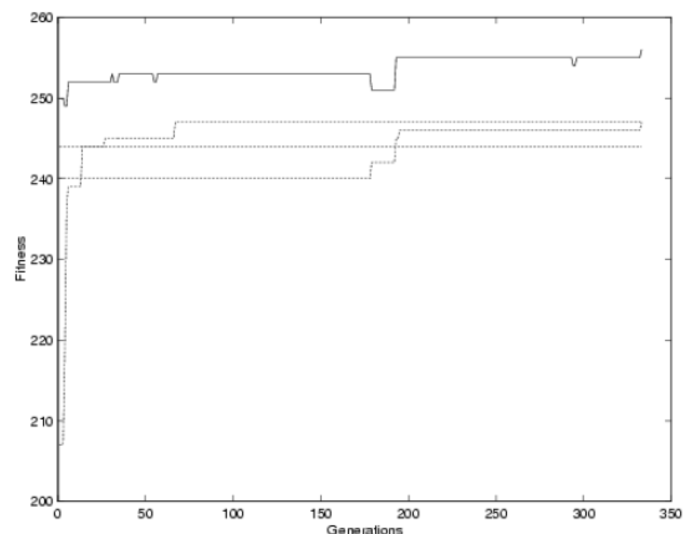

Figure 2: Results of a triplex voting arrangement of three faulty FPGAs

For the third and final group of experiments, Vigander repeated the GA several times for the same fault induced in the configuration. He found out that the same inputs cause the same outputs to fail over time. The GA was unable to fix these configurations differently every run.

C. *Insights*

Vigander's experiments brings forth many interesting results and conclusions. Interesting to note, Vigander assumes that there is some known "golden" oracle that has the correct outputs for any given input. Such an assumption cannot be realized in a real system because this oracle may obtain faults and then the system will be comparing itself to something that is wrong when it might have had the correct result to begin with.

The first experiment showed that individual configurations that already had a high fitness were very difficult to get more fit, but configurations that had a low fitness to begin with find high fitness fairly quickly. In the long run experiment, marginal improvement in fitness was achieved after tens of thousands of generations but no perfect solution was reached. This shows that the GA is continuously trying to find a new higher fitness configuration but a better fit individual may not be worth the time required to come across. The function has become asymptotic and the GA has converged to that fitness. Vigander however did not use the concept of design diversity in his experiments. The design diversity concept is one that there are many different ways for something to do the same thing. Configurations can be functionally identical, but physically distinct. He seeded the population with identical configurations. It would be interesting to see if had he used design diversity, would he have reached the same conclusion in this experiment.

In Vigander's second experiment, he uses a triplex voting arrangement similar to that of TMR. Even though all the FPGAs he used were faulty even after evolution, they still produced a correct result. Perhaps the time required to evolve these three configurations to produce a correct result could be decreased by having more than three modules vote on the correct output. For example, if five FPGAs were configured in a 5-plex voting scheme, overall perfect functionality may be reached much faster than the time required to exhaustively evaluate every configuration that is evolved. However, using a 5-plex scheme means that power and area footprints are further increased than just having a triplex voting scheme. Nonetheless, the time saved in terms of generations and evaluations is interesting to note.

Vigander's third and final experiment also reveals some interesting results. Even though the GA has randomness incorporated in it, it does not allow recovery of certain input/output pairs. After several generations, the same parts of the circuit fail. This goes to show that certain stuck-at faults cannot be worked around and in order to realize a perfect configuration, the faulty cell should not be used.

III. Competitive Runtime Reconfiguration Approach

A. *Overview*

A novel approach that uses DMR is brought forth by DeMara et al. in [4] called Competitive Runtime Reconfiguration (CRR). CRR fully exploits the ability for FPGAs to reconfigure themselves. By allowing the hardware to evolve, the amount of redundancy needed is reduced. The benefits of such an approach include recovery without increased size, weight, and power. CRR features an approach that adapts to the conditions throughout the device's lifecycle. Unlike other approaches, it does not need any test vectors for device refurbishment. Fitness is evaluated by comparing two configurations with one another. A correct "golden" configuration is not needed to make these comparisons. The basic layout of this approach is seen in Figure 3.

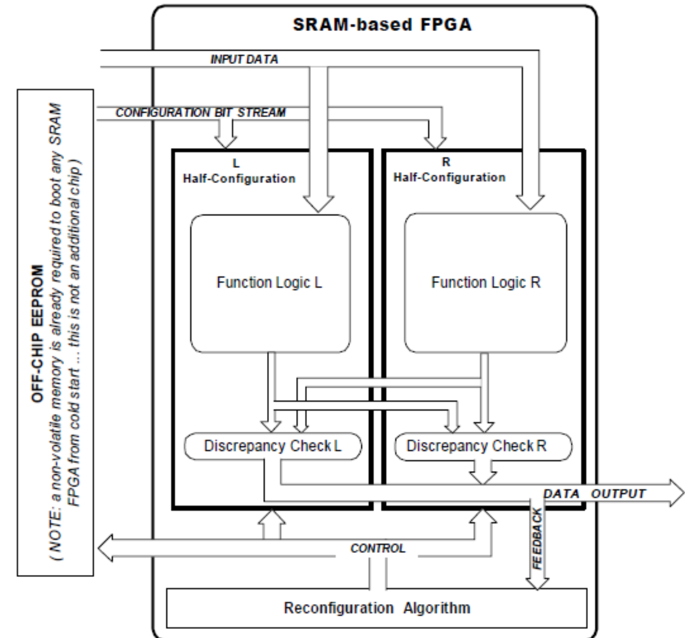

Figure 3: Tandem CRR arrangement

CRR is a very interesting approach that incorporates both competition and evolution, something that is inherent in nature, into hardware. It also allows for graceful degradation when multiple faults are encountered in the system.

This approach uses temporal voting which occurs when an alternate configuration is paired with another to vote on their outputs. This occurs at some defined rate in order to grade the fitness of this alternate configuration. Any discrepancy in either individual reduces the fitness of both individuals, and if both outputs match the fitness is raised for both individuals. Since CRR compares configurations as a whole, it does not need a fault isolation granularity. Also, fault detection happens within the FPGA and therefore has negligible detection latency.

CRR employs design diversity in its population. These functionally identical, yet physically distinct individuals are created at design time and populate the initial pool of configurations. These individuals are considered Pristine and are the highest fit individuals throughout the device's lifecycle.

The CRR voting technique works as follows. When both configuration outputs match they remain Pristine. If they are not Pristine then the fitness of both individuals is raised. When

there is a discrepancy, both of the individual's fitnesses are reduced. These individuals are now considered Suspect and will never be Pristine again. When the fitness drops below a given repair threshold, these Suspect or Refurbished individuals become Under Repair. Under Repair individuals undergo evolution for only one generation. When the fitness rises above a given operational threshold, the individual enters the Refurbished pool. This flow between states can be seen in Figure 4. A reintroduction rate is employed that allows the Under Repair individual to be reintroduced into the pool of available candidates by pairing it with another so that a fitness measurement can be taken. This reintroduction rate can be adapted to fit the required throughput of the system.

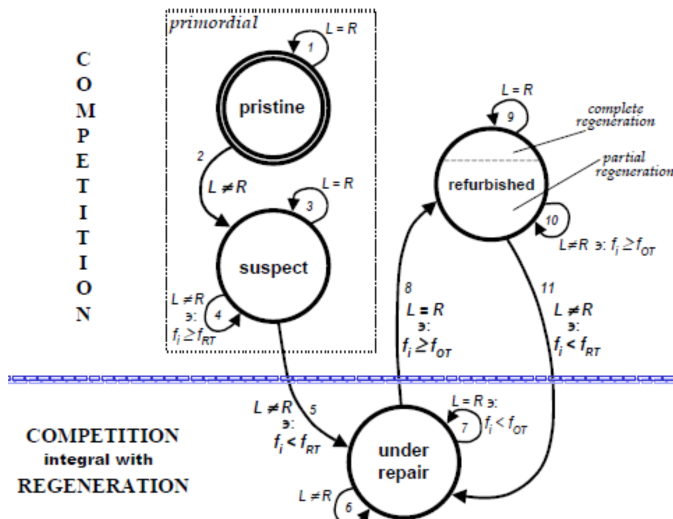

Figure 4: CRR configuration states

The genetic operators used in this CRR approach are two-point crossover, mutation, and cell swap. Two-point crossover replaces functional units with those of other good individuals. Mutation reconfigures suspect CLBs with random alternatives. Cell swap moves CLBs around within the same configuration.

Overall, CRR provides a complete fault tolerant and fault handling approach. Since the entire configuration is being compared, it addresses faults in all parts of the FPGA, including the memory, routing, and fabric itself. As seen in Figure 3, the discrepancy checking units are part of the individual configurations so that any fault in the checker is detected by the other competing configuration. This allows for one to "check the checker". Transient faults are attenuated automatically. When a transient fault occurs, its fitness will be decreased, but since it was a transient fault, and not a permanent one, when this configuration is paired up against another non-faulty one, its fitness will rise because the fault will not be present. CRR allows for the system to detect, isolate, and resolve faults without the need of any exhaustive testing while keeping the system partially online.

B. Insights

Competitive Runtime Reconfiguration is a novel approach that leverages the principle of "survival of the fittest" from evolution. CRR uses a duplex arrangement which is called the tandem arrangement. This is done in order to save space and power over a triplicate arrangement. CRR also supports a bounding arrangement in which only one configuration is run at a time. It is then compared with the next configuration loaded. By reducing the space complexity of the system, the time complexity rises. In the tandem arrangement, the system will observe a performance hit when the Under Repair individual needs to have its fitness reevaluated (caused by the reintroduction rate). Since it is unlikely that the GA will have found a solution in the first generation, both configurations will be brought down in fitness and the current inputs will have to be re evaluated again. This decreases the throughput of the system. In the bounding arrangement, the system throughput will be even lower, approximately half, because after some time a different configuration will be loaded to compare the output of the same data. This decrease in throughput can be overcome by overclocking the system enough so that even with evolved configurations being reintroduced into the pool of available candidates, the throughput can remain at 100%. This method, however, depends on the application the device is performing. The reintroduction rate, which is an upper bound of performance hit, can be modified to fit the system's needs at the time. If the system is at a mission-critical state and no degradation in performance is desired, the reintroduction rate can be set to zero. Doing so will not allow Under Repair individuals to be reevaluated and enter the pool of candidates, lowering throughput. The reintroduction rate can also be set higher during periods of the mission where the system is not doing anything critical. This allows for Under Repair individuals to get reevaluated and enter the pool of available candidates much quicker than normal. This allows for modulation of the repair rate to keep the system sustainable during its lifecycle.

Unlike TMR, this system does not provide fault masking for uninterrupted system execution. It does however extend DMR to provide an approach that allows the system to be autonomous with one-third the space savings over TMR. Also, unlike other approaches, since it leverages DMR, faults are detected instantaneously. It provides a suitable amount of fault coverage that even detects faults in the discrepancy checker. In typical nMR approaches, the voters are usually left unchecked and are therefore a vulnerable part of the system.

There are certain parts of the system left unchecked however. These are assumed "golden" and any fault in these parts may cause a fault in the entire system which any level of fault handling will be unable to overcome. As seen in Figure 3, these include any fault in the EEPROM which holds the population of individuals, and the reconfiguration algorithm. Any stuck-at fault in the EEPROM would have catastrophic behavior to the system because the configurations would be loaded and stored incorrectly. This will make it impossible for two configurations to accurately check one another. Any fault in the reconfiguration algorithm could have disastrous effects

in the system. The algorithm could be altered enough where evolution is no longer progressing but instead deteriorating. CRR however allows for graceful degradation, and such an event would not alter the current state of the system (the two configurations that are loaded). It would only be a problem once a reconfiguration happens. So if a fault in the reconfiguration logic were caught, one could just disable reconfiguration and have the system perform as is until a fault is encountered in the two competing configurations.

IV. Roving Self-Testing Areas Approach

A. Overview

The roving self-testing areas (STARs) approach put forth by Abramovici et al. in [5] allows for very fine granularity testing of a reconfigurable system. It is an adaptive computing system approach that exploits reconfigurable hardware in order to adapt to changes in the environment and its operation. This allows for new functions to be deployed on the device as well as reduced power consumption from reducing the number of parts in the system. STARs requires the use of Run-time Reconfiguration (RTR) in order to allow the system to continue functioning normally while parts of it are being reconfigured and tested.

The authors point out that traditional fault tolerant designs rely on redundant modules and voting but STARs has much smaller overhead than other such approaches. Most approaches replace faulty components with spare ones, but this method only allows for a limited number of faults to be handled before the spare resources run out. They classify fault tolerant approaches into two categories: static and dynamic. Static approaches are methods where spare resources are allocated at design-time. Having too many spares increases the area overhead, but having too little spares decreases the devices ability to handle multiple faults. Dynamic approaches allocate interconnect resources after a fault has occurred. However, spare cells are still statically allocated.

Since STARs has a high detection latency that is bounded by the physical size of the FPGA, the authors make use of CED to detect transient faults. When such a fault is detected the system rolls back to a previous checkpoint before a fault was present and runs again. If no fault is detected then it was a transient fault that has been overcome by using rollback.

STARs can be applied to any FPGA that supports RTR and does not require modification of the FPGA architecture. It is a type of Built-in Self Test (BIST) that offers exhaustive testing of all resources, both logic and interconnect. This allows it to detect dormant faults, stuck-at faults, and increase reliability. It can handle single or multiple faults in a cell and single or multiple faults in the interconnect network. This approach allows the reuse of faulty resources whenever possible by using the faulty cells for their residual capabilities. Such resources are labeled Partially Usable Blocks (PUBs) and this allows the spare capacity to increase, graceful degradation, and overall longer mission life. STARs also employs an adaptable system clock that can deal with altering the critical paths of the circuit and stopping the system clock in order to move a STAR.

The cornerstone of the STARs method is the Test and Reconfiguration Controller (TREC). This unit is an external microcontroller that controls the test, diagnosis, fault tolerance functions, configurations, and system clock. It also keeps track of which FPGA resources were declared faulty. The TREC determines when to relocate the STAR and if a fault is detected, what to do. If the detected fault is in a spare resource, it has no affect in the operational part of the system and moves the STAR to the next location. If the fault is in a resource that is used under normal operation, the TREC determines the configuration changes that are needed in order to bypass the faulty resource.

STARs works by roving a test area around the FPGA using a V-STAR and H-STAR as seen in Figure 5. This test area is independent of the working area and therefore do not have severe real-time constraints.

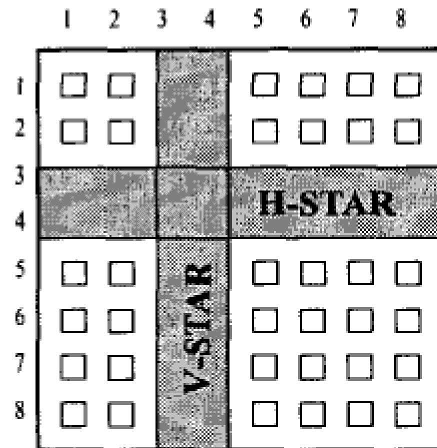

Figure 5: STARs

The basic unit of the STAR itself is the basic BIST structure (BISTER). As seen in Figure 6, it is composed of a Test Pattern Generator (TPG) that applies test patterns, two block under test or wire under test (BUT/WUT), and an output response analyzer (ORA) which reports mismatches as test failures. A STAR can contain several BISTERs. Each part of the BISTER is loaded into a different PLB such that each PLB in the tile is exhaustively tested twice. The time required to move the STAR is therefore dominated by the time required for each individual reconfiguration.

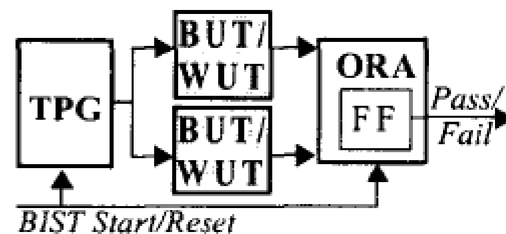

Figure 6: BISTER structure

The authors state that the faults detected are always in the STAR area and therefore do not affect the working area of the logic. They also state that fault diagnosis and fault reconfiguration do not have real-time constraints since they are not part of the working area. Overall, STARs is a novel approach that provides fine grain fault detection and handling, but relies heavily on run-time reconfiguration.

B. Insights

STARs is a BIST approach and as such handles every type of fault, even if the fault will never articulate regardless of the inputs applied. It may not be necessary to exhaustively test every portion of the FPGA if not all of it will be in use. STARs has the ability to check for dormant faults while the system is under normal operation but it may not be necessary to be checking for such a fault if that portion is never used. If this resource is selected to be used for some sort of reconfiguration then check it at that time before using it for this configuration.

The authors admit to the large detection latency and in order to overcome this disadvantage, they suggest the use of CED in order to detect a whether or not a fault is transient. This is a valid method to handle such faults. However, if CED is employed, why not only run STARs when a checkpoint rollback was executed and the fault still remains. For example, if the fault was indeed transient, it should not trigger discrepant behavior in this repeated run. If it does indeed still cause a fault, the STAR needs to proceed and find the faulty resource which may take a long period of time, especially in the worst case scenario that the fault occurred in a location the STAR recently deemed fault-free. The system would continuously be rolling back to this earlier point until the fault is detected and repaired. In the time it takes for this fault to be repaired, the system, while in an online mode, is not producing any valid output.

The authors claim that faults are always detected in the STAR area, but if the CED method is employed, the CED module will be what detects the fault. The STAR then attempts to locate the fault at a very fine granularity.

STARs consistently reconfigures the device, even in the absence of a fault, in order to detect a fault. But by doing this it may actually introduce new mechanisms for failures to occur. Nothing is checking whether the configurations are being loaded and/or stored correctly. The TREC is in control of everything dealing with this STARs method. Any failure in this device could result in catastrophic failure in the system. For example, a fault in the adaptive system clock component of the TREC means that the system will no longer function as intended. The TREC in this case must be considered "golden" and fault free. Having an entire fault handling method rely on one integral component may not be suitable. The TREC could be moved onto the FPGA by using a softcore thereby allowing some testing of itself, however the reconfiguration portion cannot be tested and must still be assumed "golden".

STARs is heavily reliant on reconfiguration and as such, is its main weakness. The time complexity of roving the STAR area is largely dominated by how quickly it takes for an area of the FPGA to be reconfigured. This is the cause for the large detection latency.

It is also interesting to note that STARs uses CED at the finest level as seen in Figure 6. Two blocks are required in order for a discrepancy to be detected. Which of the two blocks is faulty is detected by moving the elements of the BISTER around until a conclusion can be made.

V. Reconfigurable Adaptive Redundancy System Approach

A. Overview

Al-Haddad et al. design and implement a two-layered organic system called the Reconfigurable Adaptive Redundancy System (RARS) in [6]. Organic computing is a form of biologically-inspired computing that contains "self-x properties" such as self-configuration, self-reorganization, and self-healing [7, 8]. RARS does not have a predetermined level of redundancy. It can dynamically switch between different levels of redundancy to fit the mission requirements which may include power requirements. The two layers of this approach consist of a hardware layer and a software layer as seen in Figure 7. The hardware layer is implemented on a Virtex-4 FPGA while the software layer is on a host PC and consists of software that monitors the performance and status of the system, changes the level of redundancy needed, and performs recovery techniques on faulty modules.

RARS's main focus is on self-repair and self-optimization of power consumption and provides a complete software and hardware solution to handle faults. RARS allows the conservation of power by running in a low power mode and only switching on more redundant parts when needed in order to mask a fault. The power savings cause insignificant degradation in the fault tolerance of this system. However, like most fault tolerant approaches, RARS is still limited to the area left over on the device to provide additional routing and logic when it encounters a fault. This is overcome by an additional layer that observes the status of the hardware and provides active repair. There are two forms of repair employed in RARS: scrubbing [9] and evolution. Scrubbing rewrites the configuration in order to fix a simple bit-flip that may have occurred. Evolution happens when a resource encounters a more permanent fault that scrubbing will not fix.

Dynamic partial reconfiguration is also employed in this RARS approach. This allows the configuration time to be significantly reduced since the bitstream is much smaller than the original, and it allows for the system to remain online while it is being reconfigured. This method is used both when the system is performing scrubbing and when evolved individuals are being reconfigured to have their fitness evaluated.

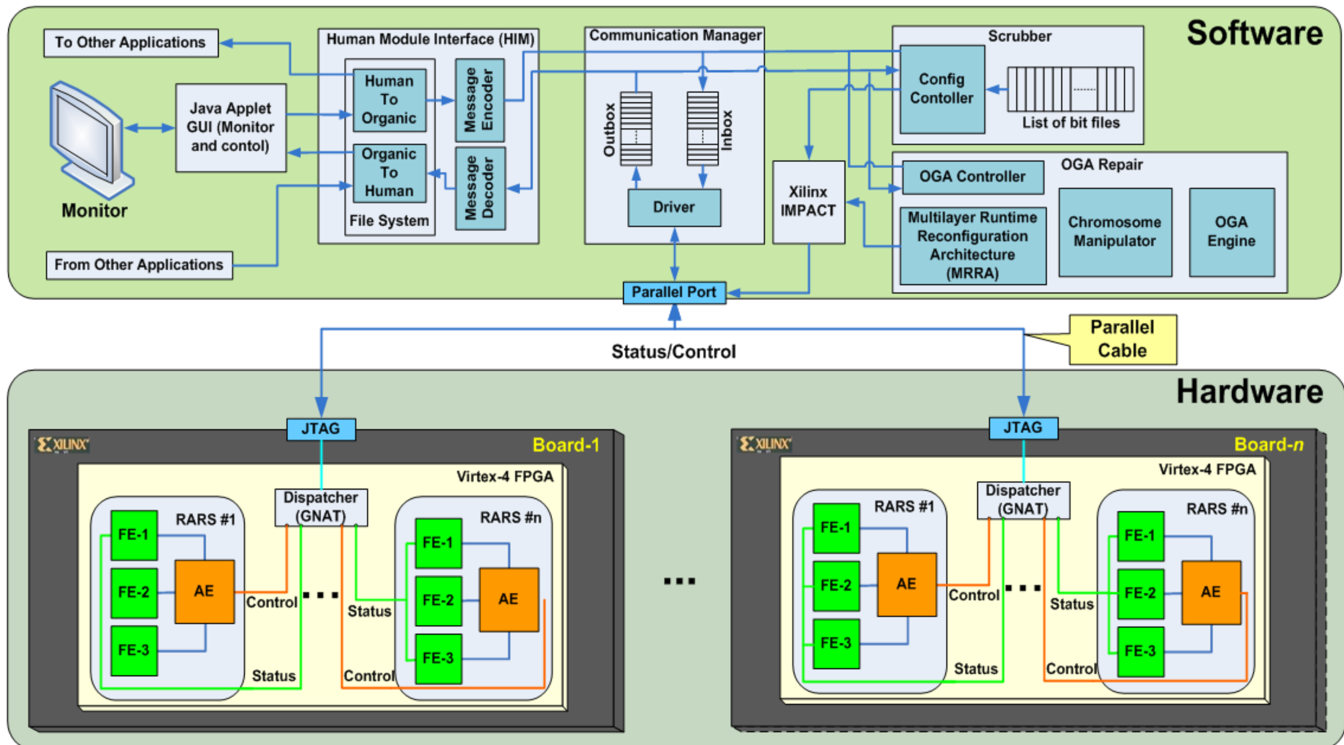

Figure 7: RARS System Architecture

RARS contains two parts: the Autonomic Element (AE), and the Functional Elements (FEs). The AE monitors the status of the redundant parts, the FEs, which are the user application. More specifically, a RARS is the smallest unit in the system and actually consists of one AE and three FEs in order to provide the adequate level of redundancy needed. The AE contains the logic to change the level of redundancy in the system and consists of five modules shown in Figure 8: the Discrepancy Sensor (DS), voter, Output Actuator (OA), Performance Monitor (PM), and Redundancy Controller (RC). The FEs are the modules that can be changed in the system. There are three FEs in order to employ a TMR arrangement.

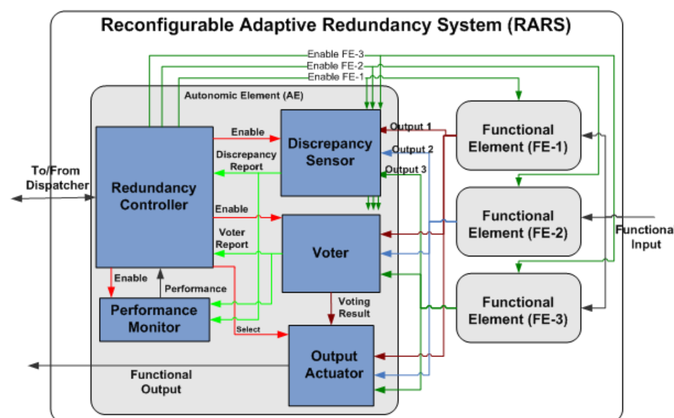

Figure 8: Reconfigurable Adaptive Redundancy System (RARS)

RARS can work in four different modes: Simplex, Duplex, TMR, and Hybrid mode. In Simplex mode, only one FE is used. The DS, voter and other two FEs are disabled. The OA is set to propagate this one output to the rest of the system. In Duplex mode, two FEs are used as well as the DS. The voter and one FE are disabled. The DS detects whether there is discrepant output between the two FEs and notifies the RC to take further action. The OA picks one of the FEs' outputs to propagate. In TMR mode, all three FEs are enabled as well as the voter. Only the DS is disabled. The OA propagates the output selected by the voter. Hybrid mode allows the RARS to switch between the other three modes as it sees fit.

When RARS detects a fault it must decide if the fault is in the data path or if it affects configurable logic. To do so, it enables a watchdog timer to see whether it is a transient fault in the data path that will fade away after an amount of time. If there is still a fault present after the watchdog timer expires, it proceeds to scrubbing. This rewrites the LUT contents in order to overcome a bit-flip that may have occurred in the logic configuration. If scrubbing does not fix the fault (it is a stuck-at fault), the system moves on to loading functionally identical, yet physically distinct configurations that may bypass the stuck-at fault. If the fault still remains the system finally invokes evolutionary repair to come up with a new configuration that does not use this faulty LUT. If the system is in TMR mode, the other two FEs will mask the fault while the faulty FE is being repaired.

The evolutionary process makes use of a genetic algorithm, and an application-independent fitness function. Since this self-healing process uses intrinsic evaluation to rate the fitness of the evolved individual, it must load the configuration onto the physical FPGA to grade its fitness.

Overall, RARS, due to its adaptive nature, is able to save

one-third of the power used by a static TMR approach while still providing adequate fault coverage, and keeping the system online while being repaired.

B. Insights

RARS is an interesting approach that allows for the level of redundancy to be altered during runtime to fit the fault handling needs of the system and to save power. For example, the system can start off running in CED mode until a fault is detected. The system will now change its configuration to TMR in order to isolate which FE is faulty. When the faulty module is detected, additional steps will be taken to recover from the fault, leaving the other two modules intact to allow the system to run uninterrupted. If evolution takes place, the new individual can be compared with the other two non-faulty FEs to get its fitness evaluated. This is similar to the previous CRR approach but only requires the fitness evaluation of the under repair individual to be altered accordingly. This comparison does not lower the throughput of the system because there is already two working non-faulty FEs providing uninterrupted behavior. This evolution and fitness evaluation can continue until an individual matches the other two FEs outputs and is considered refurbished. The system can now enter back into duplex mode once the FE has been repaired in order to save power.

While the system is in duplex mode, one FE is disabled and is sitting there as a cold spare. No action is taken to ensure that it does not develop a dormant fault. In the worst case scenario, if this cold spare develops a stuck-at fault that is irrecoverable by using alternate configurations and an active FE also develops a stuck-at fault that requires evolution, when the system switches to TMR mode it will have to evolve two different configurations while the system has no valid output because all three FE outputs are discrepant. Although this scenario is highly unlikely, it is something that was not covered and must not be overlooked.

While RARS uses less power than TMR while it is running in duplex mode, it has a larger area overhead than TMR. Three Functional Elements must be present in the system regardless of the mode it is functioning in. Also, the Autonomic Element and rest of the system takes up a significant portion of the FPGA area as can be seen in Figure 9. The three FE units are relatively small compared to the logic needed to provide the AE, clock signals, bus macros, BUFGs, and DCM. This is significantly more area than is needed to implement a voter in a simple TMR approach given the application and granularity that it is applied to. If the user application (FE) were larger, the RARS system might not be able to fit on the FPGA given its limited size.

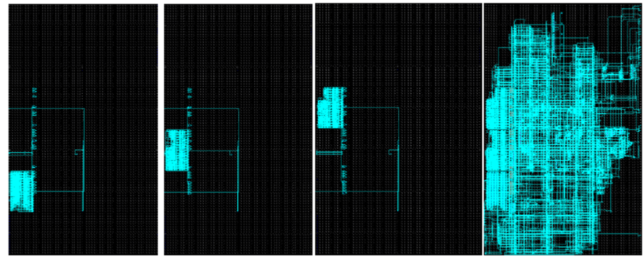

Figure 9: Area of FE1, FE2, FE3, and complete RARS system

VI. FPGA Architecture Supporting efficient TMR Fault Tolerance Support

A. Overview

Kyriakoulakos et al. proposes a slight modification to the existing Virtex-5 family of FPGAs in order for it to support fine grain redundancy with a reduced area overhead in [10]. This approach is not a high-level approach like the earlier ones discussed, but an architectural level one. As such, it requires a change in the physical fabric on the FPGA itself. The authors state that the basis of high-level approaches is redundancy and they aim to reduce the area overhead required to implement TMR. They propose a change to the LUT and CLB of a Virtex-5 to support fine grain TMR in order to better mitigate SEUs. The disadvantages of TMR they wish to address are that of area overhead of triplicating modules and adding a voter, and the latency that is added by the inclusion of the voter element.

Applying TMR at such a fine granularity allows the system to tolerate more faults than if it were applied at the module level but its area overhead is much greater because of all the voting elements required on the LUTs outputs. The authors state that using such fine granularity allows the vulnerable area to be much smaller which also reduces the upset rate. This has the effect of reducing the time between reconfigurations and therefore reduces power.

The Virtex-5 LUT structure is illustrated in Figure 10. The 6-input LUT actually consists of two 5-input LUTs that can be selected by a multiplexor (A6). The inputs of both LUTs must be the same and may output different results on O5 and O6. The authors noted this redundant nature in the LUT structure and will exploit it be reserving the other 5-LUT to be redundant and forcing the design to only use 5-input LUTs. Fine grain redundancy can therefore be applied with a significantly lower area cost.

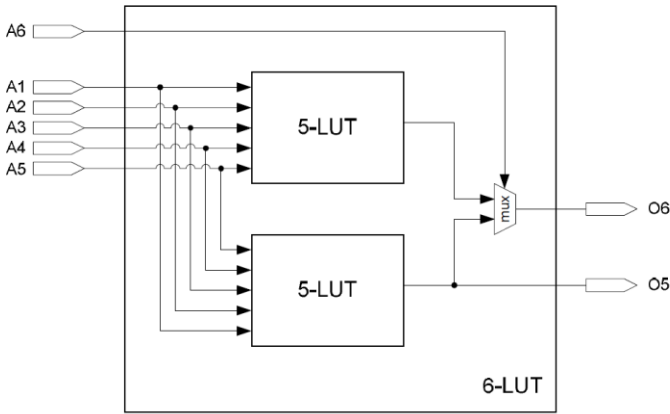

Figure 10: Virtex-5 6-LUT structure

The authors suggest two architectures: DMR and TMR. Both architectures restrict the mapping of functions to 5-input LUTs. If the application needs to use a 6-LUT as opposed to a 5-LUT, it will have to use additional LUTs to realize the function. The DMR architecture requires very little change and only calls for the addition of an XOR gate between the outputs of the 5-LUTs and the multiplexor as seen in Figure 11. The same LUT configuration will be loaded into both LUTs. The XOR gate will activate when the outputs are discrepant and notify that a fault has occurred in the column.

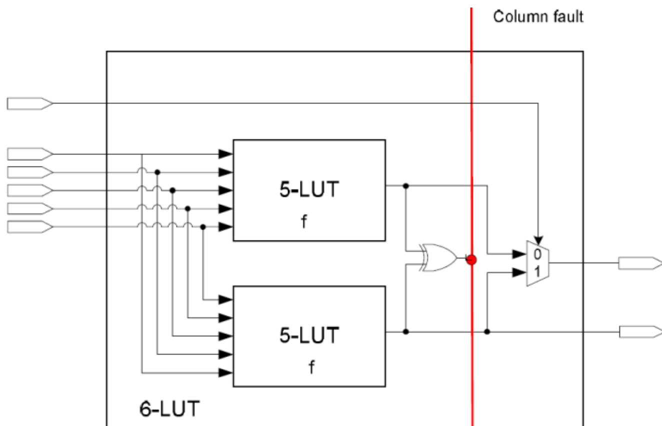

Figure 11: Dual Modular Redundancy LUT architecture

The TMR architecture proposed requires more additional logic to be fully realized. It uses the same principle as the DMR approach; however for the third redundant module it will split a 6-LUT between two different TMR configurations as outlined in Figure 12. The voting logic will be included in two of the three LUT structures and will pick the correct output to propagate. The multiplexor that was already in the LUT is being used for selecting the correct LUT output. The LUT that is shared between two functions must be able to accept two sets of signals instead of just one like the traditional 6-LUT. TMR mode can be enabled or disabled by setting the FT signal. Like with the DMR architecture, this TMR architecture also supports notifying whether or not a fault has occurred in the column. This group of three LUTs can either function as three 6-input functions like in the original architecture, or two 5-input functions with TMR.

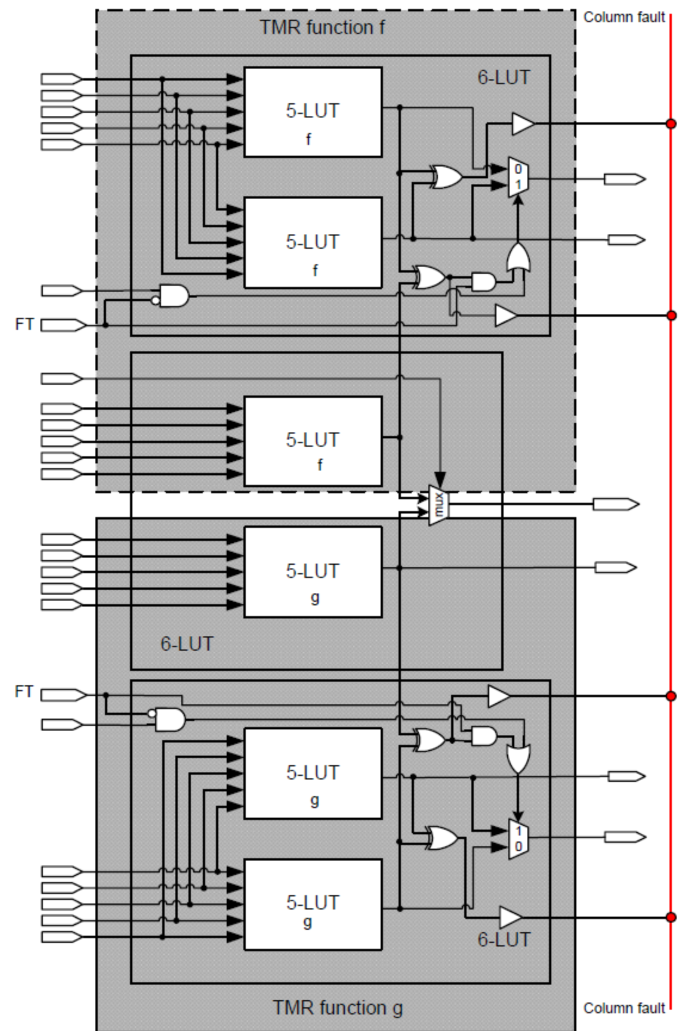

Figure 12: Triple Modular Redundancy LUT architecture

To compute the area overhead of such a design, the authors synthesized the ITC'99 circuits on a Virtex-5 FPGA with both no TMR and with TMR applied using 6-LUTs. They observed that the smallest circuits (b01, b02, and b06) had the largest overhead when TMR was applied as seen in Table 1.

	no TMR	TMR	TMR overhead
b01	5	30	6x
b02	4	24	6x
b03	38	120	3,16x
b05	174	585	3,36x
b06	8	51	6,38x
b08	22	84	3,82x
b10	35	168	4,8x
b11	105	315	3x
b12	263	885	3,37x
b15	1963	6684	3,40x
Total	**2617**	**8946**	**3,42x**

Table 1: LUT overhead for ITC'99 circuits

Since there is no family of FPGAs with 5-LUTs the authors synthesized these benchmarks for a Virtex-4 which uses 4-LUTs. They then used Xilinx tools to implement the same netlist onto a Virtex-5 using an option that can limit the number of inputs to each LUT for a 4-input, 5-input, and 6-input LUT. The outcome can be seen in Table 2. Moving to higher input LUTs decreases the amount of LUTs needed.

	4-input	5-input	6-input	Cost of using 5-input LUTs
b01	12	5	5	0%
b02	4	4	4	0%
b03	81	66	51	29%
b05	250	224	189	19%
b06	9	8	8	0%
b08	37	28	25	12%
b10	52	48	35	37%
b11	151	133	108	23%
b12	412	360	308	17%
b15	3005	2682	2295	17%
Total	**4013**	**3558**	**3028**	**17.5%**

Table 2: 4-input, 5-input, and 6-input LUT count

Using this new architecture, DMR can be added to a circuit with only 17.5% area overhead, and TMR can be added with only 76.25% area overhead (1.5 x 1.175 = 1.7625) with respect to using 5-LUTs over 6-LUTs. This overhead is greatly reduced compared to the 100% and 242% overhead needed to use DMR and TMR on a circuit, respectively. This new architecture however only applies to handling faults in the LUTs and not with the interconnection between them.

B. Insights

This architectural approach exploits the LUT structure of the Virtex-5 family of FPGAs to make it more efficient to use redundancy. The DMR architectural change is an extremely simple one that Xilinx could easily implement in their products. Adding the XOR gate into the 6-LUT eliminates the need of using another block just to compare the outputs of two LUTs. This has the effect of decreasing the latency and area required to implement DMR. Also, this architecture not only allows for the detection of any discrepancy, it also is able to isolate the location of the fault to a column. This can greatly benefit many online fault detection and handling methods that use redundancy by notifying where the fault has occurred on the column granularity.

Partial reconfiguration can take place on the faulty column instead of a larger area such as a module. Allowing the isolation of a fault to a column will also speed up the partial reconfiguration because the bitstream only has to be as large as that column. This results in saved power as the reconfiguration time will now be much less.

Scrubbing [9] can be more efficient as it only will need to scrub a column that has a LUT that shows discrepant behavior. CRR [4] can also benefit from this architecture as a discrepancy checker will not be necessary but can still be used as an extra line of defense. STARs [5] can greatly benefit from this by only running a STAR on the column that a fault was detected in. This will greatly improve the detection latency of this method. It can also employ the DMR architecture in the BISTER to simplify it. RARS [6] can also exploit this architectural change and further reduce power by not needing to have a discrepancy sensor while it is running in duplex mode.

The TMR architecture is more complicated to implement as it requires a lot more logic and control. The shared LUT will have to be changed to allow two sets of inputs and the other two will have to be altered to include the voting circuitry. One way to implement the shared 6-LUT is by allowing each 5-LUT to have their own inputs. This has the effect of increasing the amount of wires on the fabric and interconnect, but the complexity of implementation is simple. The other method is by using multiplexors to select which 5-inputs to use as seen in Figure 13. This leaves the interconnect unchanged but complicates the slice (the set of 3 6-LUTs) by increasing its area and adding latency to that LUT. If these obstacles can be easily overcome, then the TMR architecture may be viable as it will reduce the area cost of implementing TMR while being able to mask faults.

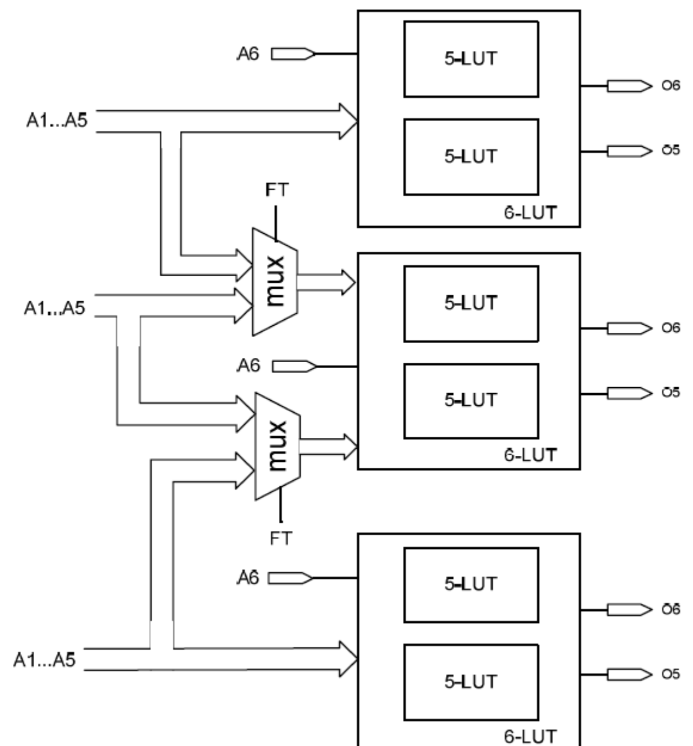

Figure 13: TMR “slice” using multiplexors

Overall, the architectural change to allow DMR, while minimal, will facilitate many different fault handling methods and should be straightforward to port existing 4-LUT based approaches to use this method. Furthermore, if the TMR architecture is realized, Xilinx’s TMRTool [12] could use it facilitate the implementation for the user.

VII. Master-Slave TMR Inspired Technique for Fault Tolerance

A. Overview

Lahrach et al. propose a novel technique that can tolerate multiple faults in [11]. Both single and double faults are covered with low overhead. This architecture consists of two types of CLBs: CLB-M and CLB-S. A Master-Slave Unit (MSU) consists of these two types of CLBs. When a fault occurs on a CLB-S, partial reconfiguration takes place to reconfigure one of the CLBs in the CLB-M to contain this function.

This technique does not detect or diagnose faults and it requires that some other method take these steps before the Master-Slave Technique (MST) takes place. A MSU is composed of a CLB-M surrounded by two, three, or four CLB-Ss. A CLB-M contains three CLBs, a voting element, and two switch blocks. This is essentially a block that is initially set up to work in a TMR configuration but will later sacrifice its redundancy to deal with faults. A CLB-S is a single CLB that when is deemed faulty, can be partially reconfigured on the CLB-M. In their approach, the routing between MSUs is fixed, but the routing within the MSUs can be altered to handle faults. Figure 14 outlines the layout of the CLB-Ms and CLB-Ss on an FPGA. Each CLB-M is surrounded by either three or four CLB-Ss. Figure 15 shows the organization of an MSU. When a fault is located in a CLB-S, a new configuration is loaded that does not use that CLB-S. Instead, it will use one of the redundant CLBs in the CLB-M.

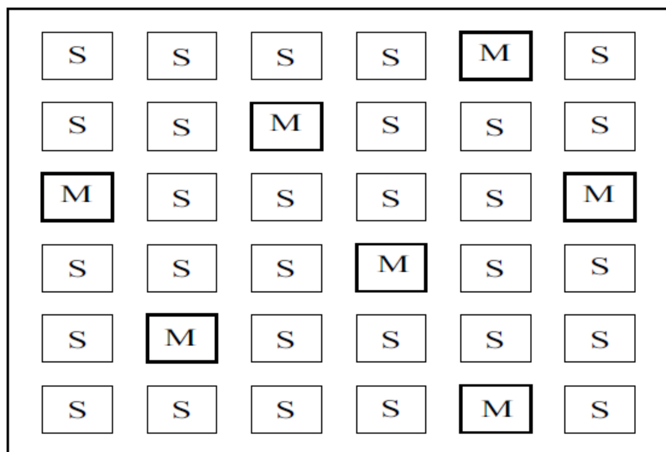

Figure 14: CLB-M (M) and CLB-S (S) organization

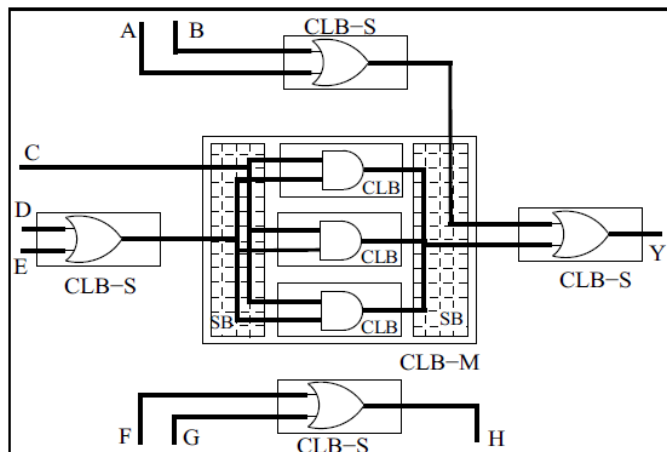

Figure 15: Master-Slave Unit

The authors state that there are three main benefits of using this approach: low overhead, runtime management, and complete availability. For a 104 x 80 FPGA they claim that the system reaches maximum reliability much sooner than other redundant methods. These results can be seen in Figure 16.

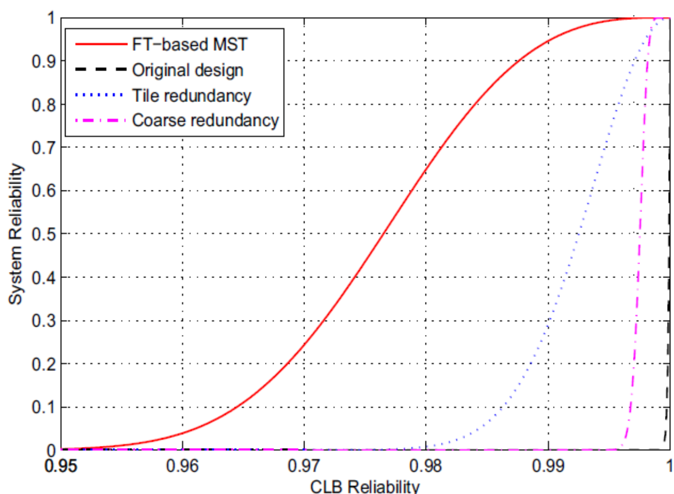

Figure 16: Reliability of different methods for a 104 x 80 FPGA

B. Insights

This approach, while preliminary and lacking fault detection and isolation, is a novel one that can be extended upon. One can use this technique to harden critical points in the circuit that may require TMR while others do not. In a sense, this is a form of partial TMR [13] that can adapt and sacrifice redundancy for faults detected in other blocks. Like other methods, it also relies heavily on partial reconfiguration to only program the portion of the MSU that is needed. It also requires the ability to program the switch blocks (SBs) effectively so that when a CLB-S is moved onto a CLB-M, the right routing is implemented.

This method can also benefit from the proposed architecture overviewed in Section VI. The CLB-M can use the TMR approach while the CLB-S can use the DMR approach. When

a CLB-S is detected as faulty and gets partially reconfigured to the CLB-M then it no longer can use the TMR approach and it will leave one CLB unprotected by any form of fault detection.

VIII. Conclusion

Several architectural and high-level approaches were discussed that leverage nMR for fault tolerance, handling, and isolation. Vigander used a triplex voting scheme with imperfect individuals. DeMara et al. extended CED to have two individuals compete with one another in his Competative Runtime Reconfiguration approach. Abramovici et al. used CED within the BISTERs in his STARs approach to detect which pair of blocks were faulty. Al-Haddad et al. allowed for a dynamic change in the level of redundancy in his Reconfigurable Adaptive Redundancy System (RARS). Kyriakoulakos et al. proposed a new architecture for both fine grain DMR and TMR that significantly reduces the area of both methods. Lahrach et al. introduced a Master-Slave TMR technique that allows for a master CLB to sacrifice redundancy in order to maintain system reliability. Table 3 summarizes the overhead all these approaches. Table 4 summarizes the sustainability of all these approaches. These tables are based on the metrics outlined in [2].

Approach	Area Overhead	Power Overhead	Throughput Reduction	Detection Latency
Basic TMR	200% of application	200% of application	Indeterminate	Negligible
Vigander	200% of application	Not Addressed	Not Addressed	Negligible
CRR	100% of application	100% of application	Variable (0-15%)	Negligible
STARs	4-11% of FPGA	Not Addressed	0-16%	0-17 seconds
RARS	>200% of application	Variable (0-200% of application)	Indeterminate	Negligible
DMR Architecture	17.5% of application	17.5% of application	Negligible	Negligible
TMR Architecture	76.5% of application	76.5% of application	Negligible	Negligible
Master-Slave	25% of FPGA	Not Addressed	Indeterminate	Not Addressed

Table 3: Overhead of discussed approaches

Approach	Fault Exploitation	Recovery Granularity	Fault Capacity
Basic TMR	No	Variable	Three configurations
Vigander	Yes	FPGA	Three configurations
CRR	Yes	Variable	Two configurations
STARs	Yes	LUT	One H-STAR & One V-STAR
RARS	Yes	Variable	Variable
DMR Architecture	No	LUT	One LUT
TMR Architecture	No	LUT	Three LUTs
Master-Slave	No	CLB	Five CLBs

Table 4: Sustainability of discussed approaches

References

[1] F. Lima, L. Carro, and R. Reis, "Designing fault tolerant systems into SRAM-based FPGAs," *DAC '03: Proceedings of the 40th annual Design Automation Conference* 2003, pp. 650-655.

[2] M. Parris, C. Sharma, and R. Demara, "Progress in Autonomous Fault Recovery of Field Programmable Gate Arrays," accepted to ACM Computing Surveys December, 2009.

[3] S. Vigander, "Evolutionary Fault Repair of Electronics in Space Applications," University of Sussex, February 2001.

[4] R. F. DeMara, K. Zhang, "Autonomous FPGA Fault Handling through Competitive Runtime Reconfiguration," *Proceeding of NASA/DoD Conference on Evolvable Hardware (EH'05)*, Washington D.C., U.S.A.. June 29 – July 1, 2005.

[5] M. Abramovici, J. M. Emmert, C. E. Stroud, "Roving Stars: An Integrated Approach To On-Line Testing, Diagnosis, And Fault Tolerance For Fpgas In Adaptive Computing Systems," Evolvable Hardware, NASA/DoD Conference on, pp. 0073, The Third NASA/DoD Workshop on Evolvable Hardware, 2001.

[6] R. Al-Haddad, R. DeMara, "A Sustainable Organic Architecture Emphasizing Partial Reconfiguration for Reduced Power Consumption," University of Central Florida, 2010.

[7] C. Müller-Schloer, "Organic computing: on the feasibility of controlled emergence," *Proc. 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ACM, 2004, pp. 2-5.

[8] G. Lipsa, A. Herkersdorf, W. Rosenstiel, O. Bringmann, and W. Stechele, "Towards a framework and a design methodology for autonomic SoC,", pp. 391-392.

[9] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," August 2009.

[10] K. Kyriakoulakos, D. Pnevmatikatos, “A Novel SRAM-Based FPGA Architecture for Efficient TMR Fault Tolerance Support,” *International Conference on Field Programmable Logic and Applications*, 2009.

[11] F. Lahrach, A. Doumar, E. Châtelet, A. Abdaoui, “Master-Slave TMR Inspired Technique for Fault Tolerance of SRAM-based FPGA,” *IEEE Annual Symposium on VLSI*, 2010.

[12] Xilinx, “Xilinx TMRTool,” http://www.xilinx.com/ise/optional_prod/tmrtool.htm

[13] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin, “Improving FPGA Design Robustness with Partial TMR,” *44th Annual International Reliability Physics Symposium*, 2006.