Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy

Jonathan Johnson and Michael Wirthlin
NSF Center for High-Performance Reconfigurable Computing (CHREC) *
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT 84606, USA
jonjohn@byu.net, wirthlin@ee.byu.edu

ABSTRACT

Triple Modular Redundancy (TMR) is a common reliability technique for mitigating single event upsets (SEUs) in FPGA designs operating in radiation environments. For FPGA systems that employ configuration scrubbing, majority voters are needed in all feedback paths to ensure proper synchronization between the TMR replicates. Synchronization voters, however, consume additional resources and impact system timing. This paper will introduce and contrast four algorithms for inserting synchronization voters while automatically performing TMR. The area cost and timing impact of each algorithm on a number of circuit benchmarks will be reported. This paper will demonstrate that one of the algorithms provides the best overall timing performance results with an average 9.8% increase in critical path length over a triplicated design without voters. Another algorithm provides far better area results at a slightly higher timing cost (an average 2.1% area increase over a triplicated design without voters).

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Reliability, Design, Algorithms

1. INTRODUCTION

SRAM-based FPGAs are an attractive alternative to ASICs for space-based computing missions because of their in-orbit reconfigurability, their ability to perform application-specific computations, and their lower non-recurring engineering costs [1, 2]. SRAM-based FPGAs, however, are susceptible to radiation effects in space environments, particularly radiation-induced single-event upsets (SEUs). SRAM-based FPGAs contain a large number of internal memory cells that can be upset by high energy particles. These include memory cells for configuration memory, user flip-flops, internal block memory, and other device-specific customization modes. SEUs within the configuration memory are especially challenging as they may change the functionality implemented by the FPGA. For example, configuration SEUs may modify the routing, logic, clocking, or other aspects of a user design.

Unique strategies for configuration SEU mitigation have been developed for systems that incorporate SRAM-based FPGAs within high radiation environments. The most common and well understood technique combines triple modular redundancy (TMR) [3] and configuration memory scrubbing [4]. TMR uses hardware redundancy to mask any single design failure by voting on the result of three identical copies of the circuit. Configuration scrubbing involves the continuous configuration of the device with a known golden bitstream stored in a protected memory to prevent the buildup of multiple coincident SEUs. Fault-injection and radiation experiments have demonstrated the robustness of this mitigation approach.

Inserting majority voters is an important step in automated TMR. Majority voters are used to resynchronize the circuit state after configuration scrubbing [5], as well as for other purposes. Voters are inserted within *all* feedback paths to ensure that state within logic feedback is updated when the bitstream scrubbing process repairs circuit resources. Identifying good locations for these voters, however, is a difficult problem. Poor synchronization voter locations lead to large area overhead and a significant increase in critical path timing. This paper investigates and identifies algorithms for inserting synchronization voters in an FPGA design during automated TMR that minimize area overhead and loss in critical path timing performance. While automated approaches for inserting synchronization voters during TMR have been developed and incorporated into automated tools [6], we are unaware of any previously published work that describes how this voter insertion is performed.

This paper will begin by describing the SEU mitigation strategy involving TMR and configuration scrubbing. Automated approaches for implementing TMR will be described with an emphasis on voter insertion. The need for synchronization voters will be demonstrated with an example. The paper will then present four algorithms that determine appropriate synchronization voter locations. These algorithms

*This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

will be compared in terms of their area overhead and timing performance impact. This paper will demonstrate that restricting voters to locations directly after flip-flops is a good timing performance heuristic for voter insertion algorithms.

2. MITIGATION TECHNIQUES FOR SRAM FPGAS

SRAM-based FPGAs have dense arrays of memory cells and consequently are especially sensitive to SEUs. Much like SRAM and DRAM, SRAM-based FPGAs contain a large amount of internal state that is sensitive to single event effects. FPGAs contain user flip-flops, block memories, and the device configuration memory. The largest portion of this state is found within the configuration memory that defines the operation of the circuit. Upsets within the configuration memory modify the behavior of the routing, logic, clock tree, etc. These upsets appear as "hardware faults" or failures within the user circuit[1].

Unique mitigation methods are needed to protect FPGA circuits from radiation-induced SEUs. The most common mitigation approach for FPGAs is a combination of TMR and configuration bitstream scrubbing. TMR is used to temporarily mask configuration SEUs and configuration scrubbing is used to repair configuration SEUs. Together, these techniques provide significant improvements in reliability.

2.1 Triple Modular Redundancy

Triple modular redundancy is a well known fault mitigation technique that uses redundant hardware to mask circuit faults. A circuit protected by TMR has three redundant copies of the original circuit and a majority voter (see Figure 1). A single fault in any of the redundant hardware modules will not produce an error at the output as the majority voter will select the correct result from the two working modules. Triplicated voters are often used to avoid a single point of failure.

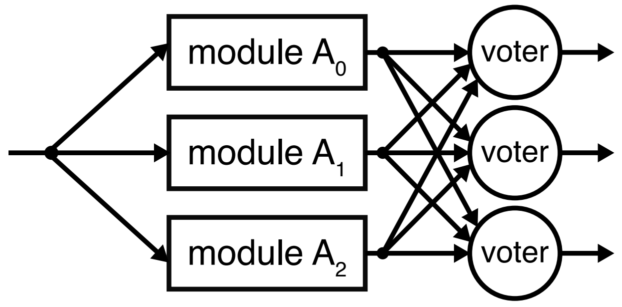


Figure 1: Structural Implementation of TMR.

TMR is used extensively in SRAM-based FPGA systems to mitigate against radiation-induced SEUs. Several experiments have demonstrated significant improvements in reliability when using TMR through fault injection and radiation testing. The use of TMR in FPGAs, however, is very expensive as the resulting circuit is at least three times the size of the original circuit, and the circuit operates at a slower clock rate than the original circuit.

[1]These faults, however, are soft faults that can be repaired through configuration scrubbing.

2.2 Configuration Scrubbing

While TMR is effective at protecting a circuit from *single* circuit failures due to SEUs, it cannot protect the circuit from multiple independent SEUs. If multiple SEUs occur within the configuration memory, two or three copies of the redundant circuit may fail. With more than one failure, the majority voters will chose an incorrect value (i.e., two incorrect circuits) and lose the benefit of redundant hardware. Configuration bitstream scrubbing is a technique used to *repair* configuration SEUs and prevent the build up of multiple independent SEUs.

Configuration scrubbing is used in conjunction with TMR to prevent the accumulation of multiple coincident SEUs [7, 8]. Like conventional memory scrubbing, configuration scrubbing involves the continuous reading and repairing of the configuration data to prevent the accumulation of SEUs. Most FPGA scrubbing techniques require some external hardware including external memory for configuration data storage. Like memory scrubbing, there are a variety of ways to implement configuration scrubbing in FPGAs [9, 10]. Additionally, the time required to perform an individual scrubbing cycle on an entire device is dependent upon the size of the device and the implementation of the scrubber.

3. AUTOMATED TMR

Although TMR is often applied to designs manually, the process is straightforward enough to be implemented by an automated CAD tool. Existing tools for applying TMR to FPGA designs include the Xilinx XTMR tool [3, 6] and the BYU/Los Alamos National Laboratory BL-TMR tool [11]. Using an automated tool can provide several advantages over implementing TMR by hand. For example, inserting voters in the proper places manually can be a tedious and error prone process.

The voter insertion algorithms presented in this paper operate on circuits represented at the post-synthesis netlist level. In this representation, circuits consist of instantiations of FPGA primitives such as LUTs, flip-flops, and dedicated hardware, and nets that define the connectivity between the primitives. The result of the algorithms is a new netlist that contains a triplicated version of the original netlist with voters inserted at appropriate locations. After automated TMR, the triplicated netlist follows the traditional FPGA process of technology mapping, placement, and routing, as shown in Figure 2.

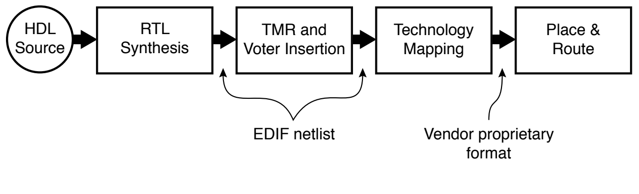


Figure 2: TMR toolflow for FPGAs.

The process of automated TMR begins with creating three identical copies of the original circuit. First, each component instantiation is triplicated. Next, each net is triplicated. The nets are then connected such that the connectivity of each of the three replicates matches the connectivity of the original circuit. This is the straightforward part of TMR.

Inserting majority voters to mask errors is a more complex process and is the focus of the algorithms presented in this paper.

3.1 Voter Insertion

The location of voters in a TMR design is specified in terms of nets from the original, unmitigated design. When inserting a voter at a net location, the net is split into two pieces and a voter is inserted in the middle. The source of the original net becomes the source of the voter and all of the sinks of the original net are driven by the voter. This voter insertion occurs in the context of TMR where there are three copies of the source and three copies of each instance. Inserting a voter on a net in the original design involves replacing the three copies of the net in the TMR design with voter nets as described in the following process:

1. Instantiate three voters to perform triple voting on the given net,
2. Identify the three copies of the source of the net and connect these sources to the inputs of each of the three voters, and
3. Connect the output of each voter to the corresponding sinks of the net.

We refer to the process of inserting voters on a net as cutting a net with voters, since the original net is replaced by two sets of triplicated nets: one feeding into the voters and one exiting from them. Figure 3 illustrates the basic triplication and voter insertion process.

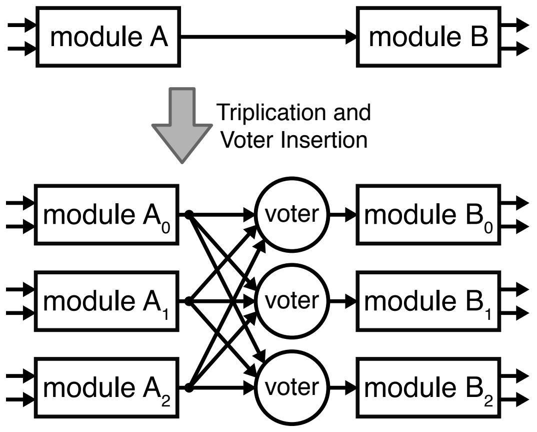


Figure 3: The net after Module A is cut with triplicated voters.

Automated voter insertion is difficult because there are constraints that govern where voters can be inserted. Finding an optimal voter configuration that meets the constraints is difficult because the voter locations affect the timing performance, area, and reliability of the resulting circuit.

One of the constraints that governs voter insertion is that there are certain nets in a netlist representation of a circuit that cannot be cut by voters because of the FPGA architecture. Figure 4 illustrates an example of this issue. The figure shows two bits of a simple ripple-carry adder implemented using the dedicated carry chain and arithmetic hardware found in the Virtex FPGA family. The adder in the figure is implemented using logic cells in two different slices. Net A in the figure cannot be cut by voters because this net is implemented by a dedicated route connection within a logic slice. Since there is no reconfigurable routing between a *MULT_AND* primitive and a *MUXCY* primitive, a *MULT_AND* cannot drive a voter and a *MUXCY* cannot receive its input directly from a voter. We refer to locations such as net A as illegal cut locations. Other illegal cut locations include nets between *MUXCY* and *XORCY* primitives, nets between internal multiplexors that are used to create wider LUTs or multiplexors (i.e. *MUXF5*, *MUXF6*, *MUXF7*, *MUXF8*), and some nets connecting cascaded *DSP48* primitives. Voter insertion algorithms must not create netlists that have voters inserted at illegal cut locations.

In addition to illegal cut locations, there are other locations where inserting voters is legal, but results in an undesirable implementation. For example, net B in Figure 4 is implemented using fast dedicated carry chain routing. Adding a voter on this net is legal but breaks the high-speed carry chain logic. To add a voter, the output of the *MUXCY* primitive in the lower slice must be routed to a different slice where the voting is performed. The output of this voter would then need to be routed into the CIN input of the upper *MUXCY*, breaking the high-speed carry chain. In addition to avoiding illegal cut locations, the voter insertion algorithms presented in this paper avoid dedicated carry chain routing nets in order to preserve timing performance as much as possible.

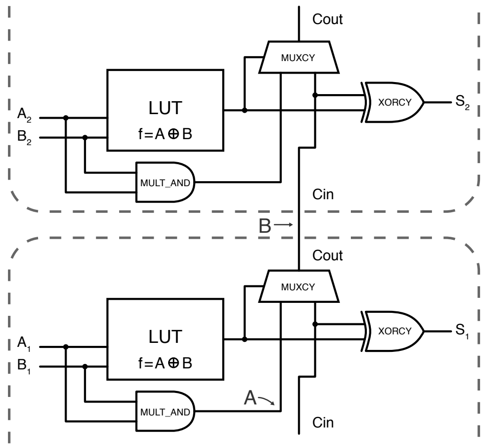


Figure 4: Two bits of a ripple-carry adder using FPGA primitives, carry chain, and dedicated arithmetic hardware.

Various reliability concerns motivate voter insertion at different circuit locations. We refer to voters by names that indicate their purpose in a circuit. Voter categories typically used in FPGA implementations of TMR for reliable operation in space-based missions include the following:

Reducing Voters are used to reduce a triplicated signal down to a single output. These are used when non-redundant FPGA outputs are needed to interface with the other components of a system (such as at the outputs of the FPGA).

Clock Domain Crossing Voters are used for resynchronizing signals crossing clock domains in a triplicated circuit.

Partitioning Voters are used to increase reliability in the presence of multiple independent upsets by creating additional TMR partitions.

Synchronization Voters are used to keep sequential logic state synchronized between TMR replicates when a scrubbing process corrects SEUs. This category is the focus of the algorithms presented in this paper.

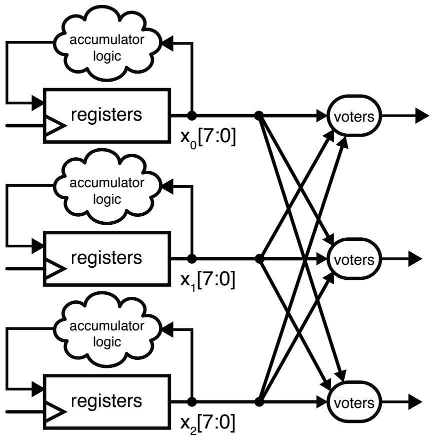
3.2 Synchronization Voters

Synchronization voters are necessary in TMR circuits used on FPGAs that employ configuration scrubbing. The purpose of these voters is to resynchronize the registered state within the design after FPGA logic problems are repaired with configuration scrubbing. Synchronization voters are placed within sequential feedback loops to restore system state. To demonstrate the importance of synchronization voters, consider the simple triplicated counter in Figure 5(a). Three copies of a register and accumulator logic are instantiated to provide fault tolerance for any single circuit failure. Voters are placed at the outputs to select the majority result should a failure occur. The synchronization problem that occurs with this circuit is demonstrated by the waveform of Figure 5(b).

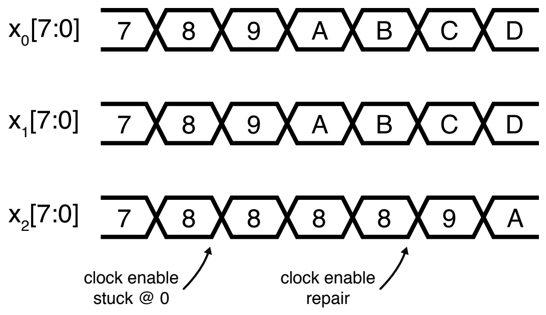
In this example, a configuration fault occurs that forces the clock enable of the third TMR replicate into a stuck-at-0 condition. Because of this fault, the counter does not increment; it remains in the same count state until the clock enable is repaired by scrubbing. Once the counter has been repaired through configuration scrubbing, it continues its count sequence from the state in which it was stuck. Although repaired and operating properly, the counter is out of sequence with the other two counters. While the TMR voter circuitry properly ignores the incorrect count value, any additional faults in the other TMR replicates would cause the redundancy to be overcome, allowing the error to propagate throughout the rest of the circuit.

Synchronization voters are voters placed within the *feedback* of a circuit to provide resynchronization after a fault occurs. Figure 6(a) demonstrates the proper use of synchronization voters by placing the voters *within* the feedback loop. Using the voters within the feedback ensures that the proper input value is provided to all of the counters no matter where the fault lies. The benefits of this technique are illustrated in the counter failure waveform of Figure 6(b).

As described in the earlier example, the third TMR replicate experiences a stuck-at-0 fault on its clock enable input. While this fault is present, the third counter retains the same value and falls out of sequence with the other counters. The voter circuitry masks this faulty value and provides a correct value on the feedback path. Once the configuration fault is repaired by online scrubbing, the proper value is loaded into the third counter and it becomes resynchronized with the other counters. With all three counters synchronized and repaired, the circuit will reliably operate in the presence of another configuration fault.



(a) Simple counter with voters *outside* the feedback loop.



(b) A simple counter is susceptible to TMR synchronization problems when SEUs occur within the feedback loop, even after scrubbing has corrected the configuration memory.

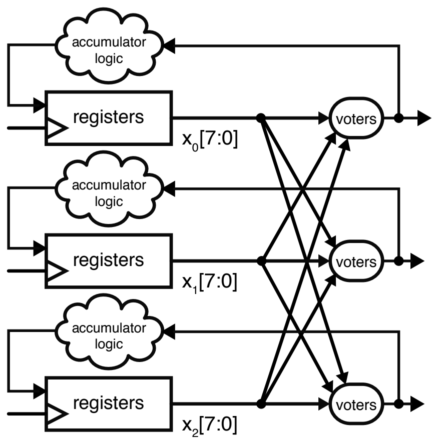
Figure 5: A simple triplicated counter.

4. ALGORITHMS FOR AUTOMATED VOTER INSERTION

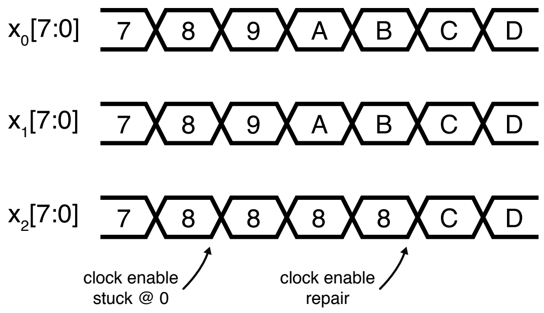
Although synchronization voters are essential in FPGA circuits that use TMR, manually adding synchronization voters in a design is a difficult and error prone process. Automated tools are necessary for selecting synchronization voter locations and inserting them in the design. This section will introduce four algorithms for automatically selecting locations for synchronization voters.

Synchronization voter insertion algorithms must determine a set of nets within a design that cuts *all* feedback in the design. Voters are placed on each of these nets to ensure synchronization voting occurs within the feedback structures of a design. Determining a set of voter locations that satisfies this constraint is an instance of the feedback edge set (FES) problem. Determining a *minimum* set of voter insertion locations to satisfy the constraint is an instance of the minimum (FES) problem, which is NP-hard [12].

While polynomial time approximation algorithms for the minimum FES problem exist ([13], [14]), the minimum set of voter insertion locations is not necessarily the best solution for FPGA implementations of TMR. In order to preserve performance, care must be taken to avoid voter insertion locations that would negatively impact timing performance. In addition, existing FES algorithms cannot be applied directly because FPGAs have illegal cut locations. Each of the algorithms in this section solves the FES problem for voter insertion in a way that avoids illegal cut locations. In



(a) Simple counter with voters *inside* the feedback loop.



(b) Synchronization voters protect the counter from TMR synchronization problems when scrubbing SEUs.

Figure 6: A triplicated counter protected by synchronization voters.

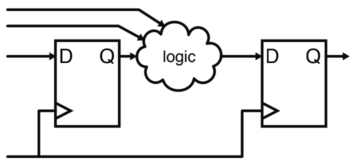
addition, the algorithms employ heuristics based on FPGA architecture that attempt to minimize circuit area and timing impact.

This section will first present a very simple voter insertion algorithm that solves the problem in a local manner. This will be followed by three algorithms based on strongly connected component (SCC) decomposition that attempt to meet the constraints while using fewer voters and applying timing-based heuristics. An upper bound on the run-time complexity of each algorithm will be given in terms of $|V|$ (the number of nodes in the circuit graph) and $|E|$ (the number of edges in the circuit graph).

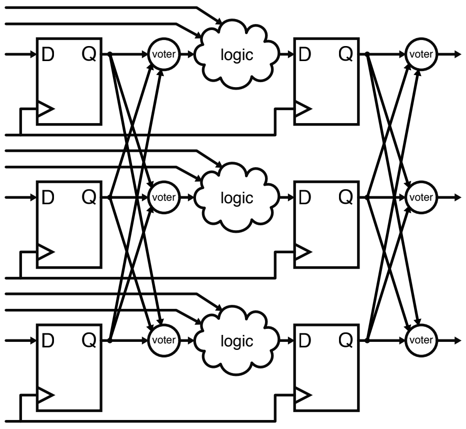
4.1 Voters After Every Flip-Flop

The *Voters After Every Flip-Flop* algorithm places a voter after the output of each flip-flop in a circuit. The two flip-flops in the circuit of Figure 7(a) would be triplicated with voters after each flip-flop as shown in Figure 7(b). While this algorithm does not perform any feedback analysis, it is guaranteed to intersect every cycle with a voter because in standard synchronous circuits, each cycle must have at least one flip-flop.

The algorithm requires only a simple analysis of the circuit and runs in $O(|V|)$ time. Although this algorithm is simple, it ensures that only one set of triplicated voters can be placed in a single timing path, which helps reduce the negative timing impact of voter insertion. A simple timing path is any path from one flip-flop to another; when voters are placed only directly after each flip-flop, it is impossible to have more than one voter in a single timing path.



(a) Original circuit before TMR.



(b) The triplicated circuit has voters after each flip-flop.

Figure 7: *Voters After Every Flip-Flop* insertion algorithm.

4.2 Algorithms Based on SCC Decomposition

While the preceding simple algorithm satisfies the constraints of synchronization voter insertion, it often inserts many more voters than are actually needed. The algorithms that follow are designed to insert fewer voters. They work progressively by identifying feedback, inserting voters in the feedback, and stopping when there is no feedback left uncut. By inserting fewer voters, these algorithms have the potential to be able to produce circuits with better timing performance and area.

The following three algorithms use strongly connected components (SCCs) to determine a more efficient feedback cut set. The SCCs of a graph are the maximal subgraphs in which there is a path from each node to every other node. SCC decomposition is the process of finding all of the SCCs in a graph. The definition of an SCC leads to the following corollaries:

- Each SCC contains at least one cycle,
- No cycle spans more than one SCC,
- There are no cycles outside of the SCCs of a graph, and
- Nodes not involved in any cycles will not be found in any SCC.

These corollaries suggest that decomposing a graph into SCCs can be a way of simplifying the problem of determining where to place synchronization voters. Since any cycle involves nodes only in a single SCC, each SCC can be treated as a subproblem of the overall synchronization voter insertion problem. Furthermore, graph edges not involved in any of a graph's SCCs need not be considered for voter insertion.

In order to use SCC decomposition to determine where to insert synchronization voters, the algorithms in this section first generate a directed graph representation of a circuit. Each component instantiation in the circuit netlist becomes a node in the graph. Each net in the netlist becomes a set of edges; an edge is created from every net source to every net sink.

Once a graph representation of the circuit has been created, the algorithms break up the SCCs of the graph into smaller and smaller SCCs by systematically removing edges until all SCCs are dissolved and there are no cycles left in the graph. The process of breaking up SCCs by removing edges is illustrated with the example graph in Figure 8. This graph contains two SCCs: $\{\{2,3,4,5,6,7,8\}, \{9,10,11\}\}$. The removal of edge $(6,3)$ would break the first SCC into two smaller SCCs, resulting in the SCC decomposition: $\{\{2,3,4,5\},\{6,7,8\},\{9,10,11\}\}$. Removing edge $(10,11)$ would dissolve the third SCC into a feed forward component, giving the SCC decomposition: $\{\{2,3,4,5\}, \{6,7,8\}\}$. Additionaly removing edges $(2,3)$ and $(7,8)$ would completely dissolve all of the SCCs in the graph. If this graph represented a circuit, then placing synchronizing voters at each of these four locations would cut all feedback with voters, ensuring proper TMR synchronization.

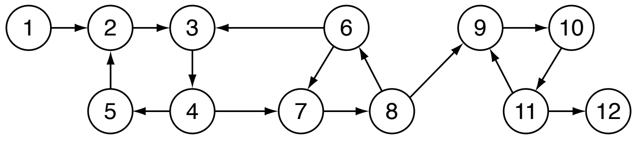


Figure 8: SCCs can be broken by removing edges.

The algorithms that follow require repeated use of SCC decomposition (after each edge removal) in order to determine what subproblems are created when edges are removed. Several algorithms for SCC decomposition exist, including Kosaraju's algorithm [15] and Tarjan's algorithm [16], both of which run in $O(|V| + |E|)$ time.

The algorithms based on SCC decomposition all have the same basic structure which is summarized with pseudocode in Algorithm 1. The basic structure of the algorithms uses a stack-based method for processing all of the SCCs. To begin, an SCC decomposition of the circuit graph is computed, and all of the SCCs are pushed onto a stack (S). The algorithm iterates over the SCCs in the stack until the stack is empty. During each iteration of the while loop, a single SCC is popped off of the stack for processing. Edges are removed from the SCC to break the SCC into smaller SCCs or single nodes. An SCC decomposition is recomputed and any remaining SCCs are pushed onto the SCC stack for processing in the next iteration. This process continues until all of the SCCs have been broken into feed forward components. The edge set used to break the feedback of the SCCs indicates the location of the synchronization voters.

The algorithms that use this structure differ in the manner in which they select edges to remove to dissolve the SCCs into feed forward components. Different edge selection strategies are used to identify feedback edge cutsets that result in, for example, a faster circuit or a fewer number of voters.

Algorithm 1 Basic Structure of SCC Decomposition Algorithms

```
Initialize List $L$
Initialize Stack $S$
Compute SCC decomposition
for all $scc$ in resulting SCCs do
  Push $scc$ onto $S$
end for
while $S$ is not empty do
  $scc = S$.pop()
  Algorithm specific edge removal
  Add removed edges to List $L$
  Recompute SCC decomposition of $scc$.nodes
  for all $newSCC$ in resulting SCCs do
    Push $newSCC$ onto $S$
  end for
end while
Insert voters on nets corresponding to edges in $L$
```

4.2.1 Highest Fanout SCC Decomposition Algorithm

The *Highest Fanout SCC Decomposition Algorithm* has the basic structure outlined above and uses a heuristic intended to minimize the number of voters needed to intersect the cycles of a circuit. The heuristic is based on the intuitive suggestion that a significant amount of feedback can be cut by inserting voters on a single net with high fan-out. Nets with high fanout are likely to be part of multiple cycles that can all be cut at a single point. At each iteration of the SCC processing while loop, the SCC in question is analyzed to find the node with the highest legal cut fanout. The legal cut output edges from this node are then removed from the graph. In this manner, edge removal is prioritized with high fanout nets. The algorithm runs in $O(|V|^2|E|)$ time, but this is a conservative upper bound. The $|V|^2$ term comes from the fact that each time an SCC is processed by the while loop, each of its nodes must be examined to find the node with the highest fan-out. In practice, the number of times the SCC processing loop executes is far fewer than $|V|$, and the number of nodes in any SCC is generally far fewer than $|V|$.

4.2.2 Highest Flip-Flop Fanout SCC Decomposition Algorithm

The *Highest Flip-Flop Fanout SCC Decomposition Algorithm* is similar to the previous algorithm but identifies high fanout nets that originate from flip-flops only. This algorithm has two priorities: inserting a small number of voters and reducing the negative impacts of voter insertion on timing performance. When more than one set of voters is inserted in a single timing path (i.e. a path from one register to the next), the voters negatively affect timing performance more than is necessary. For each SCC processed by this algorithm, the flip-flop with the highest legal cut fanout in the SCC is determined. The legal cut output edges from this node are removed. Since a timing path consists of the logic from one flip-flop to the next, inserting voters only directly after flip-flop outputs ensures that at most one voter will be inserted per timing path. The runtime of this algorithm is the same as the previous, $O(|V|^2|E|)$. As with the previous algorithm, this is a conservative upper bound. The timing performance benefits of using this algorithm will be demonstrated in Section 5.

For an example of the *Highest Flip-Flop Fanout SCC Decomposition Algorithm*, consider Figure 9. The figure is a graph representation of a circuit that includes flip-flops that are involved in feedback. The flip-flop nodes in the graph are indicated with gray shading. The initial SCC decomposition performed by the algorithm gives the SCCs $\{\{1, 2, 4, 3\}, \{5, 7, 6\}\}$. The algorithm pushes these SCCs onto a stack and begins processing them with the while loop. The first SCC popped off of the stack is $\{5, 7, 6\}$. Its only flip-flop node, node 7, is chosen to have its data output net removed from the graph. In this case, the output net from node 7 is represented by a single edge, $(7, 6)$. Edge $(7, 6)$ is removed from the graph and an SCC decomposition of the subgraph induced by the nodes $\{5, 7, 6\}$ is computed. Since the feedback has been removed, no SCCs are found in the subgraph and the while loop continues to the next iteration.

The next iteration pops the SCC $\{1, 2, 4, 3\}$ off of the stack. In this SCC, node 3 is the flip-flop node with the highest fan-out, so it is chosen to have its data output net removed from the graph. Its output net is represented by edges $(3, 1)$, $(3, 2)$, and $(3, 5)$. These edges are removed from the graph (note, however, that this results in only a single voter insertion location) and an SCC decomposition of the subgraph induced by nodes $\{1, 2, 4, 3\}$ is performed. The result of the decomposition is a single remaining SCC: $\{2, 4\}$. This SCC is pushed onto the stack and the while loop continues to the next iteration.

In the next iteration, the SCC $\{2, 4\}$ is popped off of the stack. Since node 4 is its only flip-flop node, its data output net edges are removed ($(4, 2)$ and $(4, 6)$). An SCC decomposition of the subgraph induced by $\{2, 4\}$ is performed and no SCCs are found. At this point the stack is empty and all of the SCCs have been broken up into feed forward only components. The edges removed by the algorithm result in voters being placed directly after each of nodes 3, 4, and 7. This is sufficient to correctly mitigate the circuit's feedback.

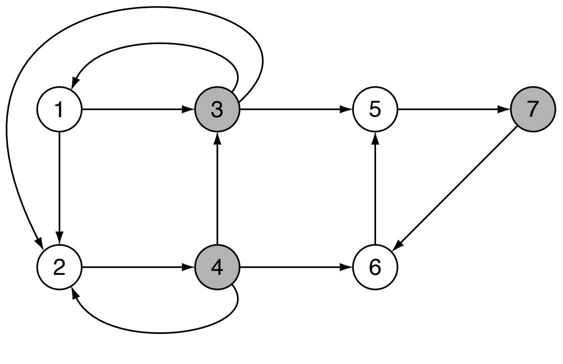

$$\begin{tikzcd}
1 \arrow[r] \arrow[d] \arrow[d, bend right] & 3 \arrow[r] \arrow[l, bend right] \arrow[dl, bend right] & 5 \arrow[r] & 7 \arrow[dl] \\
2 \arrow[r] & 4 \arrow[u] \arrow[r] \arrow[l, bend left] & 6 \arrow[u] &
\end{tikzcd}$$

Figure 9: Graph representation of a circuit that includes flip-flops involved in feedback.

4.2.3 *Highest Fan-in Flip-Flop Output SCC Decomposition Algorithm*

The *Highest Fan-in Flip-Flop Output SCC Decomposition Algorithm* uses a heuristic similar to the high fan-out heuristic. It is based on the hypothesis that just as inserting voters after flip-flops with high fan-out can reduce the total number of voters needed to cut all feedback, inserting voters after flip-flops with high fan-in can have a similar effect. In this algorithm, flip-flop fan-in is defined as the number of nets that directly or indirectly feed into the data input of a flip-flop going up to five levels backwards as computed by a depth-limited DFS traversal. For each SCC processed by the while loop, the algorithm finds the flip-flop in the SCC with the highest fan-in and a legal voter location output edge and removes the data output edge. The run time complexity is $O(|V|^3 + 2|V|^2|E| + |V||E|^2)$. As with the other algorithms, this is a conservative upper bound. The extra $|V|$ factor in the dominant term (over the $|V|^2$ of the previous two algorithms) comes from the fact that for each flip-flop node found in each SCC, a depth-limited DFS must be performed to determine the fan-in of the flip-flop. In practice, the number of nodes traversed in each of these searches is far fewer than $|V|$.

5. EXPERIMENTAL RESULTS

Experiments were performed in order to compare the algorithms in the preceding section in terms of their impact on the timing performance and area of a circuit when applying TMR. It is well known that applying TMR to an FPGA design generally causes poorer timing performance and increases the size of the circuit by at least $3X$. A poor voter insertion approach can induce a size increase of well over $3X$. The purpose of these experiments is to determine which voter insertion strategies are best for preserving the timing performance of a circuit and reducing the amount of extra area added by voters when applying TMR. A suite of 15 circuit benchmarks including both real-world and synthetic designs was used in the experiments.

5.1 Procedure

The experiments involved applying TMR to each of the test designs using each synchronization voter insertion algorithm. The toolflow used to apply TMR and determine the timing performance and area of each design is shown in Figure 2. The toolflow executes only up to the place and route phase, since at this point the timing performance and area of the resulting circuit can be determined. The number of voters inserted by each algorithm was recorded in addition to the number of logic slices consumed by the resulting design. The critical path length and area of each design after having TMR applied with each voter insertion algorithm were recorded and compared to the critical path length and area of the original, untriplicated design in addition to a version of the design that was triplicated without inserting synchronization voters. Critical path lengths were determined by repeating the place and route process with successively tighter timing constraints until the place and route tool failed to generate a configuration capable of meeting the constraint. Timing constraints were adjusted in 0.1 ns intervals. In this manner, the tightest possible critical path length achievable by the place and route tool was determined for each iteration of each design, including the original, untriplicated version.

The primary target FPGA device for these experiments was the Xilinx Virtex 1000 (XCV1000-5-fg680). The Virtex 4 SX55 (XC4VSX55-10-ff1148) was also used for one of the test designs (*ssra_core*) because the circuit required a larger part than the V1000 when triplicated.

5.2 Results

Table 1 provides the critical path length, number of voters inserted, and number of slices used by each algorithm's version of the designs. The mean values for the critical path and number of voters are calculated over 14 of the benchmark designs[2]. The best algorithm's result for each row in the table is given in bold.

The results in Table 1 show that for the test designs in question, the algorithm that produced the best timing results overall is the *Voters After Every Flip-Flop* algorithm, which increased the critical path length of the design while adding TMR by only 15.3% over the original design and 9.8% over the triplicated version with no synchronization voters. The *Highest Flip-Flop Fanout* and *Highest Fan-in Flip-Flop Output* algorithms also provided very good timing results. It is interesting to note that the *Highest Fan-out* algorithm, which is the only algorithm that does not restrict voter placement to locations directly after flip-flops, provided the worst overall timing results. Although it tied for best timing performance on some of the benchmark designs, it never outperformed any of the other algorithms. Each of the other algorithms had at least one design where it was exclusively the best for timing performance. This result suggests that the heuristic of placing voters directly after flip-flops in order to place at most a single set of voters in any timing path is effective.

In terms of circuit area, Table 1 shows that the algorithm that induced the lowest increase on average in the number of slices used by a design is the *Highest Flip-Flop Fanout* algorithm. The *Highest Fan-out* and *Highest Fan-in Flip-Flop Output* algorithms both performed nearly as well. Interestingly, the *Voters After Every Flip-Flop* algorithm, which provided the best overall timing results, gave significantly poorer results in the area category.

Overall, the best combination of area and timing performance results is obtained by using the *Highest Flip-Flop Fanout* algorithm. Its timing results are nearly as good as those of the *Voters After Every Flip-Flop* algorithm, and its area results are far better. However, when sheer timing performance is the only concern, the *Voters After Every Flip-Flop* algorithm is the best choice in the average case.

Table 1 also reports the average run times of the 4 algorithms. As noted previously, the algorithmic complexities of the algorithms are very conservative upper bounds. Due to the nature of standard digital logic circuits, we expect these algorithms to scale much better than their complexities would imply. In practice, the feedback encountered in most digital circuits is simple enough for the algorithms to manage in reasonable time. The longest run time that we have encountered for any of these algorithms was 1600 s for a particularly complex design.

[2]The *blowfish* design was excluded from the mean calculations because it did not produce a full row of data. Two of the voter insertion algorithms inserted more voters in this design than could be mapped to the target device. These entries are marked with asterisks in the table.

6. CONCLUSION

When configuration bitstream scrubbing is employed together with triple modular redundancy, synchronization voters are essential for resynchronizing the TMR replicates when faults are corrected. Using the algorithms presented in this paper, it is possible to apply TMR and insert synchronization voters using an automated CAD tool. The best overall algorithm (considering both area and timing performance impacts) is the *Highest Flip-Flop Fan-out* algorithm. The *Voters After Every Flip-Flop* algorithm can provide slightly better timing results at the cost of increased area overhead due to a greater number of voters.

The experimental results obtained in this work indicate that in order to minimize the negative timing impact of TMR, voter insertion algorithms should limit voter locations primarily to flip-flop output nets. The algorithms in this paper that follow this policy increase the critical path length of a design by only 15.8% on average (over an untriplicated version), compared to 23.5% for the other algorithm. Although algorithms that perform the best on average in the timing performance and area categories have been identified, in cases where timing performance and area are critical factors in a space-based mission, several different voter insertion algorithms should be tried in order to determine the best algorithm for the particular circuit being implemented and the constraints of the mission.

7. REFERENCES

[1] David Ratter. FPGAs on Mars. Technical report, Xilinx, August 2004. XCell Journal #50.

[2] Michael Caffrey, Michael Wirthlin, William Howes, Daniel Richins, Diane Roussel-Dupre, Scott Robinson, Anthony Nelson, and Anthony Salazar. On-orbit flight results from the reconfigurable Cibola Flight Experiment satellite (CFESat). In *17th Annual IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 2009)*, pages 3–10, Napa, CA, April 2009.

[3] Brendan Bridgford, Carl Carmichael, and Chen Wei Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note XAPP987*, 1, 2008.

[4] Carl Carmichael, Earl Fuller, Phil Blain, and Michael Caffrey. SEU mitigation techniques for Virtex FPGAs in space applications. In *Proceedings of the Military and Aerospace Programmable Logic Devices International Conference (MAPLD)*, Laurel, MD, September 1999.

[5] Carl Carmichael. Triple module redundancy design techniques for Virtex FPGAs. Technical report, Xilinx Corporation, November 1, 2001. XAPP197 (v1.0).

[6] Xilinx TMRTool. Product Brief, Xilinx Corporation, 2006.

[7] C. Carmichael, M. Caffrey, and A. Salazar. Correcting single-event upsets through Virtex partial configuration. *Xilinx Application Notes, XAPP216 (v1. 0)*, 2000.

[8] F. Lima, C. Carmichael, J. Fabula, R. Padovani, R. Reis, X. Inc, and CA San Jose. A fault injection analysis of Virtex FPGA TMR design methodology. In *Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on*, pages 275–282, 2001.

[9] J. Heiner, N. Collins, and M. Wirthlin. Fault tolerant ICAP controller for high-reliable internal scrubbing. In *Proceedings of the Aerospace Conference*, pages 1–10, 2008.

[10] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, KA LaBel, M. Friendlich, H. Kim, and A. Phan. Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis. *IEEE Transactions on Nuclear Science*, 55(4 Part 1):2259–2266, 2008.

[11] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving FPGA design robustness with partial TMR. In *44th Annual IEEE International Reliability Physics Symposium Proceedings*, pages 226–232, 2006.

[12] R.M. Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, 43:85–103, 1972.

[13] G. Even. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.

[14] P. Eades, X. Lin, and W.F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[15] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67 – 72, 1981.

[16] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, Oct. 1971.

		Original (Untriplicated)	**TMR w/out voters**	**Voters After Every FF**	**Highest Fan-out**	**Highest FF Fan-out**	**Highest FF Fan-in Output**
blowfish	**Critical Path**	28.3 ns	27.2 ns	*	36.5 ns	31.7 ns	*
	Voters	-	-	8820*	954	**777**	7158*
	Slices	3416	10293	*	10742	10462	*
des3	**Critical Path**	11.1 ns	11.0 ns	13.6 ns	15.0 ns	13.6 ns	**13.5 ns**
	Voters	-	-	449	435	**353**	407
	Slices	658	2056	2312	2295	**2242**	2281
qpsk	**Critical Path**	80.0 ns	83.7 ns	85.4 ns	89.8 ns	**83.9 ns**	**83.9 ns**
	Voters	-	-	1752	165	**96**	111
	Slices	1041	3186	3901	3268	**3207**	**3207**
free6502	**Critical Path**	29.6 ns	30.9 ns	**31.5 ns**	39.6 ns	33.1 ns	32.5 ns
	Voters	-	-	465	**237**	264	267
	Slices	484	1485	1738	**1604**	1615	1616
T80	**Critical Path**	27.8 ns	29.2 ns	**32.4 ns**	36.1 ns	33.7 ns	34.1 ns
	Voters	-	-	828	573	**483**	645
	Slices	931	2831	3304	3111	**3079**	3193
macfir	**Critical Path**	14.4 ns	16.9 ns	**14.2 ns**	19.4 ns	19.5 ns	19.5 ns
	Voters	-	-	4224	**219**	**219**	**219**
	Slices	658	2445	3761	2571	**2566**	**2566**
serial_divide	**Critical Path**	9.2 ns	9.5 ns	**11.6 ns**	13.9 ns	12.2 ns	12.2 ns
	Voters	-	-	156	66	**60**	**60**
	Slices	40	129	209	166	**164**	**164**
planet	**Critical Path**	10.9 ns	10.8 ns	**12.5 ns**	12.6 ns	12.6 ns	12.6 ns
	Voters	-	-	21	**18**	**18**	**18**
	Slices	144	435	443	**441**	**441**	**441**
s1488	**Critical Path**	9.9 ns	10.3 ns	12.3 ns	**12.0 ns**	**12.0 ns**	**12.0 ns**
	Voters	-	-	21	**18**	**18**	**18**
	Slices	145	438	446	**444**	**444**	**444**
s1494	**Critical Path**	10.4 ns	10.7 ns	12.4 ns	**12.2 ns**	**12.2 ns**	**12.2 ns**
	Voters	-	-	21	**18**	**18**	**18**
	Slices	148	447	458	**456**	**456**	**456**
s298	**Critical Path**	15.8 ns	16.2 ns	19.4 ns	19.5 ns	**19.1 ns**	20.1 ns
	Voters	-	-	96	87	**84**	87
	Slices	517	1551	1594	1600	**1593**	1601
tbk	**Critical Path**	10.3 ns	10.6 ns	13.1 ns	**12.9 ns**	**12.9 ns**	**12.9 ns**
	Voters	-	-	201	**186**	**186**	**186**
	Slices	155	501	612	**603**	**603**	**603**
synthetic	**Critical Path**	9.9 ns	10.0 ns	13.9 ns	10.2 ns	10.4 ns	**10.1 ns**
	Voters	-	-	13877	**290**	326	**290**
	Slices	3061	12286	**12286**	**12286**	**12286**	**12286**
lfsrs	**Critical Path**	9.0 ns	10.8 ns	13.9 ns	13.9 ns	**12.7 ns**	12.9 ns
	Voters	-	-	5400	**360**	450	450
	Slices	1195	7429	**6468**	7658	7578	7578
ssra_core	**Critical Path**	6.1 ns	6.5 ns	7.2 ns	7.0 ns	7.2 ns	**6.9 ns**
	Voters	-	-	30270	684	**636**	696
	Slices	5393	18651	28033	18793	**18745**	18865
Mean critical path length		18.17 ns	19.08 ns	**20.96 ns**	22.44 ns	21.08 ns	21.10 ns
% Increase over original		-	5.0%	**15.3%**	23.5%	16.0%	16.1%
% Increase over TMR w/out voters		-	-	**9.8%**	17.6%	10.5%	10.6%
Mean number of voters		-	-	4127.2	239.7	**229.4**	248.0
Mean number of slices		1040.7	3847.9	4683.2	3949.7	**3929.9**	3950.1
% Increase over original		-	269.7%	350.0%	279.5%	**277.6%**	279.6%
% Increase over TMR w/out voters		-	-	21.7%	2.6%	**2.1%**	2.7%
Mean Run Time		-	-	0.4 s	11.0 s	5.9 s	116.7 s

Table 1: Critical path length, number of voters, number of slices induced, and average run time of each voter insertion algorithm.