TLegUp: A TMR Code Generation Tool for SRAM-Based FPGA Applications Using HLS

Ganghee Lee*, Dimitris Agiakatsikas*, Tong Wu*, Ediz Cetin†, and Oliver Diessel*
*School of Computer Science and Engineering, UNSW Sydney
†Department of Engineering, Macquarie University Australia
{ganghee.lee, d.agiakatsikas, tong.wu, o.diessel}@unsw.edu.au, ediz.cetin@mq.edu.au

Abstract—**We present TLegUp, an extension of LegUp, that automatically generates Triple Modular Redundant designs for FPGAs from C programs. TLegUp is expected to improve the productivity of application designers for space; to allow designers to experiment with alternative application partitioning, voter insertion and fault-tolerant aware scheduling and binding algorithms; and to support the automatic insertion of the infrastructure needed to run a fault-tolerant system. In this paper, we examine TLegUp's capacity to make use of both combinational and sequential voters by triplicating a design before scheduling and binding occur. In contrast, traditional RTL-based tools are constrained to use only combinational voters so as to preserve the scheduling and binding of the design; critical path lengths are consequently increased. We compare the use of sequential and combinational voters for a range of benchmarks implemented on a Xilinx Virtex-6 FPGA in terms of: (i) maximum operating frequency; (ii) latency; (iii) execution time; and (iv) soft-error sensitivity. Compared to the use of combinational voters, the use of sequential voters reduces the application execution time on the CHStone benchmark suite by 4% on average.**

I. Introduction

Due to their performance/power characteristics, low non-recurring engineering costs and re-configurability, commercial SRAM-based Field Programmable Gate Arrays (FPGAs) are becoming key components for implementing payload for space applications [1]. This trend is enhanced by the development of sophisticated FPGA High Level Synthesis (HLS) tools that enable researchers and practitioners to increase their productivity and rapidly realize high-performance and energy efficient hardware [2]. Although, HLS is not a new concept, it has only recently started gaining traction in the FPGA design industry, selectively, or completely replacing Hardware Description Languages (HDL), such as Verilog, in which a digital circuit is described at the *Register Transfer Level* (RTL) rather than at the more abstract *algorithmic level*.

Unfortunately, radiation experiments [3] and spacecraft applications [4] show that systems implemented on SRAM-based FPGAs are susceptible to radiation-induced errors, or Single Event Upsets (SEUs) in their user and configuration memory [4]. While one-time programmable or FLASH-based FPGAs are used for space applications because of their immunity to SEUs affecting their configuration memory, these devices have considerably lower capabilities than SRAM-based FPGAs [4], including lower performance, logic capacity and flexibility. Many SEU mitigation techniques have been introduced to overcome the effects of SEUs in SRAM-based FPGAs [4]. The most common way to implement a reliable SRAM-based FPGA system that tolerates SEUs is to follow a Triple Modular Redundant (TMR) design methodology combined with configuration memory error correction techniques [5], [6]. Yet, manually triplicating a design is tedious and error prone, especially when dealing with complex applications.

Commercial and academic tools have therefore emerged to automate the triplication process of FPGA applications. Examples are the commercial tools Synopsis Synplify Premier, Mentor Graphics Precision Hi-Rel, Xilinx TMRtool, and the academic tool BL-TMR [3], whereby the latter two tools support only a limited number of Xilinx FPGA parts. These tools usually analyze, modify and apply fine- or coarse-grained triplication of the design in RTL, during the synthesis or post-synthesis phase of the Computer-Aided Design (CAD) flow. Triplicating a design in RTL, though, has the considerable drawback that the circuit's schedule needs to be preserved while voters are inserted into the data- and control-paths; the design's critical path length is consequently increased and flexibility regarding voter placement and the ability to retime the design are hampered. Recently the use of HLS to generate fault-tolerant FPGA designs [7], [8] was reported in the literature. However, these approaches aim to minimize the area overhead by using a partial triplication without considering the performance degradation due to voter insertion.

We report preliminary results of using HLS to automatically generate triplicated FPGA circuit designs. We propose TLegUp, which extends the LegUp HLS framework [9] in order to generate synthesizable Verilog RTL suitable for implementation in Altera and Xilinx FPGA-based systems. The triplication of the design is performed within the Low-Level Virtual Machine (LLVM) [10] compiler Intermediate Representation (IR) of the LegUp flow, before synthesis, scheduling and binding take place. This approach allows users to implement custom voter insertion algorithms, to experiment with partitioning the design in order to enhance reliability, and provides opportunities to automatically instantiate the infrastructure, such as reconfiguration controllers or scrubbers, to support the reliable running of the TMR systems.

The contributions of this paper are: (i) we describe the TLegUp tool to automatically generate triplicated RTL designs from a C language program using HLS, (ii) we outline an approach for inserting combinational voters into feedback paths within a circuit – these voters are crucial to resynchronizing a design after configuration memory errors have been corrected, (iii) we detail how pipelined sequential voters, which are very challenging to add using RTL-based tools since pipelining changes the timing of the control signals, can be seamlessly

This research was supported in part by the Australian Research Council's Linkage (LP140100328) and Discovery (DP150103866) Projects funding schemes.

added using TLegUp. We assess the performance of TLegUp in producing triplicated versions of standard benchmarks using both combinational and sequential voters, (iv) we conduct fault-injection experiments on Virtex-6 implementations of the benchmarks in order to evaluate the SEU masking effectiveness of TLegUp generated designs.

II. Background

A. SEU Mitigation Techniques for SRAM-based FPGAs

SRAM FPGAs utilize SRAM cells to store their configuration, i.e. the memory that configures the FPGA's prefabricated hardware blocks in order to implement the desired circuit or system, while a smaller portion of SRAM memory is used to store the system's state and data (referred to as user memory in this work). However, the bistable values of the FPGA's SRAM cells can change state when they are bombarded by heavy ions and protons in space [4]. This leads to corruption of the system's state or data when SEUs occur in the FPGA's Flip Flops (FFs) or Block RAMs (BRAMs). These errors are corrected when they are overwritten by new results. However, if an SEU occurs in a utilized Configuration Memory (CM) cell it may lead to a corruption of the design itself, and the system will only recover when that particular configuration cell is overwritten with its correct value.

Employing TMR by triplicating the design and applying majority voting on the outputs of each module (one of the three replicated, functionally identical designs) ensures that the memory and CM errors are masked until error recovery takes place [3], [5]. Two recovery methods are commonly used, whereby either the entire FPGA is periodically scrubbed to overwrite erroneous configuration data, regardless of errors being present or not, or the configuration of an individual module that is found to be in error is dynamically rewritten [6]. In order to avoid single points of failure, the majority voters, that are used to mask errors in the user circuit and trigger the recovery of erroneous modules, are also commonly triplicated. Particular care is needed to also vote on internal re-entrant or feedback signals that form cycles within the logic [11]. This ensures that the state of modules that have undergone CM error correction can be re-synchronized with their siblings [12]. We refer to these voters as *feedback voters* (FV) [11]. A design may also be partitioned into several TMR sub-components so as to reduce the likelihood of multiple errors affecting more than one of the modules of a TMR component [6] and to improve the recovery time when module-based error recovery is employed [12]. We refer to these voters as *partitioning voters* (PV). Prior to routing outputs off chip, so called *reducing voters* (RV) may be needed to determine the output pin values.

B. LLVM IR and LegUp

LLVM IR is a machine-independent RISC-like instruction set, which is easily represented as a Control Flow Graph (CFG), where the nodes represent Basic Blocks (BBs) and the edges represent control transitions between BBs. Nodes are composed of a sequence of LLVM IR instructions with a single entry and exit. In LegUp [9] the hardware constructs corresponding to all LLVM instructions are pre-synthesized for the target FPGA in all supported bit-widths so that the FPGA resources needed to implement the instruction and the associated delay can be determined. This FPGA-specific information is used to annotate each LLVM instruction to make early circuit speed and area predictions.

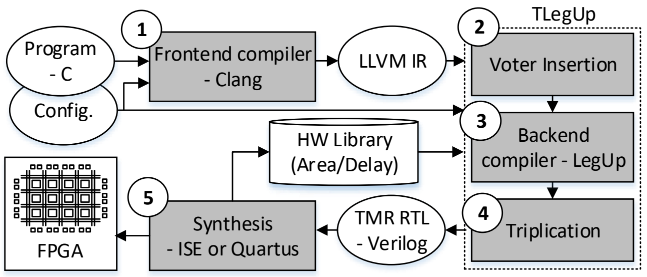

Fig. 1: TLegUp design flow

III. TLegUp

A. Design Flow

Fig.1 illustrates the TLegUp design flow. TLegUp takes as input a standard C program and a configuration file, which contains user directives regarding the target architecture, clock period, and pipeline support, etc. We added two directives that enable TMR emission and allow the voter type, combinational or sequential, to be selected. According to these directives TLegUp can generate simplex (non-TMR) RTL, triplicated RTL with combinational or triplicated RTL with registered (sequential) voters.

Referring to Fig. 1, at step ①, TLegUp compiles the C code and generates LLVM IR using the Clang frontend compiler. At step ②, voter locations in a circuit are determined on the LLVM IR based on user directives in the configuration file. At step ③, HLS as a backend compilation step is accomplished with LegUp, which performs scheduling and binding with allocated resources while considering the voter latency. At step ④, triplication with pre-located voters is performed and synthesizable Verilog RTL code is generated. Note that the voter locations are identified before HLS at ③ and that the triplication process is deferred until after HLS, to guarantee the concurrent execution of TMR replicas, and prevent common sub-expression elimination between TMR modules. At step ⑤, the triplicated RTL produced by TLegUp is synthesized to an FPGA implementation using standard tools such as Xilinx ISE or Altera Quartus. Since TLegUp is based on LLVM, which is a source and target independent IR, it can be extended to support other input or output languages.

B. Voter Insertion Algorithm

TLegUp triplicates the modules, FSMs, memories, registers and wires, but not the top module's I/O pins such as clock and reset signals. TLegUp currently supports automatic voter insertion at the following four locations in a circuit as shown in Fig. 2: 1) feedback paths of an application's datapath; 2) application FSMs; 3) memory output signals; and 4) output ports of the top module. Voters at locations 1) and 2) are classified as feedback voters (FVs), voters at location 3) are partitioning (or feedforward) voters (PVs), and those at location 4) are reducing voters (RVs).

1) *Feedback paths of an application's datapath*: When an application is translated to LLVM IR via the frontend compiler, we perform dataflow analysis using a depth first search in order to detect feedback paths in the datapath. We insert

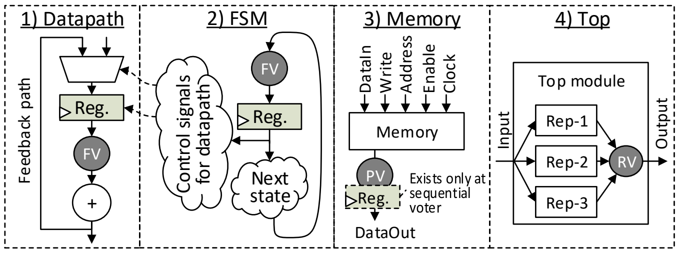

Fig. 2: Voter locations and types

FVs at the register outputs of the immediate successor of the first detected feedback edge in each case. Since TLegUp is based on an HLS approach, the user can specify for TLegUp to implement combinational or sequential voters via a user directive. However, the addition of registers on feedback paths has the effect of increasing the initiation interval of pipelined designs and increased execution time when used in loop bodies. In our experiments, we found that the use of sequential FVs increased the average execution time of the tested benchmarks by about 17% compared with the use of combinational FVs.

2) *Application FSM*: We only use combinational voters for the feedback paths in FSMs since in digital systems the FSM registers must be updated every clock cycle. We insert voters between the next state logic and the state registers since the paths between the state register outputs and the datapath register inputs are often timing-critical in LegUp.

3) *Memory output signals*: Since dense BRAMs are relatively more exposed to SEUs [13], we isolate them from the datapath to prevent error propagation. We do this by triplicating the memory blocks and inserting combinational or sequential voters (depending upon the user directive) on the memory output signals. Sequential voters increase the number of clock cycles to deliver the final result of an application. However, they may reduce the critical path delay, and thereby reduce the application's final execution time. Inserting sequential voters in feedforward paths can be especially useful for pipelined designs, since they can increase the clock frequency of the design without changing the application's throughput.

4) *Output ports of the top module*: RVs are inserted on the output ports of the top module of the design hierarchy. We use combinational voters for the output ports since they are not usually on the critical paths of designs.

In previous work [3], where triplication was applied to RTL, the tools were prevented from using sequential voters so as to preserve the existing application schedule. The critical path lengths of the resulting systems are often therefore increased. Since we perform triplication before HLS, it is possible to insert sequential voters to reduce the critical path length of signals. More specifically, the triplication of the design is carried out on the LLVM IR, before the scheduling and binding phases occur. Therefore, any timing or resource assignment effect of inserting sequential voters is accounted for when the application is scheduled. Another benefit of using HLS is that we can consider the clock frequency margin for TMR. Since TMR designs use more resources than simplex designs, they are expected to achieve lower clock frequencies [14]. Experiments we conducted on triplicated designs without voters observed a 26% average reduction in clock frequency compared with their corresponding simplex designs due to increased routing delays. To reflect these increased routing delays, we targeted a 10ns clock period for the circuits that incorporated sequential voters and used a 15ns target clock period for all other design.

IV. Experimental Results

We compared simplex and triplicated versions of the HLS CHStone benchmarks [2] in terms of resource utilization, maximum operating frequency (FM) in MHz, application latency (LAT) in number of clock cycles, execution time (ET) in ms, essential bits (EB) in Mbits, and soft error sensitivity (SES). See Table I.

We performed two sets of experiments. In the first set we 1) produced the simplex and triplicated RTL versions of the HLS benchmarks with LegUp and TLegUp, 2) performed functional simulation to verify and obtain the latency (LAT) required for each benchmark to finish its computation, and 3) synthesized, placed and routed them on a Xilinx Virtex-6 FPGA with Xilinx ISE 14.7, in order to extract post-routed resource utilizations, i.e. numbers of occupied Look Up Tables (LUT) and slice registers (REG). The numbers of DSPs and memories are not presented in Table I as these tripled precisely in number. We calculated ET as $\mathrm{FM}^{-1} \times \mathrm{LAT}$. Note that we used the default optimization settings of LegUp and TLegUp to compile the HLS benchmarks, except for the pipeline, bitwidth and memory merge optimizations, which were enabled. We disabled all resource sharing options in the Xilinx ISE to prevent common sub-expression elimination between different TMR replicas, while the optimization strategy of the Xilinx synthesizer was set to *"speed"*, since this is the default configuration of the official LegUp framework.

In the second set of experiments we re-implemented the benchmarks, but also incorporated a Microblaze (MB) soft-processor to facilitate fault-injection. The MB was implemented in one of the twelve available clock regions of the FPGA, while the benchmarks were implemented (and isolated from the MB) in the remaining eleven clock regions. All benchmarks included test vectors that were stored in BRAM and incorporated three 1-bit input ports, clock, reset and start, as well as two output ports: finish (1-bit) and result (32-bit). The MB was programmed to inject a fault into the CM of each benchmark (excluding the MB's clock region) and then to test the benchmark's functionality by setting its start bit, logging its output when finished and comparing the results with the golden value for the benchmark.

Not all corrupted CM bits result in a functional error, e.g. soft-errors in the CM bits of unused resources do not affect the functionality of the design. In order to speed up the fault injection campaign, we used the SEVAX tool [15] to extract information from the Xilinx essential bits file, which indicates the essential CM bits (EB) that may affect a circuit's functionality when corrupted, and injected faults into each of these EB. The SES of each benchmark was calculated as the fraction of injected faults that resulted in observed functional errors, i.e. SES = number of functional errors / EB. It should be noted that in this second set of experiments, the jpeg benchmark was unable to complete routing due to increased resource utilization and therefore fault-injection for this benchmark was not performed. Similarly, the sha and bfish benchmarks are not reported upon since testing had not completed by the time of going to press.

TABLE I: Simplex vs. Triplicated Designs

Benchmark	Simplex							TMR* + Combinational						TMR* + Sequential						
	REG	LUT	FM	LAT	ET	EB	SES	REG	LUT	FM	ET	EB	SES	REG	LUT	FM	LAT	ET	EB	SES
adpcm	7,701	8,850	61	8,704	142	5.48	2.75	3.05	3.84	0.70	1.42	2.04	0.38	3.06	3.94	0.75	1.00	1.33	2.08	0.40
aes	7,182	8,690	145	1,481	10	4.80	1.18	2.96	3.12	0.62	1.61	1.62	0.59	3.06	3.12	0.70	1.01	1.43	1.61	0.27
aesdec	7,883	9,419	146	3,219	22	4.94	1.48	2.96	3.10	0.67	1.50	1.62	0.19	3.14	3.08	0.64	1.01	1.57	1.63	0.24
gsm	4,459	6,458	108	4,763	44	4.88	0.84	2.87	3.30	0.59	1.68	1.56	0.13	2.98	3.52	0.73	1.23	1.68	1.64	0.06
sha	2,471	2,695	203	167K	820	-	-	3.05	3.69	0.54	1.86	-	-	3.15	3.63	0.68	1.25	1.85	-	-
bfish	3,983	4,276	180	164K	909	-	-	2.89	3.65	0.58	1.72	-	-	2.96	3.64	0.59	1.11	1.87	-	-
dfadd	2,420	5,625	147	643	4	4.59	0.53	2.83	3.10	0.63	1.58	1.40	0.07	3.20	3.19	0.67	1.20	1.78	1.42	0.07
dfdiv	12K	13K	99	1,908	19	5.59	0.76	2.97	3.10	0.81	1.23	1.74	0.02	3.05	3.14	0.80	1.03	1.28	1.74	0.02
dfmul	1,467	2,428	101	208	2	4.17	0.29	2.89	3.10	0.86	1.17	1.23	0.10	3.75	3.19	0.98	1.27	1.30	1.24	0.08
dfsin	16K	22K	92	57K	623	7.06	5.26	2.91	3.08	0.73	1.36	2.07	0.03	3.01	3.14	0.64	1.03	1.60	2.09	0.04
jpeg	22K	35K	52	1.2M	23K	-	-	3.01	3.33	0.70	1.42	-	-	3.07	3.33	0.72	1.01	1.42	-	-
mips	845	2,240	81	5,004	62	4.16	0.43	2.70	3.39	0.95	1.05	1.22	0.15	3.28	3.16	1.22	1.24	1.02	1.23	0.09
motion	7,630	12K	111	8259	74	5.50	0.26	3.06	3.42	0.61	1.63	1.86	0.05	3.08	3.49	0.67	1.00	1.50	1.85	0.04
satd	2,598	3,506	120	102	1	4.19	0.40	2.97	3.20	0.88	1.14	1.26	0.04	3.12	3.19	1.31	1.08	0.82	1.13	0.31
sobel	622	766	83	1.0M	13K	4.00	0.99	3.07	3.42	0.62	1.60	1.16	0.36	3.45	3.47	1.08	1.00	0.93	1.16	0.33
bmford	1,104	1,375	128	465	4	3.97	0.25	2.82	3.56	0.71	1.40	1.14	0.10	3.20	3.54	0.89	1.46	1.63	1.15	0.05
mmult	395	487	99	10K	102	3.85	0.15	2.93	3.21	0.93	1.07	1.05	0.15	3.43	3.25	1.29	1.04	0.81	1.06	0.14
Geomean	3,435	4,791	108	7,942	73	4.72	0.66	2.94	3.32	0.71	1.42	1.46	0.11	3.17	3.35	0.82	1.11	1.36	1.46	0.11

*TMR results are normalized to the Simplex design.

Table I compares the results between simplex and triplicated RTL designs. The TMR columns in Table I are scaled (normalized) relative to the results for the simplex designs. We evaluated two TMR variations: (1) triplicated RTL designs with only combinational voters (*'TMR+Combinational'* column), and (2) triplicated RTL designs with sequential voters in the feedforward paths and combinational voters elsewhere (*'TMR+Sequential'* column). Note that the *'TMR+Combinational'* results do not include a LAT column since the latency of these designs is the same as for the corresponding Simplex designs, i.e. relative to Simplex, LAT = 1. As can be observed from Table I, the use of sequential voters instead of combinational ones in TMR implementations reduces the ET on average by 4% while respectively requiring 1.42x and 1.36x more time to execute their tasks when compared to the simplex versions. A three-fold increase is observed in utilized resources for TMR implementations, as expected, while the number of EBs only increased by 1.46x. Further, we found the SES of TMR implementations to be 9.09x lower than the corresponding simplex version, which in our opinion is a smaller reduction than expected. We believe that the SES could be further reduced if the TMR implementations were placed and routed in such a way so as to reduce the chance of having more than one TMR module failing due to a single bit error [16].

V. Conclusions and Future Considerations

In this paper we introduced TLegUp, a TMR code generation tool for SRAM FPGA-based applications using HLS. Previous RTL-based TMR approaches are restricted to using only combinational voters to preserve the pre-defined scheduling and binding. However, since in TLegUp voter insertion is considered before scheduling and binding, we can make use of sequential voters and achieve an improvement of 16% in clock frequency and 4% in an application's execution time.

Future work will consider i) enhancement of voter insertion algorithms, ii) developing task partitioning algorithms to increase the fault-tolerance of the design and to minimize the recovery time when a module-based error recovery scheme is adopted and iii) developing FPGA layout algorithms to apply module based error recovery to a TLegUp synthesized circuit.

References

[1] *NASA Technology Roadmaps - TA 11: Modeling, Simulation, Information Technology, and Processing*. National Aeronautics and Space Administration (NASA), 2015.

[2] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[3] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin, "SEU-induced persistent error propagation in FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2438–2445, 2005.

[4] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Computing Surveys*, vol. 47, no. 2, p. 37, 2015.

[5] C. Carmichael, "Triple module redundancy design techniques for Virtex FPGAs," *Xilinx Application Note XAPP197*, vol. 1, 2001.

[6] D. Agiakatsikas, E. Cetin, and O. Diessel, "FMER: A hybrid configuration memory error recovery scheme for highly reliable FPGA SoCs," in *Field Programmable Logic and Applications (FPL)*, 2016.

[7] S. T. Fleming and D. B. Thomas, "StitchUp: Automatic control flow protection for high level synthesis circuits," in *Design Automation Conference (DAC)*, 2016.

[8] A. Shastri, G. Stitt, and E. Riccio, "A scheduling and binding heuristic for high-level synthesis of fault-tolerant FPGA applications," in *Application-specific Systems, Architectures and Processors (ASAP)*, 2015.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, p. 24, 2013.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, 2004, p. 75.

[11] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for FPGA designs using triple modular redundancy," in *Field-Programmable Gate Arrays (FPGA)*, 2010.

[12] E. Cetin, O. Diessel, L. Gong, and V. Lai, "Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration," in *Field Programmable Logic and Applications (FPL)*, 2013.

[13] H. Quinn and P. Graham, "Terrestrial-based radiation upsets: A cautionary tale," in *Field-Programmable Custom Computing Machines (FCCM)*, 2005.

[14] E. Cetin, O. Diessel, and L. Gong, "Improving Fmax of FPGA circuits employing DPR to recover from configuration memory upsets," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015.

[15] A. Sari, D. Agiakatsikas, and M. Psarakis, "A soft error vulnerability analysis framework for Xilinx FPGAs," in *Field-Programmable Gate Arrays (FPGA)*, 2014.

[16] L. Sterpone and M. Violante, "A new reliability-oriented place and route algorithm for SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 732–744, 2006.