Redundant Data Types and Operations in HLS and their Use for a Robot Controller Unit Fault Tolerance Evaluation

Jakub Lojda, Jakub Podivinsky, Zdenek Kotasek
Faculty of Information Technology, Brno University of Technology,
Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
{ilojda, ipodivinsky, kotasek}@fit.vutbr.cz

Abstract

Some environments (e.g. space, aerospace or medical systems) require electronic systems to withstand an increased occurrence of faults. Moreover, the failure of these electronic systems can cause high economical losses or endanger human health. Fault tolerance is one of the techniques, the goal of which is to avoid such situations. This paper presents an approach to evaluate the degree of importance of individual system partitions when High-Level Synthesis (HLS) methodology is used. The importance of individual partitions was evaluated by the usage of our approach to fault-tolerant datapaths *design which is based on the HLS input specification modification. The partitions are formed by sets of variables and operations. A brief description of the approach to fault tolerance in HLS is shown in the paper as well. Our experiments are evaluated using an SRAM-based FPGA evaluation platform which allows us to analyze fault tolerance properties of the Design Under Test (DUT). In the evaluation platform, functional verification in combination with fault injection is utilized.*

1. Introduction

In our everyday lives we meet various types of electronic systems. Some of them are used just for entertainment purposes, others help us to make our life easier, but some of them are very important and safety critical because they control processes the failure of which can result in injury, heavy financial losses or can endanger human health. Medical equipment, space and aerospace control systems or automotive safety assistants can serve as examples of safety critical systems. It is very important to protect these systems against faults and ensure their reliable operation in every situation.

The technique called *fault tolerance* [8] is one of main approaches to increase electronic systems reliability. Fault tolerance accepts the fact a fault can appear, but the goal of this approach is to keep the system functional even with the presence of faults. Fault tolerance is a widely used technique which is usually based on various types of redundancy. The most common are *area* and *time redundancy*. *Area redundancy* usually uses n-copies of the same functional unit and a comparator to guarantee the proper function. On the other hand, *time redundancy* is based on repeated computation and the results from the independent runs are then compared.

The main subject of our research in the field of fault tolerant systems design and evaluation are SRAM-based *Field Programmable Gate Arrays* (FPGAs). FPGAs are composed of reconfigurable blocks and an interconnection network. The SRAM memory, in which the configuration bitstream is saved, is sensitive to *Single Event Upsets* (SEUs), which are caused by charged particles [16]. The goal of the research presented in this paper is to evaluate the importance of various high level *storage elements* and associated *operations* using their impact on reliability improvement. Knowing the importance of various partitions is essential in ensuring the highest level of reliability in cases where the remaining chip-area is limited. The fault tolerance method is based on the modification of an input algorithm before it is processed by HLS. For the evaluation, the approach of SEU injections in combination with our evaluation platform is used.

In this paper, the concept of HLS can be understood as a set of methods transforming a *high-level* de-

978-1-5386-3299-4/17/$31.00 ©2017 IEEE

scription to its implementation on the *Register Transfer Level* (RTL). The description is made on a high level of abstraction, usually in the form of an algorithm described in one of the higher-level programming languages (e.g. *C++*). The resulting RTL implementation is dependent on the configuration of the HLS. The HLS tools usually incorporate an ability to effectively explore the state space of all possible configurations. There are various parallelization techniques in HLS, although we only consider the most important the loop acceleration techniques such as *loop pipelining* and *loop unrolling*. These are usually fully exposed to the designer. The HLS resulting RTL is usually composed of the so-called *control-path*, which is usually in the form of a *Finite State Machine* (FSM), and the so-called *data-path*, which contains all the data processing hardware such as *Arithmetic Logic Units* (ALUs).

These days a lot of effort in the research of fault tolerance in HLS is dedicated to *data-path* hardening. The authors of [5] present a heuristic algorithm for searching an optimal assignment of operations to *data-paths* while considering the maximal cycle length of the transient fault. The authors of [14] show that in most cases, 100% fault coverage is not necessary. This fact allows them to implement a higher degree of freedom into the phases of *scheduling* and *binding* and save HW resources. The authors of [7] present a *two-phase* resource binding heuristic algorithm with considerable processor time and memory usage reduction in the phase of system design. An approach to error detection of arithmetic oriented *data-paths* is presented in [1]. The authors of [13] show a method of detecting *multi-cycle* transient faults while connecting the higher-level synthesis with the lower (*physical*) level and reducing the overhead. A new approach for the fault-tolerant HLS controller is shown in [6]. The authors of this method show that their approach requires less overhead resources than using the TMR. All the methods presented rely on a modification of the HLS methodology, but in our approach we are trying to strictly separate from the HLS tool itself while keeping all the benefits of the HLS.

This paper is organized as follows. An overview of our fault tolerance method based on data types modifications is proposed in Section 2. The experimental system and evaluation platform are presented in Section 3. The case study and experimental results are summarized in Section 4. Section 5 contains the conclusion of the paper and presents our plans for future research.

2. HLS-based Fault Tolerance Approach

In our approach we modify the input specification of the HLS to achieve a fault-tolerant system at the output of the HLS methodology. As this approach works at the level of abstraction of the input specification, it profits from all the benefits of the HLS. This idea was already presented in our previous work [10], although in this paper the method is used to evaluate the importance of various partitions of the DUT.

In the *C++* language code, three places to make modifications can be distinguished: 1) *data types*; 2) *arithmetic and logic operations*; and 3) *flow control statements*. This research is focused on the *data types* (DTs) and *operations* modifications. The method of creating new DTs, which we call the *Redundant Data Types* (RDTs), will be shown on the well known principle of the *Triple Modular Redundancy* (TMR). RDTs are using already existing (in the following text referenced as *original*) DTs, where each RDT expresses one method of fault tolerance (e.g. the RDT *triple* expresses TMR). This approach allowed us to modify the semantics of corresponding DT operations and to implement fault tolerance methods into these operations. As a result, all the operations whose operands include at least one RDT are modified according to the particular fault tolerance method of the RDT(s).

In the phase of a new RDT creation, three types of operations considering their *arity* must be addressed: 1) *unary*; 2) *binary*; and 3) *ternary*. While in the case of *unary* operations there is no need to consider another DTs existence, in the case of *ternary* (*conditional*) operations, however, an ability to cast the RDT variable to the *Boolean* DT must be added to provide the ternary operator with a *Boolean* value in order to evaluate the *conditional expression*. In the case of *binary* operations, operands of multiple combinations of DTs or RDTs may arise. These combinations include: a) *intra-data type* operations – RDT vs. RDT of equivalent redundancy types (e.g. TMR vs. TMR subsystem); b) *inter-data type* operations – RDT vs. RDT of different redundancy types – (e.g. TMR vs. *duplex* subsystem); and c) *original-data type* operations – RDT vs. it's *original* (*unhardened*) DT (e.g. TMR vs. *unhardened* subsystem). Examples of the combinations of RDT and DT operations are shown in Figure 1. Each time a new RDT is added, all existing RDTs must be updated to address all of the newly arisen *inter-data type* operation combinations.

With our approach it is relatively easy to incorporate fault tolerance into systems already described, although the decision on which RDT to apply to which

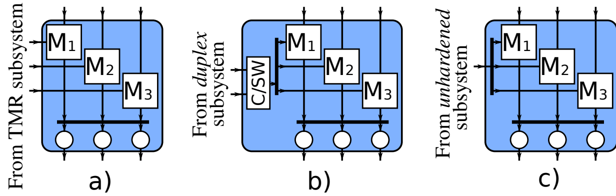

Figure 1. Three types of cases that can be distinguished when considering binary operations, intra-DT operation between two TMR subsystems (a), inter-DT operation between system with TMR and duplex hardening (b) and original-DT operation between TMR and unhardened subsystems (c).

variable might not be a trivial problem in cases where the whole system contains hundreds or even thousands of program variables. For such cases, our goal is to develop an automated methodology that would contain guides to calculate weights for algorithm operations and thus corresponding variables to estimate the impact of a particular modification of the input code. We assume that when a limited amount of HW resources is a concern, the selection of the amount of redundancy for each component of the system should be in correlation with its *importance*. To express the changes made in DTs of the input algorithm, an extension of *Activity Diagram* (AD) from *Unified Modeling Language* (UML), which is described in its original form in [2], could be used. For each action of the AD, the extension assigns a corresponding set of variable instance names utilized, their *original* DTs and eventually the RDTs replacing them. An example of the extended AD is shown in Figure 2. The example contains one action with two corresponding *original* variables and their DTs. The corresponding RDT if applied is listed for each variable on the right side of the vertical line. In this example of an AD, each variable is *hardened* with the `triple` RDT.

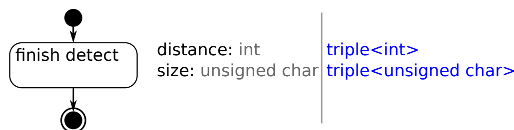

Figure 2. An example of the extended UML AD, to each action variable instances and corresponding DTs are assigned; RDTs if applied are listed on the right side of the line.

3. Experimental platform

The objective of our research comprehends not only the development and improvement of fault tolerance methodologies, but their evaluation as well. In our previous papers (e.g. [12]) the evaluation platform for checking the impact of faults was presented. Our evaluation platform uses *functional verification* [11] as a tool for monitoring the impact of faults injected into an electronic controller implemented into an FPGA. The main task of the functional verification is to check whether a verified circuit satisfies its specification. It compares outputs of a verified circuit running in the RTL simulator with a reference model. When fault injection is required, the DUT must be implemented into an FPGA. In this case the classical simulation-based functional verification is not used.

The architecture of our evaluation platform is shown in Figure 3. The two main parts are a computer and an FPGA development board with the robot controller. We use the ML506 board with the Virtex 5 FPGA, which allows us to inject faults directly into the FPGA. Another FPGA board (in the middle) serves as a bridge between the *Ethernet* interface and the *General Purpose Input Output* (GPIO) ports. The fault injector is one of the components running on the computer. Our fault injector [15] is based on the *Partial Dynamic Reconfiguration* (PDR) [17]. It reads part of the configuration bitstream from the configuration memory through the JTAG interface, then the requested number of specified bits of the bitstream is inverted and the modified bitstream is configured back to the configuration memory. The evaluation platform is able to use an electro-mechanical application as an experimental system, which allows us to monitor the impact of faults not only on the electronic controller, but also on the controlled mechanical parts. It should be noted that the simulation of the mechanical part is important and also runs on the computer. The electronic controller implemented into the FPGA is connected with the simulation of the mechanical part through the Ethernet interface. An evaluation of the impact of injected faults, both on the electronic and mechanical parts, is performed in the software of the verification environment, which runs on the computer.

We use a robot for the searching path through a maze and its electronic controller as an experimental electro-mechanical system. Our robot controller is implemented in VHDL which can be synthesized and configured in the FPGA. So, there are two possibilities on how to evaluate the fault tolerance methodologies: 1) apply a fault tolerance methodology to the implemented controller; or 2) create a new robot controller

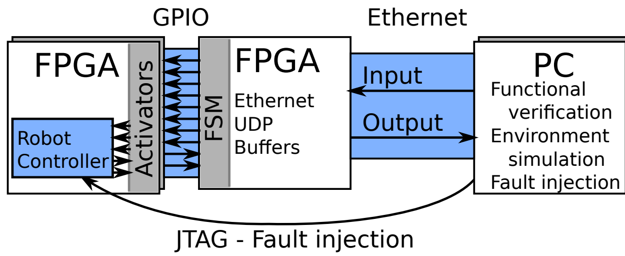

Figure 3. The architecture of our evaluation platform.

according to the evaluated fault tolerance methodology. The second approach is used in this work and we have implemented a new HLS-based robot controller using the HLS flow.

4. Case Study and Experimental Results

For the purpose of our approach evaluation, a robot controller was implemented according to HLS methodology. The input specification is written in the *C++* language and is based on the so-called *left-hand* algorithm which in the case of a crossroad in a maze always follows the *wall* on its left side. The *Player/Stage* [3] simulation environment is used to simulate the robot which has four sensors on its chassis, each facing one of the sides of the World.

In this research we are trying to evaluate the importance of each particular component of the robot controller unit and thus find out components that have the greatest potential in adding SEU resiliency if made fault tolerant. Our previous research was targeted towards an evaluation of the robot controller design fully hardened with our method when considering various HLS settings, while in this paper, the HLS settings remain constant and resilience against SEUs is always evaluated for designs with only one particular partition hardened.

An RDT named *triple*, which is based on the TMR principle, was used. For the purpose of evaluation, seven new robot controller unit versions were created using the *Catapult C University Version* tool [4] and synthesized with the *Xilinx Integrated Synthesis Environment* (ISE) [18]. Each controller of these versions has the proposed methodology applied to a different set of variables. These seven sets of variables are *mutually disjoint*. Since placing all of the seven extended ADs for each of the robot controllers to this paper would be very space-consuming, Figure 4 shows the extended AD of the robot controller unit having each variable *hardened*. The figure also highlights the seven variable sets with their proposed modifications. It is important to note that for each of the seven controller designs, only the corresponding set of variables was *hardened*. That is, for the controller unit version 1, only the set of variables marked by 1 (i.e. the `x_goal` and the `y_goal` variable) was *hardened*, while leaving the remaining variables *unhardened* for a particular design. Similarly, the other six versions were created. The HLS setup used during the synthesis is based on the *pipeline1-area* HLS settings set from our previous research which was submitted to [9]. The *pipeline1-area* settings comprehend the whole design *pipelined* with an *Initiation Interval* (II) of 1. As this HLS setup turned out to be the most sensitive to our approach (i.e. with the best reliability improvement gain when each variable used the *triple* RDT from all of the setups tested), we decided to use it in this research in order to achieve the best possible resolution when distinguishing among reliability improvements of various versions.

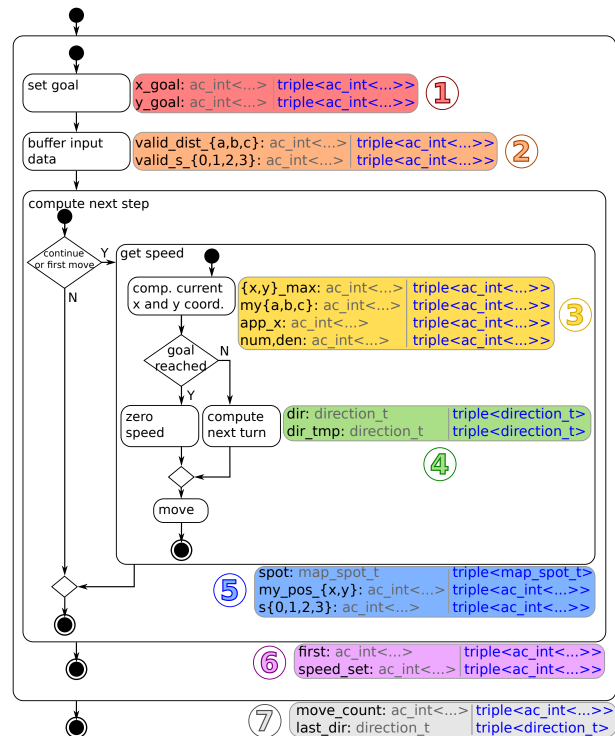

Figure 4. The extended UML AD of the robot controller having each variable hardened by the triple RDT with the variables divided into seven sets.

For each version of this robot controller, 2 000 eval-

uation runs were performed. The scenario of each run was as follows:

1. the robot controller unit was reinstated into its initial state, the maze map as well as its starting and target positions remained constant for all of the evaluation runs,
2. the *Player/Stage* simulation environment was started, the robot was placed on the starting position,
3. one SEU was injected into a bit of the bitstream, the bit was selected *uniformly at random* from all bits utilized as LUTs contents, the bit remained in a faulty state during the whole verification run,
4. the ability of the robot to reach the target position was monitored.

Table 1 shows the comparison of reliability gained for each robot controller version with the corresponding area overhead, which were calculated using the reference values of the unhardened version. The table also shows the numbers of *unary*, *binary* and *ternary* operations and numbers of *inter-*, *intra-* and *original-DT* operations associated with the hardened part using the *triple* RDT. The reliability improvement and the area overhead of each unit i were calculated using Equation 1 and Equation 2, respectively. The *reference* values of the unhardened version from our previous research were used for the calculation.

$$reliab_improv_i = \frac{failures_{ref} - failures_i}{failures_{ref}} * 100 \quad (1)$$

$$area_overhead_i = \frac{slices_{ref} - slices_i}{slices_{ref}} * 100 \quad (2)$$

As can be seen in Table 1, the designs with the higher overhead (numbers 2, 5 and 7) have achieved a higher level of reliability. In the other cases, our fault tolerant designs are even smaller than the *reference* unhardened design. In the cases of 1, 3 and 6, the smaller designs perform still better than the *reference* design. We believe this interesting behavior is caused by various pipelined settings the HLS tool chooses (i.e. various sizes of pipelined blocks the buffering registers are inserted in between). Further research is needed to find out the exact reason for this behavior, as it leads to significant reliability improvement when considering the chip-area consumed. It is important to note that

Table 1. The experimental evaluation of resources overhead, operations hardened and reliability gained in comparison with the unhardened reference values.

Robot Version	Ref.	1	2	3	4	5	6	7
LUTs bits [–]	21952	17408	55744	12800	15744	47552	15872	35840
Slices [–]	196	147	370	135	165	379	147	250
Failures [‰]	33.0‰	27.0‰	13.5‰	30.5‰	37.5‰	15.5‰	29.5‰	17.0‰
RDT ops. [–] *unary*	0	0	7	22	4	4	2	2
RDT ops. [–] *binary*	0	6	7	32	7	9	5	2
RDT ops. [–] *ternary*	0	0	0	0	0	0	0	0
RDT ops. [–] *inter-DT*	0	0	0	0	0	0	0	0
RDT ops. [–] *intra-DT*	0	0	0	30	4	0	0	2
RDT ops. [–] *orig.-DT*	0	6	14	24	7	13	7	2
Reliability improv. [%]	–	18.2%	59.1%	7.6%	-13.6%	53.0%	10.6%	48.5%
Area overhead [%]	–	-25.0%	88.8%	-31.1%	-15.8%	93.4%	-25.0%	27.6%

the operations are associated with variables of various bit-width lengths, thus the operations complexity varies. Therefore, the size of the resulting design does not correlate with the number of operations *hardened*. As part of our future research, we would like to try our approach with a different HLS tool that would allow us to to specify the synthesis parameters in a more detailed way. Our intention is to also research a method to estimate the importance of variables from the input algorithm without the need for the long process of the synthesis and fault injection.

5. Conclusions and Future Research

In this paper a newly emerging approach to fault tolerance of HLS-synthesized systems was briefly explained and evaluated after its application to a robot controller unit. The robot controller *C++* description was partitioned and into each partition a set of variables was assigned. Finally, each robot controller version, which included the corresponding set hardened by our approach, was evaluated. The experimental evaluation was performed using our evaluation platform, which incorporates injection into the utilized contents of FPGA LUTs. The main contribution of the paper is to demonstrate a way to evalate the importance of particular operation in HLS.

In the near-future, we would like to try our approach with a different HLS tool (possibly with some *open-source* alternative). As part of our future work, we would like to incorporate various fault tolerance methods and evaluate the impact of hardening other places in the *C++* code the fault tolerance could be applied to (mainly the control statements). And finally, from the results investigated, we would also like to focus on

the creation of a methodology to automate the whole process of design modifications and the proper form of redundancy selection.

Acknowledgements

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602 and BUT project FIT-S-14-2297.

References

[1] K. A. Campbell, P. Vissa, D. Z. Pan, and D. Chen. High-Level Synthesis of Error Detecting Cores Through Low-cost Modulo-3 Shadow Datapaths. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, pages 161:1–161:6, New York, NY, USA, 2015. ACM.

[2] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[3] B. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-robot and Distributed Sensor Systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.

[4] M. Graphics. Catapult HLS. `<https://www.mentor.com/hls-lp/catapult-high-level-synthesis/>`, 2017. Accessed: 2017-07-07.

[5] T. Inoue, H. Henmi, Y. Yoshikawa, and H. Ichihara. High-Level Synthesis for Multi-Cycle Transient Fault Tolerant Datapaths. In *2011 IEEE 17th International On-Line Testing Symposium*, pages 13–18, July 2011.

[6] T. Iwagaki, Y. Ishimori, H. Ichihara, and T. Inoue. Designing Area-Efficient Controllers for Multi-Cycle Transient Fault Tolerant Systems. In *2015 20th IEEE European Test Symposium (ETS)*, pages 1–2, May 2015.

[7] M. Kaneko and Y. Tsuboishi. Constrained Binding and Scheduling of Triplicated Algorithm for Fault Tolerant Datapath Synthesis. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1448–1451, June 2014.

[8] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[9] J. Lojda, J. Podivinsky, Z. Kotasek, and M. Krcma. Data Types and Operations Modifications: A Practical Approach to Fault Tolerance in HLS. In *Paper submitted to EWDTS 2017, East-West Design & Test Symposium 2017*.

[10] J. Lojda, J. Podivínský, M. Krčma, and Z. Kotásek. HLS-based Fault Tolerance Approach for SRAM-based FPGAs. In *Proceedings of the 2016 International Conference on Field Programmable Technology*, pages 297–298. IEEE Computer Society, 2016.

[11] A. Meyer. *Principles of Functional Verification*. Elsevier Science, 2003.

[12] J. Podivinsky, O. Cekan, M. Simkova, and Z. Kotasek. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-Mechanical Applications. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 312–319. IEEE, 2014.

[13] A. Sengupta and D. Kachave. Generating Multi-Cycle and Multiple Transient Fault Resilient Design During Physically Aware High Level Synthesis. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 75–80, July 2016.

[14] A. Shastri, G. Stitt, and E. Riccio. A Scheduling and Binding Heuristic for High-Level Synthesis of Fault-Tolerant FPGA Applications. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 202–209, July 2015.

[15] M. Straka, J. Kastil, and Z. Kotasek. SEU Simulation Framework for Xilinx FPGA: First Step Towards Testing Fault Tolerant Systems. In *14th EUROMICRO Conference on Digital System Design*, pages 223–230. IEEE Computer Society, 2011.

[16] D. White. Considerations Surrounding Single Event Effects in FPGAs, ASICs, and Processors. `<http://www.xilinx.com/support/documentation/white_papers/wp402_SEE_Considerations.pdf>`, Mar. 2012. Accessed: 2016-09-15.

[17] XILINX. Partial Reconfiguration User Guide. `<http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf>`, Apr. 2012. Accessed: 2016-09-15.

[18] Xilinx. ISE Design Suite. `<https://www.xilinx.com/products/design-tools/ise-design-suite.html>`, 2017. Accessed: 2017-07-07.