Data Types and Operations Modifications: a Practical Approach to Fault Tolerance in HLS

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

Abstract

Some types of electronic systems are working in the environment with an increased occurrence of faults such as space, aerospace or medical systems. Faults in these systems can lead to the failure of the whole system and 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 fault-tolerant data-paths *design that is based on the modification of High-level Synthesis (HLS) input specification. The description and evaluation of the impacts of some HLS optimization methods are demonstrated in the paper as well. Higher reliability is achieved through the modification of input description in the C++ programming language which the HLS synthesis tools are based on. Our work targets SRAM-based FPGAs that are prone to Single Event Upsets (SEUs). For the evaluation of the proposed method we use our evaluation platform, which allows us to analyze fault tolerance properties of the Design Under Test (DUT). The evaluation platform is based on functional verification in combination with fault injection.*

1. Introduction

Nowadays, electronic systems are used in various devices which play an important role in our everyday lives. The increase of chip-level integration results in a higher susceptibility to faults. The number of digital systems with a high demand on reliability, such as medicine, space, industry, is growing as well. In these cases especially, reliability is very important because the consequences of failure can result in injury or heavy financial losses or can endanger human health. One of the main approaches to increase reliability is the so-called *fault tolerance* [10]. Fault tolerance accepts the fact a fault can appear, but the goal of this approach is to keep the system functional even in the presence of faults. Techniques based on various types of redundancy are used for this purpose. Many fault tolerance methodologies exist, which combine and improve these basic methods (e.g. the HW and *temporal* redundancy is combined in approach shown in [9]).

The HLS is a set of methods transforming a digital circuit description into its RTL representation. The architecture of a typical HLS-generated circuit is composed of the so-called *data-path* and *control-path* (the controller) [4]. The HLS tools usually allow the designer to explore the state space of various RTL realizations very effectively and easily. The main decisions, such as setting a degree of parallel computation of a programming loop (*unrolling*) or *pipelining* a programming loop, are still the designer's responsibility. These two optimization techniques are not nearly all used in the process of accelerating the resulting system, but we have chosen these as we believe these are the most important ones. The *unrolling* basically allows parallel execution of individual loop iterations. The parameter of *degree of parallel execution* expresses the number of iterations executed in parallel. The *pipelining* allows the designer to create a pipelined version of the loop. When the *pipelining* is applied, each *Initiation Interval* (II) the execution of one iteration is started.

These days a lot of effort in the research of *fault tolerance* in HLS is dedicated to *data-path* hardening. Specifically modified versions of HLS methodologies are usually used for this purpose. The method described in [7] is directed against transient faults that last for several clock cycles. Another heuristic algorithm based on two-phase *resource binding* was published in [8]. A heuristic-based method that was published in [14] en-

978-1-5386-3299-4/17/$31.00

ables designers to choose a trade-off between resources consumed, resulting system latency and redundancy. The authors of [2] present an approach to error detection of arithmetic oriented *data-paths*.

This paper is organized as follows. Our fault tolerance method based on data types modifications is proposed in Section 2. The proposed method is demonstrated on the robot controller described in Section 3. The results of our experiments are summarized in Section 4. Section 5 concludes the paper and presents future plans.

2. HLS-based Fault Tolerance Methodology

Our approach is to apply modifications to the specification as the input of HLS. The modifications should produce the resulting RTL description fault-tolerant. Our method is based on the modification of the *C++* language. There are three types of locations the modification can be done at the level of *C++* language: 1) data types (*storage elements*); 2) arithmetic and logic operations and 3) flow control statements.

In this research we focus on the modification of *storage elements* and *operations* of the input description. We developed a new class of *Data Types* (DTs), which we call *Redundant Data Types* (RDTs), that are able to incorporate redundancy for all the operations and storages associated with the corresponding variables. The concept of RDTs is very similar to that of the Algorithmic C Datatypes [11], which are widely used in HLS as a principle to specify a bit width of a particular data type. In this case, this concept is used to specify a redundancy mechanism. This way we were able to achieve redundancy on certain parts of the input digital system description only. An RDT is associated with the previously used DT which we call the *original* DT in relation to the particular RDT. We show this concept on the *Triple Modular Redundancy* (TMR) principle, although it is not limited to TMR only. If a user intends to add a redundancy to a particular part of the system, simply replacing the previously used DT by the RDT of a desired redundancy in this part of the system is enough. An example of the usage of RDTs is shown in Figure 1.

In the following text, TMR is used as an example of the construction principles of the RDTs. Each instance of the TMR RDT contains three nested instances of the *original* DT. If necessary, support methods to extend the behavior of the operators can be added. In the case of TMR, one method implementing the voter is included. In the *C++* language, 1) unary, 2) binary and 3) ternary operators can be distinguished from the

	Original code	Modified code	Preprocessed result (semantically)
1	`int a;`	`triple<int> a;`	`int a_x, a_y, a_z;`
2	`int b;`	`triple<int> b;`	`int b_x, b_y, b_z;`
3	`int c;`	`triple<int> c;`	`int c_x, c_y, c_z;`
4	`b = 7;`	`b = 7;`	`b_x = 7; b_y = 7; b_z = 7;`
5	`c = 8;`	`c = 8;`	`c_x = 8; c_y = 8; c_z = 8;`

6	`a = b + c;`	`a = b + c;`	`a_x = b_x + c_x;`
			`a_y = b_y + c_y;`
			`a_z = b_z + c_z;`
			`vote(&a_x, &a_y, &a_z);`

7	`/* a = 15 */`	`/* a = 15 */`	`/* a_x, a_y and a_z = 15 */`

Figure 1. An example of a C++ program code before/after its modification and after the code is preprocessed by a C++ preprocessor.

arity point of view. For each unary operator, a new operator is constructed that is composed of three operations, each for one of the nested instances. After that, the voter method is called to choose the majority result, which is then written back to each nested instance of the original DT. For binary operators there are three cases to be distinguished when considering operations with RDTs. These 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). These three cases are schematically illustrated in Figure 2. For the ternary (*conditional*) operator, it is enough to provide a way to decide which value should be considered in place of the (*conditional*) operator. Therefore, the solution is to add an ability to cast the RDT to the Boolean DT.

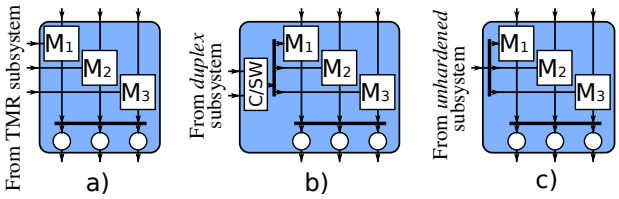

Figure 2. Three types of cases that can be distinguished when considering binary operations, intra-DT operation between two TMR subsystems (a), inter-DT operation between system with TMR and duplex hardening (b) and original-DT operation between TMR and unhardened subsystems (c).

The advantage of our approach includes the ease of its use with any HLS tool and its ability to ensure

fault tolerance for a specific part of the system that corresponds to a particular variable and its associated operations on the description level. Another benefit includes an automatic ability to interface fault-tolerant parts with the unmodified remainder of the system.

The method is intended to be a part of a larger system that would make the modifications automatically with the impacts of these changes in mind. The methodology will have to be aware of the importance of each component to assign the proper fault tolerance methodology.As the input description is in the form of an executable code, a possible option could be to involve a profiler tool, which can be used to determine the frequency of function calls. This might be a good guide to find out the functions with variables in order to apply fault tolerance to. Figure 3 shows the proposed flow.

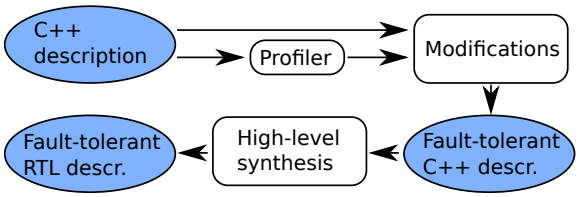

Figure 3. The new approach to FT design.

3. Case Study: Robot Controller

To demonstrate and evaluate our approach, an experimental electro-mechanical system has been developed which is composed of the robot which searches a path through a maze and its electronic controller. The robot controller unit was developed according to the HLS methodology flow using the *C++* language. The HLS tool we used in our experiments is the *Catapult C University Version* [6]. The unit is based on the left hand algorithm, that is, in case of a crossroad the robot always follows the wall of the maze on its left side.

In our previous papers (e.g. [13]) the evaluation platform for checking the impact of faults was presented. Our evaluation platform uses *functional verification* [12] as a tool for monitoring the impacts of faults injected into an electronic controller implemented into FPGA. In case of the fault injection, the verified circuit must operate in FPGA, so we do not use classical simulation-based functional verification, but modified FPGA-based functional verification.

Nowadays, many electronic systems are a part of electro-mechanical systems, where a mechanical part is controlled by its electronic controller. The trend is to move more functionality to electronic controllers because it results in lower costs on device operation. As an example, the results published in [3] and [1] can serve, where moving more functionality to electronic controllers results in a lower weight of the aircraft saving costs on aircraft operation. Based on these facts, our 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 electronic controller, but also on the controlled mechanical parts.

The main components which our evaluation platform is composed of, are shown in Figure 4. The two main parts are a computer and an FPGA development board. We use the ML506 board with the *Virtex 5* FPGA, which allows us to implement a verified electronic controller in FPGA and inject faults directly into the FPGA. The fault injector is one of the components which is running on the computer. Our fault injector [15] is based on the partial reconfiguration and uses JTAG interface for communication with the configuration memory. The platform is designed to evaluate the impact of faults on the electro-mechanical application, so the simulation of the mechanical part is important and also runs on the computer. We use the *Player/Stage* [5] simulation environment to simulate the robot and its environment. The electronic controller implemented into FPGA is connected with the simulation of mechanical part through Ethernet interface. The software part of the verification environment also runs on the computer and performs the evaluation of impacts of injected faults both on the electronic and mechanical parts.

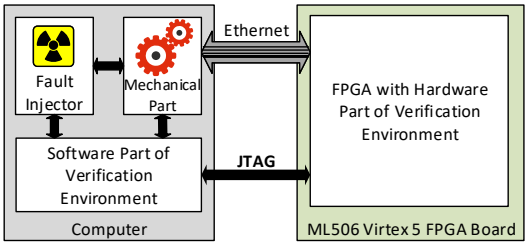

Figure 4. The evaluation platform architecture.

4. Experiments and Results

The previously mentioned robot controller implementation was used as an experimental electronic system in our experiments. In the first stage of the experiments, we evaluated the impact of some of the main optimization and acceleration techniques of the HLS methods to the resulting system's susceptibility

to SEUs. The other parameters monitored were the number of slices, slice registers and slice *Lookup tables* (LUTs) occupied.

Parts of Table 1 labeled as *noft* summarize the resource requirements for each of the four robot controller units synthesized with a different parameters set. The first and the second set of parameters, denoted as *noopt-area* and *noopt-latency*, include area and latency optimizations with no additional requirements added. As can be seen in Figure 5, the results are almost equal, which may be caused by a relatively small design size. The third one, *pipeline1-area*, includes the main loop pipelined with II set to 1 and the overall goal set to the area. The fourth one, *unroll2-area*, contains the main loop partially unrolled with the level of parallel computation set to 2. As can be seen, the unrolled loop requires more resources as the pipelined one, but it is slightly faster.

Table 1. Resources consumed for each version of the HLS synthesized robot controllers.

Version		Occupied slices [–]	Slice reg. [–]	Slice LUTs [–]	Max. frequency [MHz]	LUT bits [–]
noopt-latency	*noft*	170	346	381	74.85	19392
noopt-area	*noft*	170	346	381	74.85	19392
	triple	378	638	851	82.01	48704
	TMR	546	1038	1143	74.85	58176
pipeline1-area	*noft*	196	152	405	58.82	21952
	triple	411	512	1101	65.81	67264
	TMR	540	456	1215	58.82	65856
unroll2-area	*noft*	399	656	854	59.70	48704
	triple	1341	1791	3738	50.48	224256
	TMR	1224	1968	2562	59.70	146112

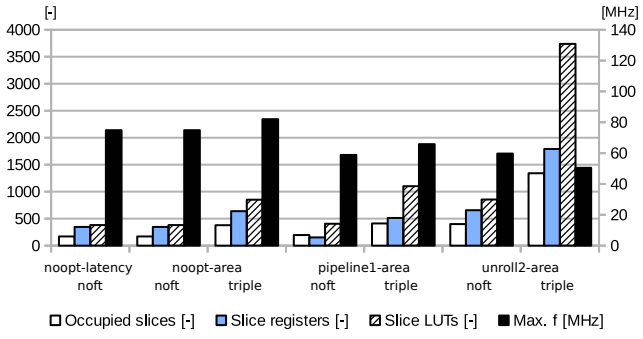

Figure 5. Comparison of resources consumed for each version of the HLS synthesized robot controllers.

The first experiments with fault injection were targeted to evaluate the resilience against faults of proposed versions of the robot controller. In these experiments only three versions (*noopt-area*, *pipeline1-area* and *unroll2-area*) were taken into account, because the detailed analysis showed that *noopt-area* and *noopt-latency* led to a very similar VHDL description.

The three resulting robot controller units were examined for their susceptibility to SEUs. Exactly 1000 verification runs were performed with each version of the three robot controller units. Before the experiments started, a list of configuration bits that were used as a content of LUTs was generated for each version of the robot controller bitstream. These bits were then used in the processes of SEU injections. The scenario of each verification 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 were the same for all the verification 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 that was utilized as an LUT content bit, the bit was selected *uniformly at random* from all bits utilized as contents of LUTs, the bit remained in a faulty state during the whole verification run,
4. the robot was started and the ability of the robot to reach the target position was monitored and eventually the reason of its failure was observed.

Results of these experiments are summarized in Table 2. The first row shows the number of verification runs without any impact on the electronic controller, while the second row indicates the number of faults that cause discrepancy on the output of the electronic controller. The third row enumerates the reliability improvement of the units with triplication applied. The reliability improvement was calculated using Equation 1 for each pair of units with corresponding HLS settings set s.

$$reliab\_improv_s = \frac{fails\_noft_s - fails\_triple_s}{fails\_noft_s} * 100 \quad (1)$$

Table 2 also shows that electronic failure sometimes led to “Goal not reached” or “Goal reached”. It should be noted that sometimes the robot reached the goal position although its electronic system failed. The table shows that the *noopt-area* version of robot controller is

the most prone to injected faults. The number of faults which led to electronic failure is 51 which is 5.1% of injected faults.

Table 2. Impact of faults on various versions of robot controller without and with fault tolerance method applied.

Monitored impact	*noopt-area*		*pipeline1-area*		*unroll2-area*	
	noft	*triple*	*noft*	*triple*	*noft*	*triple*
Electronic OK [−]	949	982	967	996	979	995
Electronic failed [−]	51	18	33	4	21	5
Reliability improvement [%]	–	64.7%	–	87.9%	–	76.2%
Goal not reached [−]	50	16	32	4	19	4
Collision with wall [−]	4	2	5	1	4	1
Goal reach. alth. el. fail. [−]	1	2	1	0	2	1

The next stage of our experiments was targeted to applying the proposed methodology to each variable of the robot controller algorithm specification (in tables and charts labeled as *triple* version). We evaluated 1) a resource consumption, 2) a susceptibility of modified robot controllers to the faults and the comparison of the results with the versions without any fault tolerance modifications.

The comparison of the resource consumption of the *noft* and *triple* versions (with the proposed method applied) is available in Table 1. The synthesis tool used was the *Xilinx Integrated Synthesis Environment* (ISE) [16]. It is evident that the robot controller hardened by the proposed methodology consumes more resources. In comparison with the complete triplication of the robot controller (rows of Table 1 labeled as *TMR*) that were synthesized using three copies of the corresponding *noft* version, less resources are consumed, although in some cases such as in the case of the *unroll2-area* the resource utilization increased. However, it is important to keep in mind the complete triplication comprises three copies of the *control-path*, while the proposed method is *data-path* only.

From the reliability point of view the same experiments were prepared. In this stage, we also injected one single fault during one verification run. The scenario of each verification run, including the maze map, was identical to that of the experiments mentioned in the first phase. Based on 1000 verification runs we can say that the proposed methodology leads to a lower sensitivity on injected faults as is shown in Figure 6. The most significant contribution can be seen in the case of the *pipeline1-area* where our methodology led to an improvement of 87.9%. In case of *unroll2-area* the improvement is 76.2%. The smallest improvement of 64.7% was achieved in case of *noopt-area*. Based on these experiments we can conclude that the proposed methodology leads to improvement of the fault tolerance against SEUs with the best efficiency for pipelined designs. However, this method needs to be combined with another approach considering the *control-path*, which would provide even better resilience to faults.

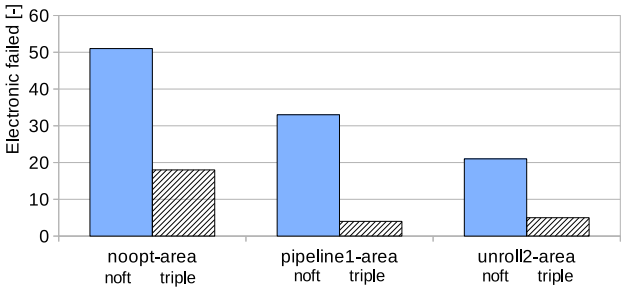

Figure 6. Number of faults that led to failure of the electro-mechanical system for each of the versions.

5. Conclusions and Future Research

In this paper we introduced a newly emerging approach to easily achieve a certain level of fault tolerance with the usage of HLS. In our experiments, robot controller system is modeled in the C++ language. After that, the model is modified using our approach and synthesized using HLS with selected settings sets. Resulting systems, which have all data-path components and data elements triplicated, are evaluated using single fault injections. The experiments monitor the impacts of injected faults, both on robot controllers without proposed fault tolerance methodology applied, and also on robot controllers hardened against faults. The experiments show that single faults injected into utilized LUTs of the hardened robot controller had smaller impact on the robot controller and its behavior. In our experiments, resource consumption was also analyzed. The proposed methodology leads to a greater consumption of resources, the comparison of each of the versions with the TMR triplication the corresponding robot controller was provided as well. The objective of our research is to improve this principle to make it generally usable and show its usability on other applications or benchmarks.

The next step in our research would be to involve the evaluation of the impact of these modifications on different parts of the system. The main idea is that each part of the system deserves a different level of reliability, based on its function and thus a different level of fault tolerance. Therefore, it is not necessary to apply the same level of fault tolerance to every part of the

system and it is very likely that another configuration of fault tolerance that has at least the same level of dependability with less resources consumed exists.

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] J. Bennett, A. Jack, B. Mecrow, D. Atkinson, C. Sewell, and G. Mason. Fault-tolerant Control Architecture for an Electrical Actuator. In *Power Electronics Specialists Conference, 2004. PESC 04. 2004 IEEE 35th Annual*, volume 6, pages 4371–4377, June 2004.

[2] 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.

[3] S. Cutts. A Collaborative Approach to the More Electric Aircraft. In *Power Electronics, Machines and Drives, 2002. International Conference on (Conf. Publ. No. 487)*, pages 223–228, June 2002.

[4] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[5] 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.

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

[7] 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.

[8] 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.

[9] F. L. Kastensmidt, G. Neuberger, L. Carro, and R. Reis. Designing and Testing Fault-Tolerant Techniques for SRAM-based FPGAs. In *Proceedings of the 1st conference on Computing frontiers*, pages 419–432. ACM, 2004.

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

[11] Mentor Graphics. AC Datatypes v3.7. `<https://www.mentor.com/hls-lp/downloads/ac-datatypes>`, June 2016. Accessed: 2016-12-14.

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

[13] J. Podivinsky, O. Cekan, J. Lojda, and Z. Kotasek. Verification of Robot Controller for Evaluating Impacts of Faults in Electro-mechanical Systems. In *Digital System Design (DSD), 2016 19th Euromicro Conference on*, pages 487–494. IEEE, 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] Xilinx. ISE Design Suite. `<https://www.xilinx.com/products/design-tools/ise-design-suite.html>`, 2017. Accessed: 2017-07-07.