Design of a Fault-Tolerant Coarse-Grained Reconfigurable Architecture: A Case Study

Syed M. A. H. Jafri, Stanisław J. Piestrak, Olivier Sentieys, & Sebastien Pillement
IRISA/INRIA—University of Rennes 1, 22300 Lannion, France
Email: jafri@kth.se, piestrak@univ-metz.fr, olivier.sentieys@irisa.fr, sebastien.pillement@irisa.fr

(S. M. A. H. Jafri was also with Royal Institute of Technology (KTH), Stockholm, Sweden
S. J. Piestrak is on leave from LICM, Univ. of Metz, France)

***Abstract*— This paper considers the possibility of implementing low-cost hardware techniques which would allow to tolerate temporary faults in the datapaths of coarse-grained reconfigurable architectures (CGRAs). Our goal was to use less hardware overhead than commonly used duplication or triplication methods. The proposed technique relies on concurrent error detection by using residue code modulo 3 and re-execution of the last operation, once an error is detected. We have chosen the DART architecture as a vehicle to study the efficiency of this approach to protect its datapaths. Simulation results have confirmed hardware savings of the proposed approach over duplication.**

I. Introduction

Recently, the increasing speed and performance requirements of multimedia processing and mobile telecommunication applications, coupled with the demands for flexibility and low non-recurring engineering costs, have made reconfigurable hardware a very popular implementation technology. Today's reconfigurable architectures enable partial and dynamic runtime self-reconfiguration. This feature allows the substitution of parts of a hardware design implemented on this reconfigurable hardware, and therefore, a single device can be adapted to implement various functionalities actually demanded, by simply uploading a new configuration.

Reconfigurable architectures can be classified depending on their *granularity*, e.g. the number of bits, which can be explicitly manipulated by the programmer. The most fine-grained architectures, whose the most widely used example are Field Programmable Gate Arrays (FPGAs), allow a bit level manipulation of data. Coarse-grained reconfigurable architectures (CGRAs) provide operator level configurable functional blocks, word level datapaths, and powerful and very area-efficient datapath routing switches. Compared to fine-grained architectures, CGRAs enjoy massive reduction of configuration memory and configuration time, as well as considerable reduction in routing and placement allocation. All this also results in a potential reduction of the total energy consumed per computation, though at the cost of a loss in flexibility compared to bit-level operations. The most recent surveys covering various design and implementation aspects of reconfigurable architectures can be found in [1]–[5]. Unfortunately, relatively few works deal with the CGRAs like Morphosys [6], Raw [7], PACT XPP [8], DART [9], [10], SmartCell [11], and a few others [5], [12], [13], [14] which are of our interest here.

With the progress in the processing technology, the size of the semiconductor devices is shrinking rapidly, which offers many advantages like low power consumption, low manufacturing costs, and ability to make hand held devices. However, shrinking feature sizes and decreasing node capacitance, the increase of the operating frequency, and the power supply reduction affect the noise margins and susceptibility to transient faults. In particular, the soft error rate induced by cosmic neutron interactions in commercial electronic devices at ground level has become an issue for a long time [15], [16]. A particle can hit directly a memory element and flip its logic state (which is called a *single event upset* (SEU)) or hit a combinational logic and trigger temporary perturbation resulting from the collection of radiation-induced charge, called *single event transients* (SETs). As the operating voltage of the devices and the node capacitancies decrease, the probability of a small transient current being interpreted as a signal also increases. SETs, if propagated and latched into a memory element as incorrect data, will also lead to a SEU. These faults are commonly called *soft errors* because the circuit/device itself is not permanently damaged—if new data are written to the bit, the device will store it correctly. To note also that electronic systems implemented with nanotechnologies are expected to experience even higher fault rates [17], [18].

The use of reconfigurable hardware in critical applications like aircrafts, space missions and transaction systems e.t.c. is increasing rapidly. Soft errors caused by radiation may result in fatal silent data corruption and unreproducible system crashes. Because it is virtually impossible to build devices which are free from faults, it is essential to embed some sort of fault-tolerance in such devices, which will enable them to work correctly even in the presence of faults. Since the past decade, a lot of research has been done to develop fault-tolerant reconfigurable systems on various granularity levels, although most of them have dealt with the lowest level such as offered by FPGAs [19]–[21]. In general, the capabilities of such systems should include on-line error detection during system operation, very fast fault location, quick recovery from temporary failures, and fast permanent-fault repair through reconfiguration.

Only relatively few works can be found on fault-tolerance in CGRAs [22]–[24]. In [22], the authors propose fault-tolerance enhancements of the Raw architecture from [7]. They use a combination of software techniques for fault detection and tolerance, which include selective replication, selective duplication, checkpoint/restart, breakpoints, and temporal triple modular redundancy (TMR) applied to input and output parts. In [23], the fault-tolerant CGRA built using a specially designed autonomous repair cell is proposed. However, the authors concentrate on tolerating configuration upsets only and do not consider transient faults of the cell proposed, assuming that some “conventional techniques can be applied to those parts”. The fault-tolerant CGRA schemes proposed in [24] are based on duplication and triplication to offer flexible reliability levels. Finally, in [25] a new reconfigurable cell array, specifically designed for fault-tolerance, was proposed. This work concentrates on automatic routing mechanisms allowing for reconfiguration of the cell array in case of faults of basic cells, without the aid of external software or hardware. This is the only CGRA wherein the (permanent) faults of the elementary cell are detected using less costly alternatives like error-detecting codes (parity and Berger codes). Unfortunately, only a few details on self-checking circuitry actually used are revealed, which do not allow for any quantitative complexity evaluation. In summary, most published fault-tolerant CGRAs require a massive amount of spare cells, because they are based on duplication or triplication of resources. The lack of experimental results on using low-cost techniques for designing CGRAs fault-tolerant w.r.t. soft errors has motivated us to study a sample CGRA with reliability enhancements. Our choice of the DART CGRA from [9], [10] was motivated by the advanced reconfigurability features of its datapath units and access to its high-level implementation code and supporting CAD. In this paper, we describe the implementation of fault-tolerant features that address concurrent error detection of temporary faults and recovery through rollback of the last operation.

This article is organized as follows. In Section II, more details on soft errors and a survey of fault-tolerant techniques used in existing CGRAs are presented. In Section III, some basic concepts of the DART architecture and details of its datapath units are presented. In Sections IV and V, we propose a modification of the DART architecture with concurrent error detection (CED) based on the combination of residue codes modulo 3 and duplication and evaluate the redundancy imposed by the proposed methods. In Section VI, we summarize our contributions and suggest directions for future research.

II. Preliminaries

A. *Reconfigurable Architectures in the Presence of Faults*

Soft errors, if undetected, may result in data corruption or system failure. They may affect reconfigurable systems in two essentially different ways: (i) they may directly corrupt computation results or (ii) they may induce changes to configuration memory, that can cause changes in the functionality and performance of the device [19]–[21]. Because in either case the cause of the failure is actually transient, some time redundancy approach could be adapted, provided that the system is equipped with some means to detect errors. Computation errors, once detected (e.g. by using error-detecting codes), can be corrected by re-execution of the last operation. In case of configuration errors, scrubbing can be used to restore the original functionality. In case of permanent faults, after the faulty elements are located—either computing or routing resources—they must be excluded and replaced by previously unused fault-free resources. However, handling permanent faults is beyond the scope of this paper.

B. *Fault-Tolerant Techniques for Reconfigurable Architectures*

Most of reconfigurable architectures are built using a number of identical blocks. Therefore, it is not surprising that some sort of hardware redundancy, that relies on replication of a block to be protected from faults, has often been preferred choice. The most widely used hardware redundancy methods for providing fault-tolerance are: (i) duplication with comparison (DWC) for detecting faults and (ii) triple modular redundancy (TMR) with voters for masking faults. In DWC, the original module is replicated twice and the results produced by the original and the replicated modules are compared to detect faults. Once an error is detected, a few attempts are made to repeat the last operation hoping that the error was due to temporary fault and, in case of failure, a permanent fault is declared. In TMR, the original module is replicated thrice and a majority 2-out-of-3 voter decides the correct output. In summary, DWC allows to tolerate only temporary faults (provided that DWC is supported by re-execution) whereas TMR allows to mask directly both temporary and permanent faults. TMR has been the basic technique used in FPGAs, because hardware parts protected by and voters can be implemented using lookup tables in any part of the device and as many as necessary [19]–[21]. DWC and TMR are conceptually relatively simple and easy to implement. Unfortunately, they are also very costly, because they involve respectively over 100% and 200% hardware overhead, which could be prohibitive e.g. in low-power applications. Therefore some other less costly fault-tolerant techniques, applicable for reconfigurable architectures, seem also worth of consideration.

A viable alternative to hardware redundancy is to use some other means for CED, e.g. by using error detecting codes (like parity codes, residue codes, Berger codes, etc.) and implementing circuits as self-checking [26], supported by some form of time redundancy. As far as we know, only cyclic redundancy check (CRC) codes have been used explicitly in reconfigurable architectures—to detect errors in configuration data.

To note also that most of research on fault-tolerant CGRAs has concentrated on tolerating faults in interconnections and reconfiguration strategies in case of permanent faults. However, relatively little details have been revealed how these faults are detected—the necessary step preceding any above mentioned action. Henceforth, we shall concentrate on temporary faults in the datapaths of CGRAs only.

III. DART Architecture

DART is a dynamically reconfigurable coarse-grained architecture developed at IRISA [9], [10]. Here we shall use DART as a vehicle to consider some alternative methods suitable to provide CED in CGRAs, which would possibly involve less hardware overhead than DWC and obviously TMR.

The overall architecture of DART is shown in Fig. 1. Broadly, the architecture of DART can be divided into four different parts: (i) configuration unit, (ii) data memory, (iii) reconfigurable data paths, and (iv) interconnection network. Because we are looking at the DART structure specifically from the point of incorporating in it fault-tolerance, we will present a detailed description of the reconfigurable data path unit followed by a brief discussion of the other parts.

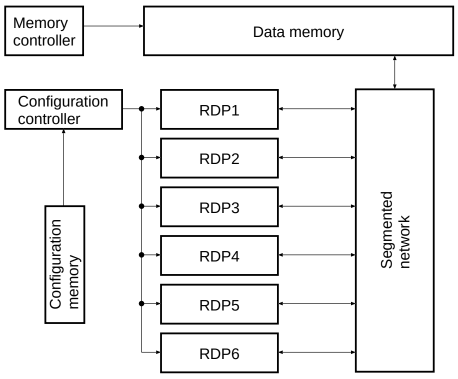

Fig. 1. Architecture of DART [10].

Reconfigurable Data Paths: The DART architecture contains six reconfigurable data path (RDP) units, in which the main processing of data is done. As shown in Fig. 2, each RDP unit contains four functional units (FU), four address generators each associated to a data memory, two registers, and a multi-bus network. Two different types of FUs are present in an RDP: (i) multiplication/addition unit, and (ii) arithmetic and logic unit (ALU). Each FU has the possibility to perform subword parallelism (SWP) on the input data.

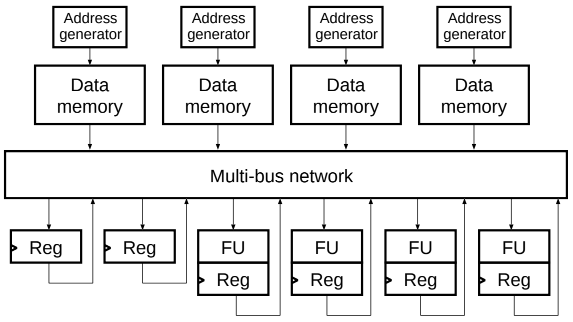

Fig. 2. Architecture of a reconfigurable data path (RDP) [10].

Multiplication/Addition Unit: The multiplication/addition (MA) unit designed to reduce energy consumption, shown in Fig. 3, contains one 16-bit and two 8-bit multipliers and adders. The input to the MA are two 16-bit words and the output is one 32-bit word. After receiving two 16-bit input words, the SWP signal decides whether the operation is to be performed on 16- or 8-bit data. SWP=0 means that the operations are to be performed on 16-bit data and the inputs are forwarded to 16-bit multiplier/adder. SWP=1 means that the operations are to be performed on 8-bit data and the received inputs are actually four different operands. Then, the 8 MSBs of the received input words are sent to one of two 8-bit multiplier/adder units whereas the 8 LSBs of the received inputs are sent to the other multiplier/adder unit. The control signal M/A decides whether addition or multiplication is to be performed. If M/A=0, multiplication is performed, and if M/A=1, addition is performed. Table I shows how the functionality of the multiplication unit is controlled by these two signals.

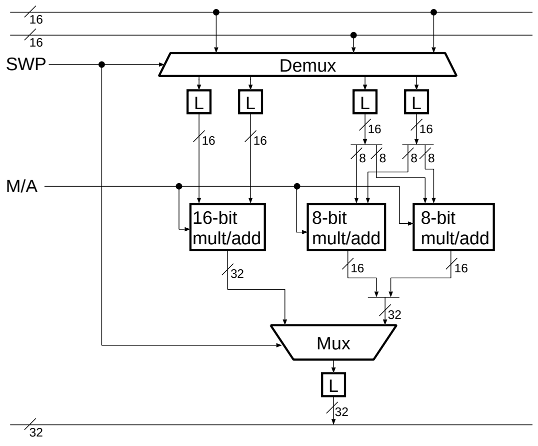

Fig. 3. Multiplication/addition unit (MA) of DART [10].

TABLE I
Operations performed by the multiplication/addition (MA) unit

SWP M/A	Functionality
0 0	multiplication of two 16-bit operands
0 1	addition of two 16-bit operands
1 0	separate multipl. of 8 LSBs and 8 MSBs of two op.
1 1	separate addition of 8 LSBs and 8 MSBs of two op.

Arithmetic and Logic Unit (ALU): The ALU, which is actually composed of a pair of separate arithmetic and logic units, is shown in Fig. 4. The arithmetic unit receives two 32-bit operands and the result is a 32-bit operand. For accumulation operation, it can also operate on 40 bits. It is controlled by two signals: (CD_ALU) and (CD_SIMD_ALU). Tables II and III show how the functionality of the arithmetic unit is controlled by these two signals, respectively. As for the logic unit, it receives and outputs 32-bit data and (depending on the 2-bit control signal CD_OP, specified in parentheses) executes four operations: AND (00), OR (01), XOR (10), and NOT (11).

The data can reach the reconfigurable data paths by two methods: (i) from an I/O device using FIFO and (ii) from the data memory, as shown in Fig. 1. Each word of data memory is

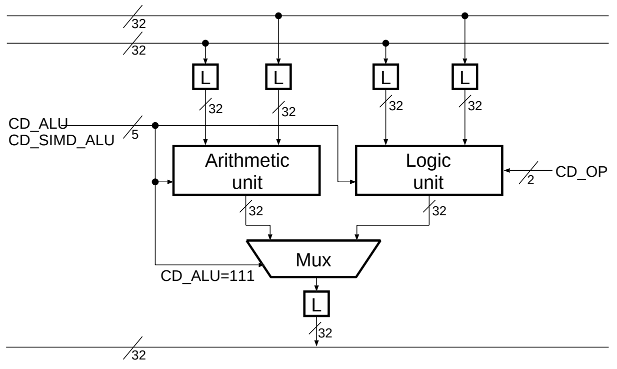

Fig. 4. Arithmetic and logic unit of DART [10].

TABLE II
OPERATIONS OF ARITHMETIC UNIT CONTROLLED BY CD_ALU/CD

CD_ALU	Functionality
000	Addition
001	Addition with saturation
010	Subtraction
011	Subtraction with saturation
100	Minimum operation
101	Maximum operation
110	Absolute
111	Logic operations

32-bit wide. If the data are to be provided to the multiplication unit which requires 16-bit operands, its 16 LSBs are truncated. The DART architecture contains a hierarchical network for communication. The functional units within the reconfigurable data paths communicate by using a multi-bus network, while the communication between different reconfigurable data paths and data memories is done by using a segmented network. Some other details of these networks and other parts of the DART architecture, which seem irrelevant for this research, can be found in [9], [10].

IV. FAULT-TOLERANCE TECHNIQUES IN CGRA DART

The main design goal of DART was to propose a power efficient device with high processing capabilities without taking into account fault-tolerance features, however. Having available full details of its structure (including VHDL RTL codes), it provided us with an excellent platform to verify the efficiency of the redundancy imposed by the CED approach proposed here (which fully takes into account the particularities of DART) and to understand the complexity involved in implementing these methods on CGRAs. To protect DART,

TABLE III
OPERATIONS OF ARITHMETIC UNIT CONTROLLED BY CD_SIMD_ALU

CD_SIMD_ALU	Functionality
00	40-bit operation
01	32-bit operation
10	16-bit operation by adding 16-LSBs of operands and 16-MSBs of the operands
11	16-bit operation by adding 16-LSBs of each operand with its MSBs

our main goal was to achieve CED at the lowest possible cost, which prompted us to keep the uniform CED method for the entire structure, so that no conversion due to using different coding schemes would be needed.

In the next section, we will present some results indicating the area and time redundancy, as well as evaluation of extra power consumption required for implementing DART with CED.

1) Residue Modulo 3 Codes and Related Circuitry: As already mentioned, DART contains two different types of functional units: (i) multiplication/addition unit and (ii) ALU. Simple DWC or TMR for the functional units would not only have been too expensive in terms of area and time but also would have left the data memory unprotected. To protect the memory and the computation unit against temporary faults at low cost and, using the same scheme, we had to use some systematic error detection codes like parity checking or arithmetic residue modulo (mod) A code, where $A \geq 3$ is odd. Although simple parity requires just one additional bit, unfortunately, it is inefficient for arithmetic circuitry (in particular for the multiplication unit), as it requires relatively large area overhead to protect it against errors resulting from all single bit faults. Taking into account the drawbacks of using the parity code, we have opted to use the least costly residue mod 3 code for protecting not only arithmetic units (multiplication and addition units) but memory as well, to avoid extra check bit generators, checkers, and converters. On the other hand, we had to use DWC with comparison for the logic unit, because no other scheme capable of detecting all errors due to single stuck-at faults exists.

The residue mod 3 codes enjoy the following advantages which have made them the codes of our choice:

- they provide protection against not only all single bit arithmetic errors (the erroneous computation result differs from the correct one by $\pm 2^i$), but also all multiple errors which do not cumulate to a multiple of the check base $A = 3$;
- they require only two additional check bits throughout the data path;
- they are separable codes, i.e., the same operation (+, -, *) is executed separately and in parallel on input operands and their check parts mod 3; and
- the residue of an entire word equals the sum of the residues of all parts of the word, arbitrarily partitioned (which is crucial while taking into account the peculiarities of the DART architecture).

Below, we will show all basic arithmetic hardware blocks needed to include residue mod 3 checking in the DART architecture for CED.

The check part $C = (c_1, c_0)$ of the mod 3 residue code is generated using a residue mod 3 generator. Assuming that $X = (x_{n-1}, \ldots, x_1, x_0)$ is an integer (an operand to be protected against errors) , the residue mod 3 generator calculates $C = (c_1, c_0)$ which is the remainder of the division of X by 3. The most efficient residue generators mod 3 can be designed according to the methods from [27] and [28], which both offer highly regular structures, using only one basic block. In [27], an n-input residue generator mod 3 is built using $n-2$ full adders (FAs) with a total of $\lceil n/2 \rceil + n - 3$ signals inverted. In [28], an n-input residue generator mod 3 is built using $\lceil n/2 \rceil - 1$ 4-input modules.

Recall that three word sizes used in DART imply the need to generate suitable check parts by residue mod 3 generators with 8, 16, and 32 inputs. Because the proposed structure is scalable, we designed an 8-input residue mod 3 generator and combined respectively 2 and 4 8-input generators to make 16- and 32-input generators. To convey a reader with some details, Fig. 5 shows the modular structure of the 16-input residue mod 3 generator built of 2 8-input circuits (each composed of 6 FAs in 4 stages), followed by the 4-input residue mod 3 generator (which is also nothing else but the mod 3 adder).

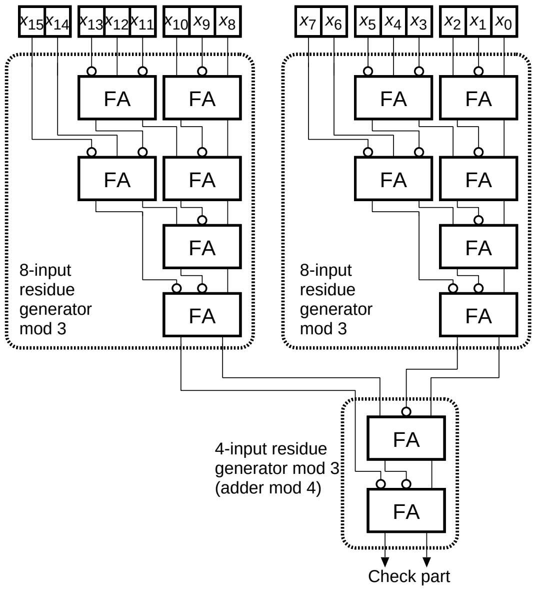
x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
FA
8-input residue generator mod 3
4-input residue generator mod 3 (adder mod 4)
Check part

Fig. 5. 16-bit residue mod 3 generator.

The structure of the residue mod 3 adder was already shown in Fig. 5. To perform subtraction, the bits of the operand to be subtracted are inverted and added mod 3 using the mod 3 adder.

2) Design of Self-Checking Functional Units: For protecting the computations against undetected errors resulting from temporary faults that could occur in functional units, we have modified the architecture of all functional units, making them self-checking. The self-checking functional units combined with recomputation triggered by an error detection signal provide DART with fault-tolerance for temporary faults.

In this subsection, we will present the modifications made in the multiplication/addition unit and the arithmetic and logic unit to make them self-checking.

Multiplication/Addition Unit: The multiplication/addition (M/A) unit is capable of performing multiplication and addition operations on both 16-bit and 8-bit operands. The modified, self-checking multiplier is shown in Fig. 6. We will explain the diagram from the top to the bottom considering how the data flows in it. Initially two 18-bit words, each containing 16 data bits and 2 check bits arrive at the input of multiplication unit. At this stage, the data part and the check part are separated.

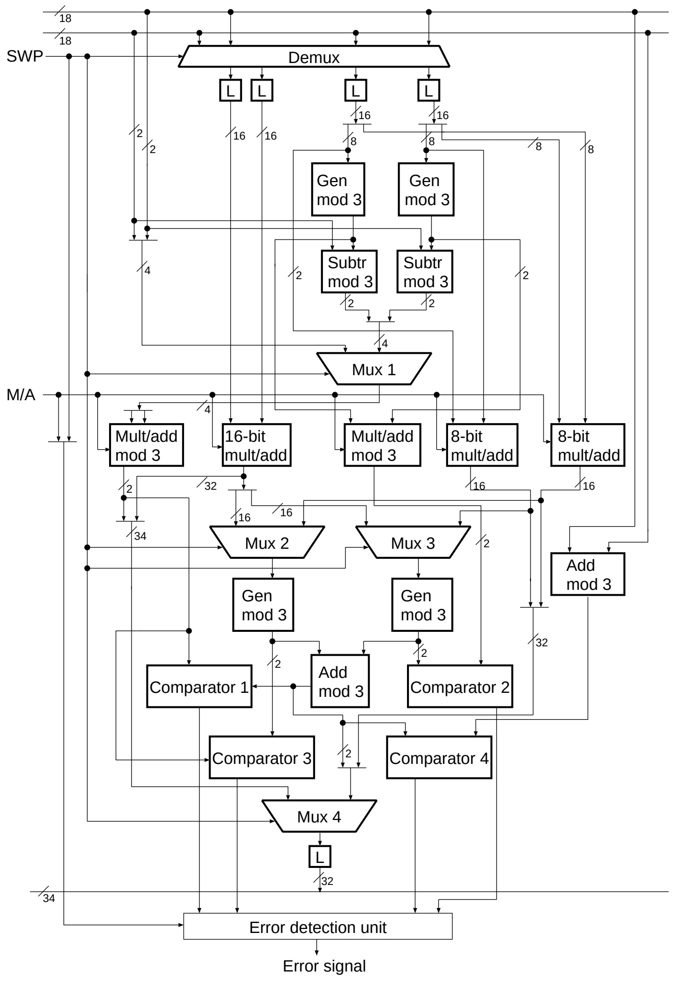

Fig. 6. Self-checking M/A unit.

If SWP=0, indicating that the operation is executed on 16-bit operands, the data bits are sent to the 16-bit mult/add unit. At the same time, the check bits are sent to the mult/add mod 3 unit, using Mux 1. At this stage, the signal M/A indicates whether 16-bit addition or multiplication will be executed. If the M/A=0, the data bits are multiplied using 16-bit mult/add while the check bits are multiplied using mult/add mod 3 unit. If M/A=1, the data bits are added in 16-bit mult/add unit while check bits are added using mult/add mod 3. The 32-bit result of the 16-bit multiplier is partitioned into two parts. The 16 MSBs are sent to Mux 2 and the 16 LSBs are sent to Mux 3, which are both controlled by SWP signal. The residues of the MSBs and the LSBs are calculated separately and then added mod 3. The result calculated from mult/add mod 3 is compared with the newly calculated sum in Comparator 1. Any disagreement indicates erroneous data and an error signal is sent to the error detection unit.

On the other hand, SWP=1 means that two 16-bit operands actually represent four 8-bit operands and two separate multiplications/additions are to be performed. The 8 LSBs of one operand are to be multiplied/added with the 8 LSBs of the other operand and that the 8 MSBs of one operand are to be multiplied/added with the 8 MSBs of the other operand. Because in this case the unit handles four 8-bit words, the check bits for each of the four words are needed. The simplest solution could be using four separate residue mod 3 generators, but they are too expensive both in terms of area and time. Therefore, we decided to use two residue mod 3 generators for generating the residues of the MSBs of the operands. On that basis, the residues of the LSBs can be calculated by subtracting the residue of MSBs from the residue of the entire 16-bit operand. Once the residues of all four operands are known, they are sent to two different 8-bit mult/add units, where, depending on the value of M/A, they are either multiplied or added and the results are sent to two generators mod 3 where the new residues are calculated. At the same time, the check bits are sent to two mult/add mod 3 units, where they are either multiplied or added. Finally, they are compared by the Comparators 2 and 3 with the corresponding residues previously generated by generators mod 3. In case of a disagreement, an error signal is sent to the error detection unit.

Arithmetic and Logic Unit: As shown in Fig. 7, two different methods are used for protecting the arithmetic and logic unit: (i) for protecting the arithmetic unit, we have residue mod 3 codes used almost exactly in the same way as in the multiplication/addition unit and (ii) for protecting the logic unit, we have employed duplication with comparison.

To reduce the overhead below duplication, unfortunately we had to reduce the overall functionality of the arithmetic unit. The self-checking version of the arithmetic unit is capable of performing only 32-bit and 16-bit addition and subtraction. In our design, we have allowed the unit to perform other operations as well, but unprotected.

For protecting the logic unit, we have employed duplication with comparison. All the logic functions are performed twice and in case of an error, the error signal is generated.

3) Protecting Data Memory: For protecting the data memory, two additional residue mod 3 check bits are added to each line of memory. The addition of check bits is done before storing the data in data memory, as shown in Fig. 8. If an error is received by the configuration controller, the entire data memory is flushed and reconfiguration is carried out.

V. Complexity Evaluation

We have synthesized the original and the self-checking versions of DART functional units using STMicroelectronics 130 nm technology. The main constraint imposed was that the architecture should work for the clock frequency of 200 MHz for all versions. Table IV shows the results obtained. The self-checking version using residue code mod 3 required very low area overhead of approximately 18.6% which is significantly less compared to its duplicated counterpart DWC. Both self-checking versions require at least 80% time overhead. On

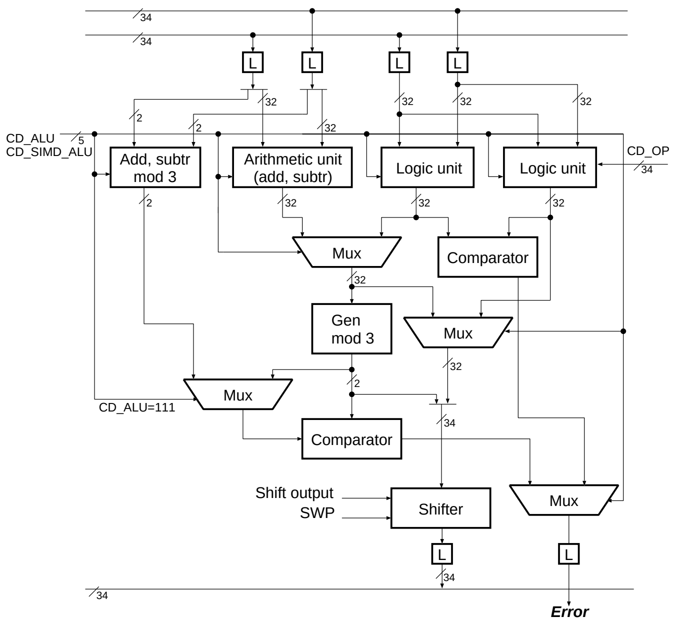

Fig. 7. Self-checking ALU.

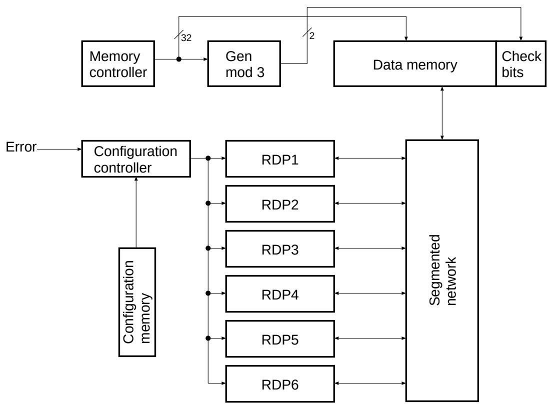

Fig. 8. Self-checking DART unit.

the other hand, the area redundancy of the arithmetic unit using residue code mod 3 was found to be approximately 98%, which is quite large, but still less than for its duplicated counterpart DWC. The reason for such a large overhead is the size of the output 32-bit generator mod 3, comparable to the adder itself (the generator requires 30 FAs). The timing overhead also was quite excessive mainly due to the use of the 32-bit residue mod 3 generator. Therefore, to protect efficiently the arithmetic unit by using residue code mod 3, some attempts must be made to reduce the overhead related to the residue generator mod 3, perhaps by allowing to share it with some other circuitry and using a pipelined architecture. Nevertheless, we can conclude that using residue mod 3 code could result in lesser area redundancy than duplication, in particular when multiplication units are predominant. We have shown that this code is a valuable alternative worth of consideration to ensure fault-tolerance against the SEUs in functional units and the memory at the same time.

TABLE IV
AREA AND DELAY PARAMETERS OF VARIOUS VERSIONS OF DART FUNCTIONAL UNITS

Circuit	Area [μm^2]	Delay [ns]
Multiplication/Addition (M/A) Unit		
Unprotected	58075	2.24
Self-checking, using residue code mod 3	68887	4.04
Self-checking, using DWC	118050	4.19
Arithmetic unit		
Unprotected	16881	1.23
Self-checking, using residue code mod 3	33469	4.89
Self checking, using DWC	41314	2.13

VI. CONCLUSION

In this paper, we have presented fault-tolerance techniques for making coarse-grained reconfigurable architectures (CGRAs) fault-tolerant w.r.t. soft errors caused e.g. by radiation. The reason for choosing CGRAs was that they have a great potential for dominating the reconfigurable hardware market in the future and overtaking their significantly more widely used fine-grained counterparts, FPGAs. Some of their major advantages are lesser reconfiguration time and better suitability for computations involving larger word lengths.

Our next goal was to choose a CGRA which can be used as a model to propose architectural modifications. While choosing the CGRA for our case study, we have looked for the following four characteristics:

1) its architecture should be general enough, so that the architectural modifications proposed for it can be easily migrated to other architectures;
2) it should be very practical, so that some other architectures could follow the same design;
3) its architecture should be simple enough to support the commonly used fault-tolerance methods in FPGAs; and
4) we should have available complete information about its architecture, so that we could evaluate the redundancy imposed by techniques suggested by us.

The first three characteristics were met by SmartCell [11] but, unfortunately, we did not have the complete information to test our proposed techniques on SmartCell.

Finally, we have implemented a few of the proposed methods on the DART architecture for protecting its data memory and functional units. The obtained results suggested that even after embedding out proposed methods, the device was still able to meet the clock frequency requirement of 200 MHz. The area overhead depends on the type of functional unit. The multiplication/addition unit, and the arithmetic and logic units had area overheads of about 18% and 98%, respectively, which are still much better than simple duplication. The area of implementing fault-tolerance techniques in CGRAs is still new and a lot of work is needed to be done in this field. The main contributions of this work were to present techniques, having the potential to make most of the parts of a general CGRA self-checking, investigation of redundancy (area, time) imposed by them, and to test the feasibility of a few of the proposed techniques.

Future research on fault-tolerance in DART will include further hardware overhead reduction, implementation of error recovery procedures, protection of reconfiguration data as well as inclusion of means for dynamic reconfiguration in case of permanent faults.

REFERENCES

[1] T. J. Todman *et al.*, ”Reconfigurable computing: Architectures and design methods,” *IEE Proc.—Computers & Digital Techniques*, vol. 152, no. 2, pp. 193–207, March 2005.

[2] P. Garcia *et al.*, ”An overview of reconfigurable hardware in embedded systems,” *EURASIP J. on Embedded Systems*, vol. 2006, 2006, Article ID 56320, 19 pages. [Online]. Available: http://www.hindawi.com/journals/es/2006/056320.pdf

[3] R. Hartenstein and TU Kaiserslautern, ”Basics of Reconfigurable Computing,” Ch. 20 in: J. Henkel and S. Parameswaran (Eds.), *Designing Embedded Processors: A Low Power Perspective*, Springer, 2007, pp. 451–501.

[4] S. Hauck and A. DeHon (Eds.), *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann Publishers, Amsterdam, 2008.

[5] Zain-ul-Abdin and B. Svensson, ”Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing,” *Microprocessors and Microsystems*, vol. 33, pp. 161–178, March 2009.

[6] H. Singh *et al.*, ”Morphosys: An integrated reconfigurable system for data-parallel computation-intensive applications,” *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[7] M. B. Taylor *et al.*, ”The Raw microprocessor: A computational fabric for software circuits and general purpose programs,” *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.

[8] V. Baumgarte *et al.*, ”PACT XPPA self-reconfigurable data processing architecture,” *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, Sept. 2003.

[9] R. David, D. Chillet, S. Pillement, and O. Sentieys, ”A dynamically reconfigurable architecture dealing with future mobile telecommunications constraints,” in *Proc. 16th Int. Parallel and Distributed Process. Symp. (IPDPS 2002)*, 15–19 April 2002, Fort Lauderdale, FL, USA.

[10] S. Pillement, O. Sentieys, and R. David, ”DART: A functional-level reconfigurable architecture for high energy efficiency,” *EURASIP J. on Embedded Systems*, vol. 2008, 2008. [Online]. Available: http://www.hindawi.com/journals/es/2008/562326.pdf

[11] C. Liang and X. Huang, ”SmartCell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications,” *EURASIP J. on Embedded Systems*, vol. 2008, 2009. [Online]. Available: http://www.hindawi.com/journals/es/2009/518659.pdf

[12] C. Plessl and M. Platzner, ”Zippy: A coarse-grained reconfigurable array with support for hardware virtualization,” in *Proc. 16th Int. Conf. on Application-specific Systems, Architecture and Processors (ASAP 05)*, July 2005, pp. 213–218.

[13] G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis, ”Resource aware mapping on coarse grained reconfigurable arrays,” *Microprocessors and Microsystems*, vol. 33, no. 2, pp. 91–105, March 2009.

[14] Y. Kim, R.N. Mahapatra, I. Park, and K. Choi ”Low power reconfiguration technique for coarse-grained reconfigurable architecture,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 593–603, Apr. 2009.

[15] C. R. Baumann, ”Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Trans. Device and Materials Reliab.*, vol. 5, pp. 305–316, Sept. 2005.

[16] H. Quinn and P. Graham, ”Terrestrial-based radiation upsets: a cautionary tale,” in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 17–20 April 2005, Napa, CA, USA, pp. 193–202. [Online]. Available: http://www.rasr.lanl.gov/RadEffects/publications.php

[17] M.A. Breuer, S.K. Gupta, and T.M. Mak, ”Defect and error tolerance in the presence of massive numbers of defects,” *IEEE Design & Test of Computers*, vol. 21, no. 3, pp. 216–227, May-June 2004.

[18] J. Han *et al.*, ”Toward hardware-redundant, fault-tolerant logic for nanoelectronics,” *IEEE Design & Test of Computers*, vol. 22, no. 4, pp. 328–339, July-Aug. 2005.

[19] F. L. Kastensmidt, L. Carro, and R. Reis, *Fault-Tolerance Techniques for SRAM-Based FPGAs*, Frontiers in Electronic Design, vol. 32, Springer, Dordrecht, The Netherlands, 2006.

[20] R. Velazco, P. Fouillat, and R. Reis (Eds), *Radiation Effects on Embedded Systems*, Springer, Dordrecht, The Netherlands, 2007.

[21] M. Brogley, ”FPGA reliability and the sunspot cycle,” White Paper, Actel, Sept. 2009.

[22] D. K. K. in Singh, A. Agbaria, and M. French, ”Tolerating SEU faults in the RAW architecture,” in *Proc. 3rd Int. Workshop on Dependable Embedded Systems*, Oct. 2006. [Online]. Available: http://www.east.isi.edu/˜mfrench/Singh-FT-Raw-1.pdf

[23] K. Nakahara *et al.*, ”Fault tolerant dynamic reconfigurable device based on EDAC with rollback,” *IEICE Trans Fundamentals*, vol. E89-A, no. 12, pp. 3652–3658, Dec. 2006.

[24] D. Alnajjar *et al.*, ”A coarse-grained dynamically reconfigurable architecture enabling flexible reliability,” in *Proc. IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE5)*, March 2009. [Online]. Available: http://www.selse.org/Papers/selse5_submission_11.pdf

[25] X. She, ”Self-routing, reconfigurable and fault-tolerant cell array,” *IET—Computers & Digital Techniques*, vol. 2, no. 3, pp. 172–183, May 2008.

[26] D. K. Pradhan, *Fault Tolerant Computer System Design*, Prentice-Hall, Englewood Cliffs, N. J., 1996.

[27] S. J. Piestrak, ”Design of residue generators and multioperand modular adders using carry-save adders,” *IEEE Trans. Comput.*, vol. 43, no. 1, pp. 68–77, Jan. 1994.

[28] S. J. Piestrak, ”Design of residue generators and multioperand adders modulo 3 built of multi-output threshold circuits,” *IEE Proc.—Comput. Digit Tech.*, vol. 141, pp. 129–134, March 1994.