Coarse-Grained TMR Soft-Core Processor Fault Tolerance Methods and State Synchronization for Run-Time Fault Recovery

Karel Szurman and Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Bozetechova 2, 612 66, Brno, Czech Republic
Email: {iszurman, kotasek}@fit.vutbr.cz

***Abstract*—Triple Modular Redundancy (TMR) applied with various granularity combined with periodical scrubbing of a configuration memory or with Partial Dynamic Reconfiguration (PDR) for a fault recovery are preferred Single Event Upset (SEU) mitigation techniques used by Fault Tolerant Systems (FTSs) implemented into SRAM-based FPGAs. Usage of PDR and TMR allows FTSs to recover from all transient SEU induced faults and offers run-time fault mitigation compared to scrubbing methods which only correct configuration upsets and are limited by a scrubbing period latency. Reconfigurable TMR architecture may require a global state maintenance after the PDR is applied for a fault removal. In such situation, an operational state of reconfigured circuit copy needs to be synchronized with the remaining circuit copies which were operating during PDR. This paper evaluates existing state synchronization methods used in reconfigurable TMR architectures and soft-core processors, presents our recent research focused on development of a state synchronization methodology compared to the state of the art methods and further investigates strategy for a state synchronization of TMR protected soft-core processor NEO430.**

***Index Terms*—soft-core processor, state synchronization, TMR, SEU mitigation, partial dynamic reconfiguration**

I. Introduction

Emerging technologies used in avionics and space systems have growing demands on computing frequency, latency and data throughput. Therefore, the SRAM-based FPGAs have become broadly used inside these systems although Antifuse and Flash-based FPGA technologies are more resistant to radiation induced faults [2]. These systems are exposed to various failure conditions during their lifetime and the Single Event Effects (SEEs) caused by energetic particles in the harsh space radiation environment are the ones of the most serious. The major concern is SRAM configuration memory which is susceptible to Single Event Upset (SEU), the most common SEE effect. SEUs can cause changes in a state of bi-stable element which affects configuration memory and user logic. Consequently, the usage of SRAM-based FPGAs in safety-critical systems generally requires an implementation of SEU mitigation strategy and employing fault tolerance to operate correctly even in the presence of faults.

Conventional SEU mitigation methods which are accepted across the industry [6] [2] [12] are based on Triple Modular Redundancy (TMR) for fault masking applied on different granularity levels and periodical scrubbing which corrects accumulated configuration memory upsets. When faults are detected in any part of the system implemented into SRAM-based FPGA then a possibility to reconfigure it and to extend the system lifetime exists. For this purpose, Partial Dynamic Reconfiguration (PDR) can be used [3]. In Fault-Tolerant Systems (FTSs), the PDR is frequently combined with TMR methods to implement Fault Detection, Isolation and Recovery strategy. After the reconfiguration of a failed redundant subsystem is finished, its operational state is not actual and need to be synchronized with the correctly operating subsystems of FTS before it is included back into the TMR architecture.

The main goal of the paper is to investigate applicable state synchronization methods for soft-core processors protected by coarse-grained TMR architecture and to describe proposed run-time fault recovery for soft-core processor NEO430.

II. Related research

In the TMR architecture, a single fault in any of three redundant hardware modules will not produce an error at the output as the majority voter will select the correct result from the remaining two correctly working modules. Following rationales for usage TMR exist: (1) the possibility of fault masking by implementing the process of voting, (2) the method of scaling the TMR protection by changing its granularity from fine-grained (FG) to coarse-grained (CG) application, and (3) the availability of tools allowing for a completely automated TMR generation [1]. There is always a trade-off between usage of FG and CG TMR schemes. The FG TMR reduces the highest system frequency in dependence on number of inserted voters added to the critical path. Another compromise is made between power consumption, resources utilization and design robustness. On the other hand, CG TMR is often combined with PDR in reconfigurable FTS with advantage of ability for self-recovery and adaptation by means of reconfiguration or relocation of FPGA design modules.

A critical component which oversees PDR and recovery process is a reconfiguration controller (RC). RC may be used to control periodical scrubbing over FPGA configuration memory frames or to control run-time reconfiguration for selected area in the configuration memory, identified as Partial Reconfiguration Module (PRM). In RC designs, Internal

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

Configuration Access Port (ICAP) interface is often preferred due its easy integration in a HDL design. Various RCs have been developed with the goal to optimize reconfiguration time, minimize FPGA overheads or to provide fault-tolerant design [4] [15]. In our PDR methodology, we use previously developed Generic Partial Dynamic RC (GPDRC) [4]. The GPDRC is based on ICAP. It supports detection and correction of single appearing transient faults caused by SEUs and the mitigation of permanent faults by reducing FT architecture scheme and changing its layout in utilized PRMs. In integration with TMR, the GPDRC receives an information about a PRM including TMR module with localized faulty circuit. After the reconfiguration is finished, the GPDRC initiates the start of the synchronization procedure to return repaired TMR module into a fully operational state.

A. Soft-Core Processor State Synchronization

Authors in [1] identified the synchronization process as essential step for recovering a fault tolerant processor in the TMR after its erroneous instance is reconfigured. Their research evaluates four different synchronization methods for soft-core processor Xilinx Picoblaze which balance differently the trade-off between the synchronization speed and hardware overhead. Following representative methods were evaluated:

1) Synchronization reset – a simple approach which requires bringing all three soft-core processors to a known state and restarting the program execution from the beginning.
2) Synchronization through shared memory with manual trigger – this method is based on sharing memory between processors and their resynchronization by concurrent saving all registers through majority voted signals, followed by concurrent reading. It is started by forcing a JUMP instruction to synchronization routine which reloads processors registers when ongoing calculations are finished. No HW changes are needed but only internal registers accessible by programmer can be synchronized.
3) Synchronization through shared memory triggered by an interrupt – improved method n. 2 which allows also usage of interrupt processing and with advantage of that, immediate resynchronization in an Interrupt Service Routine (ISR). However, this method requires partial HW synchronization of internal registers processor objects which are not accessible by SW, e.g. stack-pointer and stack data memory.
4) Ad-hoc HW synchronization – method which requires custom design of synchronization control logic and corresponding modification in a processor.

Synchronization reset is a method suitable for simple circuits or systems performing periodical calculations (e.g. network packet processing). The time needed for processor reset and program re-execution is the only overhead. Similar approach was used in [8] for Finite State Machine (FSM) based system. It used the principle of predicting a future state to which the system will soon converge and presetting the reconfigured circuit to it. Per authors, the prediction should consider well-known state which will be the best checkpoint state for actual system behavior and conditions.

Synchronization through shared memory was also used in [7] where processors share Block RAM and an interrupt is used to trigger synchronization after the reconfiguration is finished (or it can be invoked periodically). The main benefit is that recovery process can be performed on the fly and an overhead of the synchronization is only the time required to store and restore the processor's state context. The same method is used in [10] for FT multiprocessors architecture. Synchronization through shared memory was used also in [11] for FTS based on 32-bit RISC Plasma soft-core processor which can recover from both transient and permanent errors using scrubbing applied per single frame for detection and configuration rewriting with following state resynchronization.

Thus, the synchronization based on shared memory is suitable for recovery of registers which are accessible by a program. This approach also solves synchronization of processor memories. However, some processors (e.g. Picoblaze) do not have all registers accessible by software. Furthermore, this method does not address internal architectural registers which are hidden before programmers.

According to [9], the key of a proper synchronization is to avoid any blocking situation in a system operation. Proposed method leads the reconfigurable modules into a safe state recognized by the rest of the system and avoids blocking it.

In [13] [14], we proposed and evaluated serial and parallel hardware synchronization for a reconfigurable fault tolerant CAN bus control system where the synchronization of application-related registers was necessary. The serial synchronization is based on the principle of data shifting through internal registers to next unit in the oriented circle topology. In this interconnection, only for the reconfigured unit the synchronization data of predecessor unit are brought to its first register serial input. The other units have connected their serial output to their first register serial input. This principle ensures that only the reconfigured unit will receive new data to its internal registers. The second synchronization approach uses two parallel buses for data transfers between registers in reference and synchronized circuits. Individual registers are addressed through the address bus. The principle of the synchronization is based on sequential addressing of each register through the address bus and enabling write or read signals for circuits which are active during the synchronization process. Reference circuit transfers the content of its addressed registers to the data bus while the synchronized circuit reads these data from the bus and stores them into its internal registers.

III. NEO430 RUN-TIME FAULT RECOVERY

NEO430 soft-core processor is customizable 16-bit soft-core microcontroller compatible with TI MSP430 [5]. The CPU has Harvard architecture including program memory (IMEM) and data memory (DMEM) with configurable sizes and multiple peripherals available which are interconnected

with the CPU through a system bus. The CPU internal architecture is shown in Fig. 1. It includes an Arithmetic Logic Unit (ALU), a CPU control arbiter (CA), an address generator unit (AG) and a register file. These components identify the data path which is necessary to synchronize after the reconfiguration of failed CPU instance is finished. The CPU incorporates sixteen 16-bit registers. Program Counter (PC), Stack Pointer (SP), Status Register (SR) and Constant Generator Register (CG) have dedicated functions. R4 to R15 are working registers for general use. Moreover, also the CPU architecture includes internal registers: Memory Address Register (MAR) in the AG, Instruction Register (IR) in the CA and temporary operand registers (SRC and DST) in the ALU. Besides internal registers, another data are located in processor memories. The DMEM contains interrupt vectors, global variables, the stack and the heap. The IMEM is dedicated to store instructions of the actual application.

An instruction execution is conducted by performing sequence of micro operations, thus the CPU needs several consecutive cycles to complete a single instruction. The execution of micro operations is controlled by the CA FSM which generates the control signals for the data path. The FSM is after reset in default state IFETCH_0, where execution of all CPU instructions starts. The CPU in this state also waits for an incoming Interrupt ReQuest (IRQ) which can bring the CPU out of the SLEEP mode.

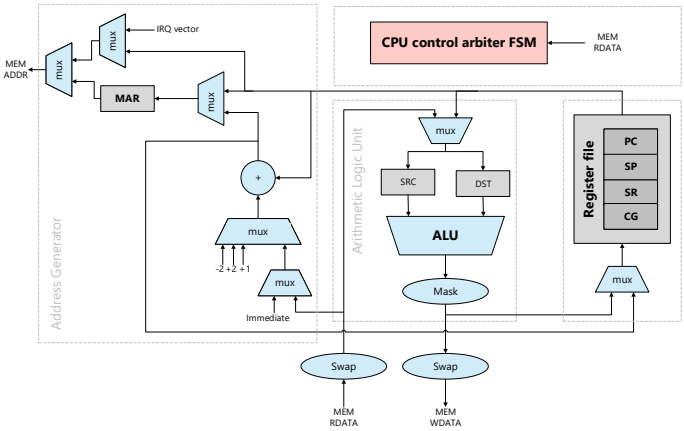

Fig. 1. NEO430 architecture

A. Reconfigurable CG TMR Architecture

The reconfigurable architecture is shown in Fig. 2, the NEO430 CPU is triplicated, the same inputs are connected to all redundant CPUs and their outputs are brought into dedicated TMR majority voters which were improved in order to ensure both functions: fault masking for each single output and identification of a faulty CPU instance and thus, an index of PRM to be reconfigured. A design FPGA floorplan is divided into static and dynamic areas. In the dynamic part, each TMR module is placed into dedicated PRM. The static part of the design includes GPDRC, synchronization controller, ICAP core, flash controller for communication with a parallel flash containing stored partial PRM bitstreams, TMR voters, DMEM and IMEM memories, and all peripherals shared between triplicated CPUs. Note that in this research, we focused mainly on design of CPUs synchronization, therefore we implemented this experimental architecture without considering FT parameters for entire system. Nevertheless, proposed synchronization is developed with the goal to allow easy integration with SW executed on NEO430 CPU without unexpected blocking the system operating.

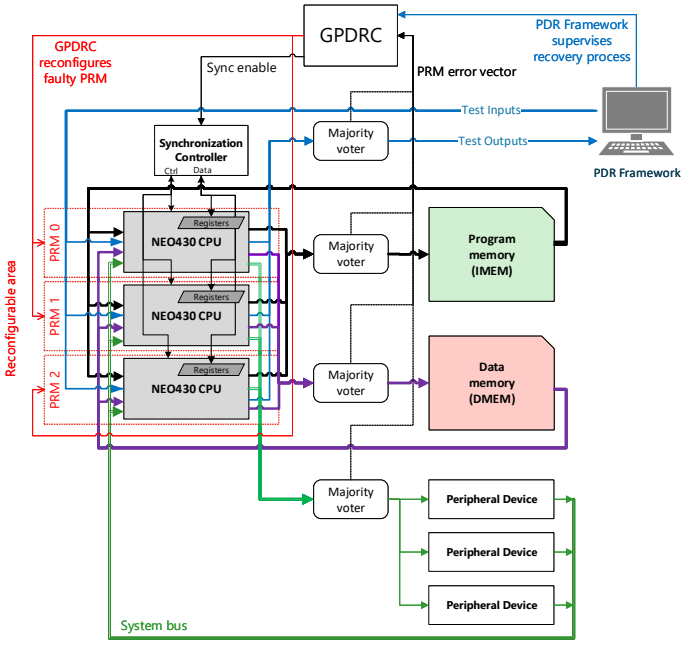

Fig. 2. Reconfigurable FT architecture

B. State Synchronization for NEO430 CPU

As the processor memories and peripherals are in the reconfigurable architecture shared between triplicated CPUs, their synchronization is implicitly performed by majority voters protecting the access from the CPU to shared resources over the system bus. Considering SEU mitigation for entire system, all memories should be protected by means of error detection and correction codes since their triplication would cause unnecessary resource overhead. The microcontroller peripheral devices could be protected by FG TMR or again by CG TMR methods.

Finally, the state synchronization for the NEO430 CPU architecture and entire fault recovery process were implemented by the following sequence of steps:

1) Reconfiguration of a PRM with faulty CPU is started based on a PRM error vector generated by TMR voters which detect a mismatch on compared CPU outputs.
2) A program executed by triplicated CPUs periodically checks the digital input indicating the Sync enable request for performing the state synchronization. The request is generated from GPDRC after the reconfiguration of failed CPU instance is finished.
3) After the reconfiguration, repaired CPU is restarted. During its startup, a program reads the digital inputs

and checks if request for synchronization is active. Since the request was activated by GPDRC, the CPU switched into the SLEEP mode as well.

4) When the program executed by operating CPUs is in a state suitable for synchronization, it will indicate readiness for the hardware synchronization through processor digital output to a synchronization controller. This is a special circuit responsible for parallel addressing of all synchronized registers and their copying from the correctly working CPUs to the recovered one.
5) Afterwards, operating CPUs go into the SLEEP mode. In this state, CPUs are in an idle state IFETCH_0 waiting for an external IRQ generated by the synchronization controller which will activate normal operating mode.
6) In parallel with the previous step, the synchronization controller performs synchronization procedure of all architectural registers (PC, SP, SR, CG, GP R4-R15) and all processor internal registers (MAR, IR, ALU SRC and DST) by their copying from correctly working CPUs into the faulty one, while all CPUs are idle in the SLEEP mode.
7) After the hardware synchronization phase is finished, the external IRQ signal is triggered to bring all CPUs from the SLEEP mode to the operating mode.
8) All three CPUs continue in the program execution from the synchronized state as all internal registers required for correct program execution and architectural registers were synchronized.

In comparison with the existing synchronization methods, and with omitting a reset as a synchronization method, the complete synchronization for a soft-core processor (or just its CPU) as for a complex system, requires combination of all mentioned methods, considering following options:

- Processor memories should be shared between all redundant instances and then, the concurrent access to a memory needs synchronization through majority voters.
- Processor peripherals can be also shared or brought into a safe state. Eventually, all peripherals can be reinitialized by SW after CPU synchronization is finished.
- Some internal CPU registers may be synchronized by SW routine, however CPU always contain some internal architectural registers which should be also addressed. Therefore, some form of hardware implemented synchronization is always necessary.
- Synchronizing ISR can be used for handling an external event triggered e.g. when the PDR is finished or when the next step in the recovery process is on the turn.

IV. Conclusions and future research

The paper described the state synchronization strategy developed for NEO430 CPU included in soft-core processor NEO430. In the design of the synchronization, we focused mainly on the seamless integration with the internal CPU architecture. This was achieved: (1) by implementation of a software trigger allowing the start of the synchronization procedure in a consistent state determined by an executed program and (2) by incorporating hardware mechanism to synchronize all architectural registers. Moreover, this paper also summarizes existing synchronization methods which are applicable for a processor fault recovery.

Our future research will be focused on evaluation of synchronization methods with respect to various reliability and overhead parameters.

Acknowledgment

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 the BUT project FIT-S-17-3994.

References

[1] U. Kretzschmar, J. Gomez-Cornejo, A. Astarloa, U. Bidarte, and J. Del Ser, "Synchronization Of Faulty Processors In Coarse-Grained TMR Protected Partially Reconfigurable FPGA Designs", Reliability Engineering & System Safety, 2016, vol. 151, pp. 1–9.

[2] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications", ACM Comput. Surv., 2015, vol. 47, no. 2, pp. 37:1-37:34.

[3] B. Osterloh, H. Michalik, S. A. Habinc, B. Fiethe, "Dynamic partial reconfiguration in space applications", NASA/ESA Conference on Adaptive Hardware and Systems, vol. 0, 2009, pp. 336–343.

[4] L. Miculka, Z. Kotasek, "Generic Partial Dynamic Reconfiguration Controller for Transient and Permanent Fault Mitigation in Fault Tolerant Systems Implemented Into FPGA", in 17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, Warszawa, PL, 2014, ISBN 978-0-7695-5074-9, pp. 171–174.

[5] S. Nolting, "The NEO430 Processor", github.com/stnolting/neo430.

[6] Xilinx Inc., "Correcting Single-Event Upsets Through Virtex Partial Configuration", XAPP216, June 2000.

[7] S. Tanoue, T. Ishida, Y. Ichinomiya, M. Amagasaki, M. Kuga, and T. Sueyoshi, "A novel states recovery technique for the tmr softcore processor", in International Conference on Field Programmable Logic and Applications (FPL), Aug 2009, pp. 543–546.

[8] J. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt, "Evaluating large grain tmr and selective partial reconfiguration for soft error mitigation in sram-based fpgas", in 15the International On-Line Testing Symposium, June 2009, pp. 101–106.

[9] A. Morillo, A. Astarloa, J. Lazaro, U. Bidarte, J. Jimenez, "Known-blocking synchronization method for reliable processor using tmr & dpr in sram fpgas", in VII Southern Conference on Programmable Logic (SPL), Cordoba, Arg., April 2011, pp. 57–62.

[10] H.-M. Pham, S. Pillement, and D. Demigny, "A fault-tolerant layer for dynamically reconfigurable multi-processor system-on-chip", in International Conference on Reconfigurable Computing and FPGAs (ReConFig), December 2009, pp. 284–289.

[11] M. Fujino, H. Tanaka, Y. Ichinomiya, M. Amagasaki, M. Kuga, M. Iida, T. Sueyoshi, "Fault Recovery Technique for TMR Softcore Processor System Using Partial Reconfiguration", Springer, ICA3PP, 2012, vol. 7439, pp. 392–404, ISBN 978-3-642-33078-0.

[12] P. Adell and G. Allen, "Assessing and Mitigating Radiation Effects in Xilinx FPGAs", NASA Electronic Parts and Packaging (NEPP) Program Office of Safety and Mission Assurance, NASA, JPL 08-9 2/08, 2008.

[13] K. Szurman, L. Miculka, and Z. Kotasek, "State Synchronization after Partial Reconfiguration of Fault Tolerant CAN Bus Control System", in 17th Euromicro Conference on Digital Systems Design, Verona, 2014, pp. 704-707, ISBN 978-0-7695-5074-9.

[14] K. Szurman, L. Miculka, and Z. Kotasek, "Towards a State Synchronization Methodology for Recovery Process after Partial Reconfiguration of Fault Tolerant Systems", in 9th IEEE International Conference on Computer Engineering and Systems, Cairo, 2014, pp. 231-236, ISBN 978-1-4799-6593-9.

[15] L. Gong, A. Kroh, D. Agiakatsikas, N. T. H. Nguyen, E. Cetin, and O. Diessel, "Reliable SEU monitoring and recovery using a programmable configuration controller", in 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, 2017, pp. 1-6, ISSN 1946-1488.