Overview and Investigation of SEU Detection and Recovery Approaches for FPGA-based Heterogeneous Systems

Ediz Cetin[1], Oliver Diessel[2], Tuo Li[2], Jude A. Ambrose[2], Thomas Fisk[1], Sri Parameswaran[2], Andrew G. Dempster[1]

[1] School of Electrical Engineering & Telecommunications
University of New South Wales,
Sydney, Australia
{e.cetin, t.fisk, a.dempster}@unsw.edu.au

[2] School of Computer Science and Engineering
University of New South Wales,
Sydney, Australia
{odiessel, tuol, ajangelo, sridevan}@cse.unsw.edu.au

***Abstract*—Growing international interest in the development of space missions based on low-cost nano-/microsatellites demands new approaches to the design of reliable, low-cost, reconfigurable digital processing platforms. To meet these requirements, future systems will need to include application-specific processors to handle control-dominated tasks and hardware accelerators to cope with data-intensive workloads. COTS FPGAs provide an ideal platform for meeting these requirements with application-specific processors implemented as soft or hard cores along with hardware accelerators on FPGA fabric. However, the main challenge to deploying reconfigurable systems in space is mitigating the impact of radiation-induced Single Event Upsets (SEUs). In considering the design of such heterogeneous systems, we present a survey of techniques commonly employed to guard against soft errors in application-specific processors targeted at ASICs and assess their suitability to FPGA implementation when partial reconfiguration is used to deal with SEUs in logic circuits. Finally, we report on the development of the RUSH payload, to be deployed on the UNSW-EC0 CubeSat due for launch in 2015, to test our design approach.**

Keywords—single event upsets, FPGA, CubeSats, QB50

I. INTRODUCTION

The low-cost, nano-/microsatellite (1-50kg) segment, primarily based on the CubeSat standard and with applications in science, *Earth Observation* (EO) and reconnaissance, is expected to experience between 16.8% and 23.4% compound annual growth over the period 2013-2020 [1]. This burgeoning international interest in the development of satellite-based space missions demands new approaches to the design of *reliable*, *low-cost*, *reconfigurable* digital processing platforms.

To meet these requirements, future space systems will need to include application-specific processors to handle control-dominated tasks and hardware accelerators to cope with data-intensive workloads. Some of these applications include secure and reliable communications, attitude determination and control, guidance, navigation and control as well as on-board image and *Synthetic Aperture Radar* (SAR) data processing and compression. Implementing these systems as *Application-Specific Integrated Circuits* (ASICs) is not viable due to their high cost, long lead times, and inflexibility. The implementation devices most suited to meeting these requirements are *Commercial-Off-The-Shelf* (COTS) *Field-Programmable Gate Arrays* (FPGAs) with application-specific processors implemented as soft or hard cores along with hardware accelerators on FPGA fabric. FPGAs, like custom hardware chips, provide the means for implementing custom processors and accelerators, they can also be reconfigured on demand to perform new or different functions, and have significantly lower lead times and associated costs. Furthermore, by reusing the same device to implement an architectural variation, FPGA reconfiguration can be exploited to reduce mission-critical parameters, such as the system's size, mass and power requirements, which must be kept as small as possible. The main challenge to deploying reconfigurable systems in space, however, is radiation-induced *Single Event Upsets* (SEUs).

As part of our ongoing research activity into rapid recovery from SEUs in reconfigurable hardware [2],[3], we are currently developing a payload for the *University of New South Wales - Educational CubeSat Zero* (UNSW-EC0) CubeSat as part of the European QB50 project to be launched in 2015 [4]. The RUSH (Rapid recovery from SEUs in Reconfigurable Hardware) payload will enable us to carry out in-situ flight testing of various FPGA-based rapid SEU detection and recovery approaches and compare them with vendor specific tools such as *Soft Error Mitigation* (SEM) from Xilinx [5].

This paper considers heterogeneous systems consisting of application-specific processors and hardware accelerators implemented on FPGAs, and investigates the suitability of various circuit- and processor-based SEU detection and mitigation approaches with a view to final deployment on the UNSW-EC0 CubeSat RUSH payload.

The paper is organized as follows: Section II provides an overview of ASIP soft-error mitigation approaches and assesses their suitability for FPGA based implementations. Section III provides details of approaches for rapid recovery from FPGA configuration memory upsets and discusses how these approaches could be applied to ASIPs. The RUSH payload and experiment are detailed in Section IV, while concluding remarks are given in Section V.

II. ASIP SOFT-ERROR MITIGATION

Application-Specific Instruction-set Processors (ASIPs) are tailored by analyzing the characteristics of the specific application(s) that will be executed in the ASIPs. ASIPs are typically used in embedded systems, where properties such as area, power, and performance are critical. An ASIP can be tailored by including custom instructions to improve performance, or by removing unnecessary components based on the mapped application(s) to reduce power, or by adding custom components to improve reliability. In contrast, *General-Purpose Processors* (GPPs) are designed to support a wide range of applications, and are not therefore customized for a particular set of applications. As embedded systems are commonly used in safety-critical applications such as aerospace, automotive, medical electronics, etc., maintaining the system's reliability is of great importance.

ASIPs are typically implemented in standard cells (such as ASICs), where radiation-induced soft errors mainly impact on sequential logic. For example, the register file and on-chip memory are the vulnerable parts of ASIPs implemented as ASICs, whereas the circuits themselves, such as the adder circuit, remain largely unaffected. However, when an ASIP is implemented in an FPGA device, the entire circuit is implemented in configuration memory, including the combinational circuit elements and the component interconnections. Since SRAM-based FPGA fabrics are susceptible to radiation-induced SEUs, the functionality of FPGA-based ASIPs can be affected, and unless they go corrected, configuration memory SEUs have the appearance of permanent errors in ASICs.

Existing processor-level soft-error countermeasures for ASIPs can be grouped into two major categories: hardware (Section II-A1, II-A2, and II-A3) and software (Section II-B1, and II-B2) based approaches. In this section, we present and elaborate a few representative genres of techniques in both categories when considering FPGA implementations. The fundamental idea behind these techniques, which detect and recover from errors, is to add redundancy into the system with regards to architectural states e.g., register file and memory. The techniques are compared with the literature on SEU mitigation for soft FPGA-based GPPs in Section II-C. Note that since *Error-Correcting Codes* (ECC) are well established for storage elements such as the register file and memory, in this discussion we focus on the entire processor or the execution of instructions in the datapath pipeline. For each genre of techniques, we introduce the concept, system impact, and applicability to FPGA implementation.

A. Hardware-based soft error mitigation approaches

1) Instruction Space Triple Modular Redundancy:

Instruction space triple modular redundancy (space-TMR) adds two redundant instruction executions in parallel with the usual instruction execution, and recovers the error by selecting the result in majority with minimal overhead on processor performance. Theoretically, *N*-MR is able to detect errors when *N*=2 by comparing two results from two modules, and recover errors when *N*=3 by performing majority voting with three results from three modules.

Since ASIPs are typically implemented using pipelined datapaths, each pipeline stage or indeed the entire pipeline can be triplicated based on the cost constraints such as area, power, and performance (delay). Fig. 1 depicts an example for space-TMR where the *EXecution pipeline stage* (EX) is triplicated, and the three outputs are passed to a voter, before the final commit of the instruction at the *Write-Back* (WB) stage. The other stages could be triplicated as well to achieve better reliability, however this would incur additional area and power overheads. The impact of the approach on the processor architecture is to triplicate hardware components that execute the instructions i.e. the EX unit and to add a majority voting hardware unit. Thus, the main impact is hardware complexity, which leads to additional area and power costs. The additional hardware complexity is slightly more than twice that of the EX unit.

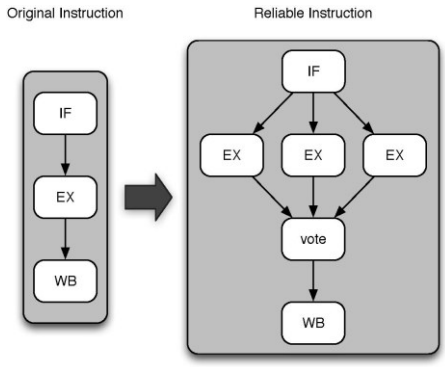

Fig. 1. Instruction space-TMR

Considering FPGA implementations, instruction space-TMR is applicable to soft processors for which the RTL description of the processor is available so that the required modifications to the architecture can be made. However, modifying the architecture is infeasible for hard-core processors and commercially acquired soft processors for which the RTL is generally not provided.

2) Instruction Time Triple Modular Redundancy:

Instruction time-TMR triplicates the execution of an instruction in a temporal manner. The redundant executions are generated by re-issuing the instruction two additional times. The result of the instruction is committed after majority voting on the three results. For example, the work in [6] locks the *Program Counter* (PC) and executes the same instruction three times starting from the *Instruction Fetch* (IF) stage. In the first two executions, the output of the instruction is saved without committing at the WB stage. In the last execution, the three outputs are voted upon and then committed at the WB stage.

The major architectural impact of instruction time-TMR is the logic to handle re-issuing of the instruction, temporary storage to hold the results before majority voting, and the majority-voting unit. The additional hardware is insignificant in comparison to instruction space-TMR. However, the performance of the processor is decreased by a factor of 3, due to the additional issues per instruction.

The applicability of instruction time-TMR to FPGA implementation is the same as for space-TMR. For hard and commercial soft IP, adding the re-issue logic, the temporary storage and majority voter are infeasible. For soft processors for which the RTL is available, the approach could be used.

3) Instruction Checkpoint Recovery:

Instruction *Checkpoint Recovery* (CR) is a recovery-only solution to soft errors or transient faults. Performing CR at each instruction within a basic block [7] allows the processor to save a subset of the architectural state as a backup. These preserved values can be used to re-write/restore the architectural state that was modified by the basic block (this process is called rollback or restoration), when an error is detected at the end of the block. Generally, instruction CR ensures that the execution of the program is backed up and can be recovered periodically.

For example, the original instruction *ADD R2, R3, R4* adds the values of register *R3* and *R4* in the register file and writes the result into register *R2*. With CR enhancement, this instruction will first save the current value of *R2* into a specialized reliable storage unit before committing the new value at the WB stage. Similarly, all the instructions in the current basic block that modify the values of the register file or data memory are enhanced to store the current values before being committed. If an SEU is detected at the end of the basic block, an interrupt is triggered to execute specialized rollback instructions that fetch the previous values from storage to write them back into the corresponding locations of the register file or data memory. It is worth noting that comparisons (branch instructions) are customized to trigger rollbacks internally when errors are detected.

A variety of detection techniques can be applied with CR. One possibility is a control-flow based detection technique [8]. In this work, a compile-time signature of every basic block of the program is calculated by performing an XOR of the machine code (however more advanced encoding techniques could be applied as well). These signatures are then inserted into the corresponding basic blocks. At runtime, specialized hardware calculates a signature for the executed instructions. At the end of each basic block specialized hardware in branch instructions is used to compare the compile-time and runtime signatures. A mismatch of the signatures indicates the presence of an SEU in the instruction stream.

Instruction CR augments the architecture of the processor with: a checkpoint buffer, logic for managing the update of the checkpoint buffer i.e. reading architectural states and writing to the checkpoint buffer, and logic for flushing the pipeline and rewriting architectural states. The detection method imposes additional architectural modifications. There are similar limitations to the application of this technique to FPGA-based ASIPs as for the previous two approaches.

B. Software-based soft error mitigation approaches

1) Software-Implemented Error Recovery:

Software-Implemented Error Recovery (SIER) is a solely software-based approach. Following TMR principles, SIER triplicates each instruction to allow majority voting as the program is executed [9]. Each instruction copy uses different registers and different memory locations so as to not interfere with the others. As all instructions are processed using the original hardware the processor architecture does not need to be modified. For example, instruction *ADD R2, R3, R4* is transformed to three instructions *ADD R2, R3, R4*, *ADD R2', R3', R4'*, and *ADD R2", R3", R4"*. Where *R2, R2'* and *R2"* are the different registers representing the same variable in the program. These three instructions are executed sequentially. An extra segment of code is inserted after these three instructions are executed to vote on the value of *R2* at runtime.

SIER necessitates modification of the compiler backend e.g., to perform register allocation. The voting segment can be added directly into the program. The SIER program code length is at least there times that of the original code, but the processor hardware is not modified. Applying SIER to FPGA implementations is feasible since SIER does not modify the processor architecture. However, memory costs might increase due to the increased code size.

2) Profile-Guided Code Transformation:

Profile-Guided Code Transformation (PGCT) alters the software code based on an analysis of the program. The program is profiled to understand the dependencies between instructions, liveliness of variables/registers, and the execution frequency of instructions to determine the vulnerability of each instruction. The transformations include loop unrolling and data type reassignment [10]. By transforming the code, the variables that are estimated to be vulnerable to soft error are enhanced (to reduce their chance of corruption). For example, considering instruction *ADD R2, R3, R4*, decreasing the time period that a variable/register (e.g., *R2* or *R3* or *R4*) spends in more vulnerable sequential logic (e.g., register file) and increasing the time period that it spends in less vulnerable sequential logic (such as memory with ECC) can increase the reliability of that variable. Hence, by applying these transformations, the vulnerability of the program can be reduced by up to 90%, as reported in [10].

PGCT induces no hardware complexity cost. However, the code size might change and the resultant performance can be degraded as well. To implement PGCT, the compiler backend must be modified to allow the transformation. In addition, knowledge of the processor architecture is needed to perform the vulnerability analysis. For example, to calculate the vulnerability of an instruction, the area and logic type of the hardware components occupied by that instruction are used. This technique is applicable to FPGAs since no hardware modifications are needed. However the increase in code size may affect the memory requirement.

C. Discussion

Table I summarizes the processor-level techniques discussed in this section. The techniques of column 1 are evaluated with respect to the characteristics of cols 2–6. Overall, the hardware-based techniques induce considerable area overheads, whereas the software-based ones result in execution time and instruction space penalties. With regard to FPGA applicability, most of the hardware-based techniques require the baseline processor architecture to be transparent and described in RTL, while software-based techniques simply require more memory.

TABLE I. SUMMARY OF PROCESSOR-LEVEL SEU MITIGATION TECHNIQUES

Technique	Hardware Impact	Software Impact	Performance Impact	FPGA Applicability	
				Hard/Commercial Soft IP	*RTL Soft IP*
S-TMR	Significant (>3x)	None	Critical path can be impacted by voters	N/A	Applicable
T-TMR	Insignificant	None	Significant (>3x)	N/A	Applicable
CR	Dependent on number of states and storage type	Insignificant (rollback routine)	Insignificant	N/A	Applicable
SIER	None (memory for additional code lines)	Significant (>3x)	Significant (>3x)	Applicable	Applicable
PGCT	None (memory for additional code lines)	Dependent on code	Dependent on transformations	Applicable	Applicable

SEU mitigation in soft FPGA-based GPPs has been studied extensively – we outline some representative examples of the work. [11] and [12] studied dual modular redundancy (DMR) at the processor level, operating Leon2 [11] and MicroBlaze (MB) [12] in lock step, and performing checkpointing and recovery to correct datapath memory errors. Configuration memory errors (CMEs) were corrected by scrubbing and partial reconfiguration (PR), respectively. [13] used TMR to protect MBs and synchronized the register state after PR to correct CMEs. [14] have employed DMR at the IF and EX stages of an OpenRISC processor; instruction execution is stalled, the faulty stage is reconfigured, and the instruction is re-executed when an error is detected. The work to date has tended to focus on mitigation techniques and reported the impact on area and performance. In contrast, our research goals are to achieve specified performance criteria (area, speed and power) while meeting recovery time guarantees.

In soft ASIPs targeted at FPGAs, time-TMR and CR techniques do not guard against configuration memory errors because they do not provide any redundancy in the processing hardware. Currently, we therefore focus on spatial-TMR and outline our approach to recovering from configuration memory errors in the next section.

III. RAPID RECOVERY FROM FPGA CONFIGURATION MEMORY UPSETS

The configuration memory of COTS FPGAs, being implemented in SRAM, is as prone to corruption due to radiation as the memory elements (FFs and BRAMs) of user circuits. Therefore, when COTS FPGAs are used in radiation prone environments, it is necessary to provide protection from radiation and/or methods for detecting and recovering from radiation-induced configuration memory errors. Moreover, in time critical applications, it is also desirable to detect and recover from errors very quickly.

There are two principal methods for detecting and recovering from configuration memory SEUs in COTS FPGAs. The first, direct method, typically referred to as scrubbing, involves scanning the configuration memory checking for upsets either via ECC associated with individual configuration frames, or by comparison with a golden reference stored off-chip in protected memory. Any elements that have been modified are refreshed in the course of the scan. FPGA vendors, such as Xilinx, provide in-built components to perform this function [5]. An alternative, indirect method, involves checking the behaviour of the user circuit, and reloading the circuit configuration if the circuit no longer behaves as expected [3],[15]. In the latter case, TMR is typically employed to identify the module in error, and dynamic partial reconfiguration is used to reconfigure the erroneous unit. Built-in self-tests could also be employed to check correct functioning of the user circuits.

The scrubbing technique is usually deployed as a background process that operates periodically. There can therefore be a considerable delay between errors occurring and them being detected and corrected. The TMR-based approach, on the other hand, is able to detect errors in the unit that is affected by checking for repeated errors. If the module that is triplicated is acyclic, then the occurrence of repeated errors in the same unit suggests its configuration memory is corrupted since transient errors affecting the datapath only give rise to isolated errors. Of course, if the module includes feedback paths, then even a transient error can lead to recycling of the erroneous value, and potentially give rise to multiple errors at the output. In any case, when the TMR-based approach determines that a unit is in error, it can trigger a *partial reconfiguration* (PR) of that unit, which can therefore be expected to incur less delay in correcting the error and require less energy as partial reconfiguration is only triggered when needed.

Regardless of the detection and configuration memory correction method used, thought must also be given to recovering the state of the affected user circuits. This detail is less comprehensively studied in the literature. When scrubbing is used, the designer needs to employ additional mechanisms, such as TMR and checkpointing, in the user circuit to recover the state. TMR-based approaches rely on checking each feedback state or on waiting until the circuit enters a known state before resynchronizing the constituent modules of a TMR component [16]. In [3] the circuit to be protected is partitioned into acyclic components with each feedback edge being voted upon (see Fig. 2). After a module is reconfigured, its state is resynchronized with that of its siblings when the inputs to the module (including any feedback edges that have been voted upon) have emerged as outputs. The latency of the component therefore determines the resynchronization delay.

As outlined in the previous section, we propose using spatial-TMR to protect ASIPs for which the RTL description is available. It is relatively straightforward to then triplicate any single stage of a pipelined architecture whereby the pipeline register contents are voted upon. For example, triplicating just the EXecute stage (as depicted in Fig. 1)

involves instantiating three copies of the ALU and the result (EX/WB pipeline) registers. The contents of the result registers are voted upon, and the majority value is then again used as a singular value to access memory or to be written back to the register file. This scheme allows transient errors in any single EX unit to be overwritten. Since the EX stage is invariably acyclic in structure, when any one unit is found to be in error over successive clock cycles, it is more likely that this has been caused by a configuration memory upset than for it to have been caused by successive datapath SEUs. A partial reconfiguration of that unit is then triggered. While the unit is being reconfigured, its two siblings continue to operate and the voter continues to check that they agree. After the partial reconfiguration of the erroneous unit has been completed, the output of the reconfigured unit can once again be expected to agree with that of its siblings after the next instruction is executed and its result is registered.

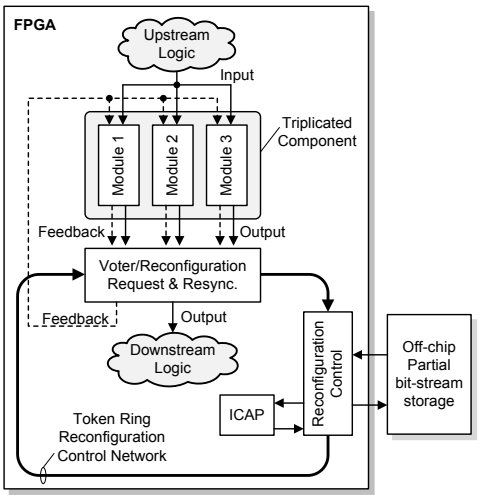

Fig. 2. PR-based recovery from configuration memory SEU errors

The same approach can be used to protect the instruction decode, register fetch, and register writeback logic after an ALU or memory load instruction. The on-chip control logic for off-chip memory accesses on instruction fetches, loads and stores can also be triplicated. Since off-chip memory is readily protected with ECC, triplicating the storage as well should not be necessary except in the most sensitive of applications.

For the above approach to be applicable, each component that is to be protected must be partitioned into acyclic sub-components. This is also a requirement of any extraneous accelerator or glue logic that is to be protected. Some means of coordinating the requests for reconfiguration between many voters and the reconfiguration controller also needs to be implemented. In [2], we outlined and assessed a token-ring architecture we use to implement a *Reconfiguration Control Network* (RCN) for this purpose (Fig. 2). The resulting system is resilient to radiation-induced errors as long as these errors don't re-occur at time intervals that are shorter than the time it takes to detect a configuration memory error, communicate a reconfiguration request, perform a partial reconfiguration, and resynchronize the reconfigured module. This design criterion determines the maximum component size and latency we need for reliable operation [2], [3].

IV. The QB50 RUSH Payload and Experiment

The QB50 project, funded through the European Union Framework Programme 7 (FP7) and overseen by the *Von Karman Institute* (VKI) in Belgium, is a planned network of around 50, 2U and 3U CubeSats due to launch in 2015 into *Low Earth Orbit* (LEO) that aims to provide a temporal and spatial image of the largely unexplored lower thermosphere. The individual CubeSats of the QB50 mission are to be developed by various universities around the world compliant with the QB50 requirements [17] and are expected to carry one of the three VKI sensor payloads.

RUSH is one of three payloads that are currently under development for the UNSW-EC0 QB50 CubeSat. The primary objective of this payload is to demonstrate and validate new approaches to rapidly recovering from SEUs in reconfigurable hardware. The experimental goals of the payload are:

- Demonstrate and validate the partial reconfiguration approach to rapidly recovering from SEUs in reconfigurable hardware.
- Compare reconfiguration time and power consumption of scrubbing with partial reconfiguration approach
- Map SEU event occurrences in the thermosphere
- Demonstrate in-orbit reconfiguration.

Refer to Fig. 3 for a block diagram of the RUSH payload.

As can be observed from Fig. 3, at the heart of the RUSH payload design is a Xilinx Artix 7 XC7A200T FPGA, chosen for its high logic density to power consumption ratio. The FPGA is connected to two flash devices. One stores the base configurations for the FPGA, while the other stores the partial bitstreams of the modules that can be partially reconfigured via *Dynamic Partial Reconfiguration* (DPR). The FPGA is connected via a UART interface to a *Microcontroller Unit* (MCU) which acts as an interface between the FPGA and the UNSW-EC0 CubeSat system bus, and communicates with the *On-Board Computer* (OBC) via the I2C interface. Additionally, the MCU oversees the overall operation of the RUSH payload and controls the power-up/down of the FPGA as well as logging of the SEU detection and recovery statistics along with power consumption details. To fulfil the requirements for the MCU in the proposed design, a Microsemi SmartFusion 2 *System-On-Chip* (SoC) was selected. Furthermore, since the SoC is based on non-volatile FLASH memory it is resilient to SEUs [18]. A small number of additional components provide ancillary functions such as providing regulated power, clock sources, programming interfaces and status indicators.

The primary objective of the RUSH experiment is to test and validate new approaches to rapidly recovering from soft errors in reconfigurable hardware involving accelerator logic and soft ASIPs and to compare the performance of the approach with that of the Xilinx SEM controller [5]. To this end, two configurations will be developed that are essentially identical in terms of their resource utilization, whereby one configuration will employ the method outlined in Section III to guard against and recover from soft errors in user logic and configuration memory, and the other configuration will utilize

the SEM controller to continuously scan and scrub the FPGA configuration memory. To enable comparison of SEU susceptibility and recovery, the two configurations comprise essentially the same circuitry, but the SEM configuration will not partially reconfigure its triplicated components.

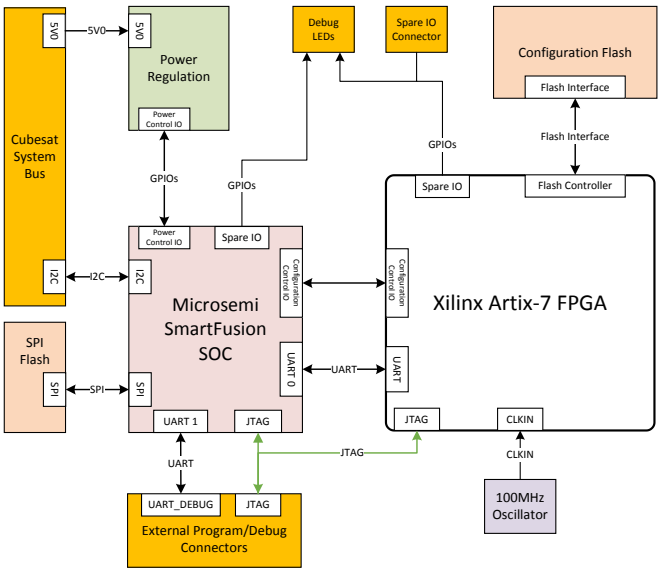

Fig. 3. RUSH payload block diagram.

The experiment will play a vital role in testing the susceptibility of Artix-7 FPGAs in low-earth orbit, and will demonstrate the use of dynamic partial reconfiguration on an FPGA in space. The design will be composed of two base components: a *Portable Instruction Set Architecture* (PISA)-based *Advanced Encryption Standard* (AES) custom processor with triplicated execution units, and a *Block Adaptive Quantization* (BAQ) circuit, chosen for its utilization of all FPGA resource types (LUTs, FFs, DSPs, and BRAMs). These base components will be replicated to fill the FPGA area, thereby creating the largest possible surface for SEUs to occur. During the experiment the SEU events will be logged by the MCU and the time, location, and time to recover will be transmitted to Earth when UNSW-EC0 passes over any of the ground stations available for the QB50 mission. Due to power limitations of the UNSW-EC0, the RUSH experiment will not run continuously. To deal with this, the available uptime will be evenly distributed between the two configurations. Furthermore, activity of both configurations will be scheduled such that they occur at similar times and locations.

V. Conclusions

We have argued for the need to support soft ASIPs and logic in COTS FPGAs for future low-cost space missions. We have surveyed techniques commonly employed to guard against soft errors in ASIPs targeted at ASICs, where the processor state is susceptible to corruption and assessed the applicability of these techniques to ASIPs implemented on FPGAs. We have outlined an experiment that is to be conducted as part of QB50 in 2015 involving an off-the-shelf Xilinx Artix-7 FPGA that will be flown into a low-earth orbit. As part of the experiment we will trial approaches to protecting soft processor and logic circuits that are expected to result in quicker recovery and lower power consumption than standard techniques. Our experiment will also help to gauge the susceptibility of modern high-density COTS FPGAs to SEUs in the thermosphere. If our methods prove to be beneficial, we aim to refine and generalize them to provide a low-cost, rapid development platform for protecting FPGA-based processor and logic systems against radiation-induced soft errors.

References

[1] SpaceWorks, "Nano/Microsatellite Market Assessment". 2013, Available at: bit.ly/17p9M5F

[2] E. Cetin, O. Diessel, L. Gong, V. Lai, "Reconfiguration Network Design for SEU Recovery in FPGAs", ISCAS, June 2014.

[3] E. Cetin, O. Diessel, L. Gong, V. Lai, "Towards Bounded Error Recovery Time in FPGA-Based TMR Circuits Using Dynamic Partial Reconfiguration", *FPL*, September 2013.

[4] QB50 Project Description, [Online]. Available: https://www.qb50.eu/index.php/project-description

[5] "LogiCORE IP Soft Error Mitigation Controller v4.1 Product Guide"," *Xilinx App. Note PG036 April 2, 2014*.

[6] T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran, "Raster: Runtime adaptive spatial/temporal error resiliency for embedded processors," in DAC, 2013 pp. 62:1–62:7.

[7] T. Li, R. Ragel, and S. Parameswaran, "Reli: Hardware/software check-point and recovery scheme for embedded processors," in *DATE, 2012*, pp. 875–880.

[8] R. G. Ragel and S. Parameswaran, "Impres: Integrated monitoring for processor reliability and security," in DAC, 2006, pp. 502–505.

[9] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36–47, 2007.

[10] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: Embedded code generation aiming at reliability," in *CODES+ISSS*, Oct 2011, pp. 237–246.

[11] M. Sonza Reorda, M. Violante, C. Meinhardt, and R. Reis, "A low-cost SEE mitigation solution for soft-processors embedded in systems on programmable chips", *DATE*, 2009, pp. 352-357.

[12] H.-M. Pham, S. Pillement, S. J. Piestrak, "Low-Overhead Fault-Tolerance Technique for a Dynamically Reconfigurable Softcore Processor," *IEEE Trans. Comp.*, vol. 62, no. 6, pp. 1179-1192, 2013.

[13] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and I. Sueyoshi, "Improving the Robustness of a Softcore Processor against SEUs by Using TMR and Partial Reconfiguration," *FCCM*, 2010, pp.47-54.

[14] A. Vavousis, A. Apostolakis, and M. Psarakis, "A Fault Tolerant Approach for FPGA Embedded Processors Based on Runtime Partial Reconfiguration," *JETTA*, vol. 29, no. 6, pp. 805-823, 2013.

[15] C. Bolchini, A. Miele, and D. M. Santambrogio, "TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs," in *DFT*, Sep 2007, pp. 87–95.

[16] C. Pilotto, J. R. Azambuja, and L. F. Kastensmidt, "Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications," in *SBCCI*, 2008, pp. 199–204.

[17] "QB50 System Requirements and Recommendations and Interface Control Document, Issue 3," VKI, Tech. Rep., Feb. 2013.

[18] Microsemi Corp., SmartFusion Customisable System-on-Chip (cSoC), 2013. http://www.actel.com/documents/SmartFusion_DS.PDF