Partially Reconfigurable System-on-Chips for Adaptive Fault Tolerance

Shaon Yousuf, Adam Jacobs, Ann Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computing Engineering, University of Florida, Gainesville, FL 32611
{yousuf, jacobs, ann}@chrec.org

Abstract—Due to the runtime flexibility of modern dynamically reconfigurable SRAM-based FPGAs, FPGA devices have become an attractive platform for developing system-on-chips (SoCs) for space applications (*space SoCs*). However, since the FPGA's SRAM is highly susceptible to space radiation, system reliability is a primary concern for space SoCs. To maintain system reliability and mitigate space radiation effects, space SoCs must be designed with redundant copies of system functionality. Space SoCs must contain enough redundancy to ensure system reliability for the highest anticipated radiation level, which imposes a large area overhead. However, since radiation levels vary based on the system's orbital position, the system does not always require the highest level of redundancy. Space SoCs that can adapt the system redundancy based on the current radiation level can achieve more effective device utilization. In this paper, we present a flexible, FPGA-based, adaptive SoC for space system development. Our space SoC leverages partial reconfiguration to dynamically adapt the system's level of redundancy according to varying radiation levels. We present a software algorithm to manage the system's adaptability, implement the SoC on a Xilinx Virtex-5 device, and evaluate the SoC's resource utilization using the International Space Station's orbit.

Keywords–Adaptive fault tolerance, FPGA, partial reconfiguration, reconfigurable architectures, system-on-chips

I. Introduction

Many space systems (e.g., orbiting satellites, spacecrafts, space probes, etc.) use remote sensing applications in conjunction with attached sensors to gather information about a target of interest (e.g., terrain, atmosphere, rainfall, etc.) from a distance. These applications collect and transmit large amounts of data to ground systems or other space systems for further processing. However, as remote sensing applications grow in complexity and the rate and amount of data collected increases, the overall system performance is bottlenecked by the limited communication bandwidth. To overcome this communication bandwidth limitation, space system designers leverage on-board data processing.

On-board data processing reduces a space system's communication bandwidth requirements by locally (on-board) processing the data and only transmitting the processed results. System-on-chips (SoCs) provide a good platform for increasing on-board data processing capabilities, however, since increasing a space system's processing capabilities increases the system's payload, SoCs that are optimized/customized for use in space (*space SoCs*) can be leveraged to provide cost effective, high performance, and reliable data processing.

Traditionally, space SoCs use specialized radiation hardened (rad-hard) devices for protection against high-energy charged particles (radiation) in space. Rad-hard devices enable reliable on-board data processing but the rad-hard process requires the design to be fixed/static, which forces the SoC to provide all the application's required functionality all of the time. This fixed design requirement can impose large hardware requirements and significantly increase the system's payload. However, since an application's functionality requirements typically vary over time, all functionality is not needed all the time. Commercial-off-the-shelf (COTS) SRAM-based field-programmable gate arrays (FPGAs) can leverage this variation in functionality to time-multiplex mutually exclusive application functionalities in shared hardware resources and decrease overall hardware requirements.

FPGAs provide several benefits as compared to rad-hard devices, such as reprogrammability, lower cost, and higher performance. However, SRAM-based FPGAs are not rad-hard, requiring fault tolerant (FT) techniques to provide reliability. Even though most FT techniques rely on redundant functionality, FPGAs can be leveraged to reduce much of this additional hardware overhead by adapting the level of hardware redundancy based on the current radiation level.

FPGAs offer two reconfiguration (reprogrammability) methods: full reconfiguration, which halts and reconfigures the entire FPGA, and partial reconfiguration (PR), which only halts and reconfigures the portion of the FPGA (partially reconfigurable region (PRR)) that changes. Full reconfiguration can impose performance overhead as the entire system must halt execution during the reconfiguration process. PR's isolated reconfiguration mitigates this performance overhead because only the reconfigured PRR halts while the remainder of the system continues executing.

PR can be leveraged to build efficient and cost-effective space SoCs, but FT PR SoC design is complex. System designers must partition the SoC's functionality into a static region and one or more PRRs. The static region's functionality remains fixed and the PRRs are reconfigured with partially reconfigurable modules (PRMs). To vary the system's FT, system designers must design an *adaptive fault tolerance* (AFT) controller to vary the system's reliability mode. For example, orbital positions with low radiation require less/no redundancy and orbital positions with high radiation require more redundancy. In low radiation orbital positions, redundant hardware can be shut down to save power or reconfigured to extend the system's current functionality (e.g., perform more intensive data processing). In high radiation orbital positions, the level of redundancy can be increased to account for the higher error probability. In contrast, a statically designed system constantly provides the highest level of redundancy and therefore, the system's FT is over provisioned in low radiation orbital positions. Although static systems are simpler to implement, it is necessary to leverage PR to decrease payload and cost in space systems. Moreover, once an FT PR SoC has been designed, the SoC can be leveraged as a base platform to deploy a multitude of different space applications.

In this paper, we present an AFT PR SoC that leverages VAPRES—the Virtual Architecture for Partially Reconfigurable Embedded Systems [11]—which provides an ideal platform for space SoCs. VAPRES contains a MicroBlaze soft-core processor [1] connected to a designer-specified number of PRRs, which are reconfigured via the internal configuration access port (ICAP) [1]. VAPRES independently clocks each PRR, which allows unused PRRs to be shutdown/halted and reduces power consumption for low performance applications. We design a software-based AFT controller for the MicroBlaze that dynamically reconfigures the PRRs and changes the reliability mode according to the current orbital position and shuts down unused PRRs to save power.

Our AFT PR SoC leverages the AFT controller and the VAPRES architecture to provide a flexible and scalable base platform for space SoC development, as compared to previous FT FPGA-based PR SoCs, which are either FT system modeling frameworks [5][6][10] or specialized fixed architectures [4][19]. We implemented our AFT PR SoC on a Virtex-5 FPGA [1] (this device choice is arbitrary and does not affect our system's applicability to other devices) and measured the resource utilization using the International Space Station's (ISS's) orbit.

II. Background and Related Work

Space radiation causes single event upsets (SEUs) [24] in an FPGA's SRAM-based configuration memory by flipping the logic state (0 to 1, or vice versa) of static memory elements (e.g., latches, flip-flops, RAM cells, etc.), which may cause deviations in the FPGA's expected functionality and/or performance. Since SRAM-based FPGAs are not rad-hard devices, FT techniques must be used to protect an FPGA application from SEUs.

Traditional FPGA FT techniques consist of fault mitigation along with configuration scrubbing. Fault mitigation replicates an application's functionality, where the replicated functionality serves as a backup in the case of hardware failure. For example, triple modular redundancy (TMR) [16][17] replicates the application's functionality (components) three times and a voter compares the outputs for inconsistencies. Although fault mitigation protects an application from SEUs, accumulated SEUs can affect all replicated copies, potentially culminating in system failure. To remove accumulated errors, the configuration memory is periodically refreshed using configuration scrubbing [24].

In the area of COTS FPGA-based FT systems, the Remote Exploration Experimentation (REE) project by NASA JPL [2] proposed an FT high performance supercomputer for space using FPGA-based processors and communication network. REE adapted the FT level using a software middleware layer called the Adaptive Reconfigurable Mobile Objects of Reliability (ARMOR). The REE application and the middleware contained mechanisms for both fault detection and recovery. Troxel et al. [20] presented similar work using the NASA Dependable Microprocessor (DM), which consisted of redundant FPGA-based data processors and a rad-hard system controller. A mission manager and the DM middleware provided a general-purpose management system to adapt the FT level. Even though the REE project, the DM system, and other similar FPGA-based FT systems [22][23] did not leverage PR, PR could have been incorporated to increase the system's flexibility and performance.

In the area of FT PR SoC system modeling frameworks, Erdogan et al. [7] proposed a reconfigurable, adaptive fault tolerant (RAFT) framework that used a five layered hierarchical-based approach for fault detection and correction. The five layers included the monitoring and diagnostic layer, the application layer, the FT layer (FTL), the adaptive processing layer, and the hardware redundancy layer. The RAFT framework used multiple FPGAs running in parallel, where the FTL managed the required redundancy. The FTL assigned redundant components to the FPGAs, stored information on the components' location, and recovered from component failures when results were inconsistent. Di Carlo et al. [6] presented a similar FT framework that used a hybrid platform with a general purpose processor (GPP) connected to an FPGA. The authors defined two packages, the exploitation package and the software support package, which abstracted system designers from the hardware level and used the GPP software to perform fault mitigation operations, respectively. Iturbe et al. [9] proposed a Reliable Reconfigurable Real-Time Operating System (R3TOS) layered architecture framework to provide fault tolerance in FPGAs. The R3TOS consisted of an operating system kernel to provide high-level software developer control, a real-time scheduler and placer to allow access and control the ICAP, a 2-D resource allocator that allocated tasks to non-damaged parts of the FPGA, a network-on-chip manager and a dynamic router to allow proper communication with hardware tasks, a diagnostic unit to verify correct system functionality, and an inter-device coordinator to enable communication between separate instances of R3TOS running on different devices. Jacobs et al. [10] proposed a reconfigurable fault tolerance (RFT) framework that adapted to changing space radiation levels by switching between three different reliability modes (Triple-Modular Redundancy Mode (RFT-TMR), High-Performance Mode (RFT-HP), and Self-Checking Pair Mode (RFT-SCP)). The authors proposed a hardware controller that switched between RFT modes to efficiently manage hardware resources. We point out these previous works were only system modeling frameworks and were not implemented in hardware.

In the area of FT PR SoC architectures, Doumar et al. [4] presented an FT PR JPEG2000 compression SoC implemented on the Stratix-II and Cyclone-II FPGAs. FT operation was periodically verified by testing each JPEG2000 component. The tested component's inputs, outputs, and internal registers were serially passed into a fault-testing PRR, where the original component's functionality was emulated using several test vectors. If the test vectors determined that the original component was faulty, the fault-testing PRR assumed the original components functionality, otherwise the tested component continued operation. Straka et al. [19] proposed and developed three specialized FT PR architectures on a Xilinx Virtex-5 device: a TMR architecture with comparators, a duplex architecture with checkers, and a duplex architecture with one checker and one corroborative checker (the checkers and comparators are specialized voters). The comparators formed a separate component and compared the outputs of the TMR components for errors. The checkers were built into each duplex component to detect and isolate errors from within the component. The corroborative checker was similar to a checker but also checked the output of the other duplex component. All errors were reported to a PR hardware controller, which reconfigured the failed component. We point out these architectures are specialized, not scalable and flexible, and do not adapt to varying orbital position radiation.

III. VAPRES Overview

VAPRES is an architecturally customizable multipurpose PR FPGA SoC consisting of a MicroBlaze soft-core processor connected to a designer-specified number of PRRs. VAPRES's highly-customizable architectural parameters provides a scalable system to enable system designers to meet varying design specifications for a wide range of application domains. Since a software controller on the MicroBlaze can reconfigure the PRRs using software commands sent to the ICAP, VAPRES is an easy-to-program foundation for PR FT (as opposed to hardware-based PR controllers). Each PRR has an independent and configurable clock, which allows PRMs to operate at different clock frequencies in accordance with an application's operational requirements. A customizable inter-module communication architecture—SCORES [11]—provides inter-PRR communication. In this section, we provide a brief overview of the VAPRES architecture and the VAPRES system and application design flows.

A. VAPRES Architecture

The VAPRES architectural layout consists of two regions: the controlling region and the data processing region.

1) Controlling Region

The controlling region contains the MicroBlaze and the static peripherals (e.g., ICAP, UART, memory controller, etc.). The controlling region executes software modules on the MicroBlaze, reads bitstreams from internal or external memory to perform PR via the ICAP, and controls data processing in the PRRs via the *PRSockets* (Section III.A.2).

2) Data Processing Region

The data processing region consists of reconfigurable streaming blocks (RSBs), where each RSB has a designer-specified number of PRRs, I/O modules (IOMs), and a linear array of communication switches, which comprises the SCORES inter-module communication architecture.

Figure 1 shows a sample VAPRES architectural layout with one RSB containing three PRRs and two IOMs, each connected to one of the four SCORES switches *SW*. Each PRR communicates with the MicroBlaze using asynchronous FIFO-based fast simplex links (FSLs) and each IOM allows external I/O pins or peripherals to communicate with the PRRs using SCORES. For each switch-PRR or switch-IOM pair, a *PRSocket* allows the MicroBlaze to control switch, PRR, IOM, and module interface operation using the device control register (DCR) [11] and additional interfacing logic. The DCR contains control bits that are set by the MicroBlaze using the general-purpose input/output (GPIO) peripheral. The MicroBlaze also configures the PRRs' independent clock frequencies (local clock domains (LCDs)) using the *PRSocket* [11].

B. VAPRES System Design

VAPRES system design combines a base system design flow and an application design flow to create a PR SoC [11].

1) Base System Design Flow

The base system design flow leverages designer-designated base system specifications, which includes parameters such as the maximum number of PRRs, communication channel width, number of one-way communication channels between switches, and the number of input and output channels between each PRR and the PRR's switch. After setting the base system specifications, the system designer defines the base system floorplan in a user constraints file (UCF) and creates a VHDL file modeling the static region, the microprocessor hardware specification (MHS) file defining the system structure, and the microprocessor software specification (MSS) file defining the base system build process. Finally, the synthesis and implementation steps manipulate the UCF, VHDL, MHS, and MSS files to generate the base system's static bitstream.

2) Application Flow

The application flow requires the application designer to decompose an application into the hardware and software modules. Hardware modules follow the hardware module design flow, which requires the application designer to write, synthesize, and implement the VHDL hardware modules' (PRMs') code and respective VHDL wrappers (Section IV.A) to generate the PRR partial bitstreams. Software modules follow the software design flow, which requires the application designer to write and compile the software code to generate an executable for the MicroBlaze.

IV. Adaptive Fault Tolerant PR SoC Design

We divide AFT PR SoC design into two design steps: (1) the dataflow controller design step creates an HDL-based finite

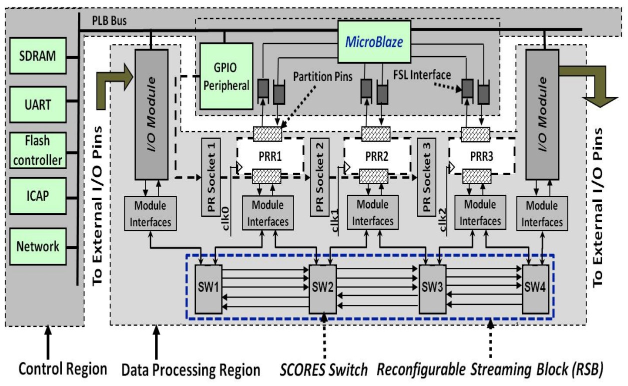

Figure 1: Sample VAPRES architectural layout showing a single reconfigurable streaming block (RSB).

state machine (FSM) to orchestrate the dataflow between the MicroBlaze and the PRRs; and (2) the AFT controller design step creates a C-based AFT controller module that allows the MicroBlaze to adaptively change the reliability mode.

A. Dataflow Controller

In a VAPRES-based SoC, data flows between the MicroBlaze and the PRRs through consumer FSLs (data flowing from the MicroBlaze to the PRRs) and producer FSLs (data flowing from the PRRs to the MicroBlaze) [11]. We designed a dataflow controller to manage this data flow, as VAPRES provides top-level VHDL PRR wrapper templates containing the consumer and producer FSL port declarations only. Our dataflow controller enables high SoC throughput by continuous data stream processing (i.e., the dataflow controller simultaneously processes input and output data). Figure 2 depicts the five-state FSM dataflow controller that orchestrates the continuous stream processing and TABLE I describes the associated state transition signals. Only critical signals are shown and if not specified, a signal is implicitly set to 0.

To allow seamless integration of PRMs into the SoC, the dataflow controller is designed considering FSL communication interfaces (control signals, IO data busses) and to ensure each PRR has a dedicated dataflow controller, we integrated the dataflow controller one level below each VAPRES top-level PRR wrapper. The dataflow controller encapsulates each PRR's PRM and the PRMs' communication interfaces and currently, the input and output data bus sizes are fixed at 32 bits (a VAPRES architectural constraint), however, the dataflow controller's VHDL code can be easily modified to incorporate additional or custom bus sizes.

B. AFT Controller Design

The MicroBlaze allows a C-based AFT controller to easily reconfigure the PRRs using the ICAP and adapt the reliability mode in accordance to changing radiation levels. In this section we describe the AFT controller's operation and algorithm.

1) AFT Controller Operation

The AFT controller offers four reliability modes—low reliability, high reliability, medium reliability, and hybrid

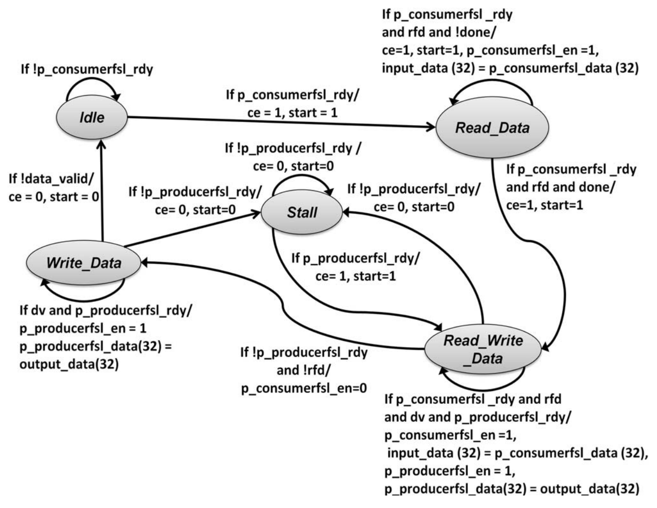
If !p_consumerfsl_rdy
Idle
If p_consumerfsl_rdy/ ce = 1, start = 1
If p_consumerfsl _rdy and rfd and !done/ ce=1, start=1, p_consumerfsl_en =1, input_data (32) = p_consumerfsl_data (32)
Read_Data
If !p_producerfsl_rdy / ce= 0, start=0
If p_consumerfsl _rdy and rfd and done/ ce=1, start=1
If !data_valid/ ce = 0, start = 0
If !p_producerfsl_rdy/ ce= 0, start=0
Stall
If !p_producerfsl_rdy/ ce= 0, start=0
Write_Data
If p_producerfsl_rdy/ ce= 1, start=1
If dv and p_producerfsl_rdy/ p_producerfsl_en = 1 p_producerfsl_data(32) = output_data(32)
If !p_producerfsl_rdy and !rfd/ p_consumerfsl_en=0
Read_Write _Data
If p_consumerfsl _rdy and rfd and dv and p_producerfsl_rdy/ p_consumerfsl_en =1, input_data (32) = p_consumerfsl_data (32), p_producerfsl_en = 1, p_producerfsl_data(32) = output_data(32)

Figure 2: Dataflow controller's five-state FSM. Unspecified signals are set to 0.

TABLE I. DATAFLOW CONTROLLER'S SIGNALS AND ASSOCIATED FUNCTIONS

Signal name and size	Signal type relative to data-flow controller	Function
p_consumerfsl_rdy (1 bit)	In	Indicates valid input data in consumer FSL
p_producerfsl_rdy (1 bit)	In	Indicates producer FSL is ready for data
rfd (1 bit)	In	Indicates PRM is ready for data
done (1 bit)	In	Indicates PRM will produce valid output data in next clock cycle
dv (1 bit)	In	Indicates PRM has produced valid out data
input_data (32 bit)	In	Input data signal to PRM
P_producerfsl_data (32 bit)	In	Input data signal from producer FSL
p_consumerfsl_en (1 bit)	Out	Allows reading data from consumer FSL
ce (1 bit)	Out	Halts PRM in current state (overrides start signal)
start (1 bit)	Out	Starts PRM when asserted
p_producerfsl_en (1 bit)	Out	Allows writing data to producer FSL
output_data (32 bit)	Out	Output data signal from PRM
p_consumerfsl_data (32 bit)	Out	Output data signal to consumer FSL

reliability—which are similar to the three reliability modes introduced by Jacobs et al. [10] (Section II). The AFT controller switches reliability modes depending on the FPGA's current orbital position's SEU rate (models [10] can be used to predict the SEU rate using the CREME96 tool [3]). The SEU rate provides the error probability in the FPGA and governs the active reliability mode. Depending on the active reliability mode, the AFT controller loads single or multiple copies of the required PRMs into individual PRRs. To reduce the power consumption, the unused PRRs' clocks are disabled. The AFT controller also includes a power saving mode to reduce system power consumption and a refresh mode to recover from faults.

a) Low Reliability Mode

The low reliability mode is similar to RFT-HP and loads only a single copy of each required PRM into individual PRRs. Since only one PRR is used per PRM, this mode can achieve high resource utilization and performance relative to other reliability modes because unused PRRs can provide additional processing as opposed to replicated functionality in higher level reliability modes. However, since there is no PRM redundancy, this mode offers no reliability (any necessary reliability must be included by the designer within the PRM) and is only suitable for low SEU rate orbital positions.

b) High Reliability Mode

The high reliability mode is similar to RFT-TMR and loads three copies of each required PRM into three individual PRRs. Since three PRRs are used per PRM, this mode achieves low resource utilization and performance relative to other reliability modes because two additional PRRs are required per PRM and the possibility of leveraging unused PRRs for additional processing is low. However, since there are three copies of each PRM, this mode offers high reliability and is suitable for high (or any level of) SEU rate orbital positions.

c) Medium Reliability Mode

The medium reliability mode is similar to RFT-SCP and offers a balance between the low and high reliability modes by loading two copies of each required PRM into two individual PRRs. Since two PRRs are used per PRM, this mode achieves higher resource utilization and performance than the high reliability mode but lower resource utilization and performance than the low reliability mode. This mode is suitable for medium to low SEU rate orbital positions.

d) Hybrid Reliability Mode

The hybrid reliability mode can be used when some or all PRMs contain internal reliability using methods such as algorithm-based fault tolerance (ABFT)) (without loss of generality, we assume that any PRM with internal reliability uses ABFT) and these PRMs do not require replication. Therefore, PRMs without ABFT are replicated and loaded into individual PRRs depending on the current orbital position requirements (i.e., load two copies in medium radiation orbits, three copies in high radiation orbits or a single copy in low radiation orbits) and PRMs with ABFT are loaded into single PRRs. The resource utilization and performance of this mode can be higher than medium and high reliability modes because unused PRRs can be leveraged for additional processing by PRMs with ABFT. This mode is suitable for any SEU rate orbital positions.

e) Power Saving Mode

In the power saving mode, all PRRs are turned off and the required PRMs run on the MicroBlaze. This mode is only available if software versions of the required PRMs are available. This mode offers higher power savings but lower performance as compared to other reliability modes since there is no hardware acceleration. Since there is no hardware redundancy, this mode is only suitable for low SEU rate orbital positions. However, if the system contains redundant copies of the MicroBlaze, this mode can also be used in medium to high SEU rate orbital positions.

f) Refresh Mode

When the software voter in the AFT controller's algorithm (Section IV.B.2) detects a fault, the refresh mode repairs the faults by scrubbing the affected PRR(s) using the recommended scrubbing commands as described in [24]. This mode also periodically refreshes the entire FPGA using scrubbing commands [24] to prevent SEU accumulation.

2) AFT Controller's Algorithm

Algorithm 1 depicts the AFT controller's algorithm that orchestrates reliability mode switching. The algorithm takes as input the set of all SoCs available PRMs' partial bitstream starting addresses (*PB_Addr*[]) and the partial bitstream sizes (*PB_Size*[]), the corresponding PRMs' functionality names (functionality names must be unique) without ABFT (*PRM_Func*[]) and with ABFT (*ABFT_PRM_Func*[]), the SEU rate threshold values (*Thresholds*[]) for switching between reliability modes, the total number of available PRRs (*Total_PRRs*), and a designer-specified scrubbing time period (*Time_P*) to prevent SEU accumulation.

Lines 1 through 4 verify that the system has been properly loaded using *VAPRES_init*() (line 1) and gather data about the system via the VAPRES IOMs' external pins connected to a ground station communication antenna or data sensors. Line 2 and 3 obtain the current SEU rate (*Cur_SEU_Rate*) and the current input data (*Cur_Input_Data*[]) from the data sensors, respectively. Line 4 obtains the required functionality corresponding to the current input data (*Cur_Req_Func*[]).

Lines 5 through 41 execute an infinite loop that orchestrates the reliability mode switching process. Line 6 sets the system reliability mode using *Call_Mode*(), which takes as input the gathered data *Cur_SEU_Rate* and the algorithm's input data *Thresholds*[], *PRM_Func*[], and *ABFT_PRM_Func*[], and switches (based on the *Set_Mode,* line 7) to the required reliability mode or the power saving mode. *Set_Mode* defaults to case 4 (hybrid reliability mode, line 17), if *ABFT_PRM_Func*[] contains PRM functionality names.

Depending on the required reliability mode, specialized functions perform the necessary architectural changes to complete the reliability mode switching process. In order to make the necessary architectural changes, the functions *Low_R*(), *Med_R*(), and *High_R*() switch to low, medium, and high reliability mode, respectively, and take as input *PRM_Func*[], *Cur_Req_Func*[], *PR_Addr*[], *PB_Size*[], and *Total_PRRs*. All three functions operate using five mode-switching steps: step one compares *PRM_Func*[] and *Cur_Req_Func*[] to identify which PRMs offer the current required functionalities; step two retrieves the corresponding

```
     Input: PB_Addr[], PB_Size[], PRM_Func[], ABFT_PRM_Func[],
                  Thresholds[], Total_PRRs, Time_P
1  VAPRES_init();
2  Cur_SEU_Rate ← SEU_Rate(); //from I/O module
3  Cur_Input_Data[] ← Input_Data();//from I/O module
4  Cur_Req_Func[]←Req_Func();//from I/O module
5  while (1) do
6     Set_Mode ← Call_Mode (Cur_SEU_Rate, Thresholds[],
                    PRM_Func[], ABFT_PRM_Func[]);
7     Switch (Set_Mode)
8        Case 1: //Low Reliability Mode
9           HW_SW_Func_Alloc[]←Low_R(PRM_Func[],
                 Cur_Req_Func[], PR_Addr[], PB_Size[], Total_PRRs);
10          break;
11       Case 2: //Medium Reliability mode
12          HW_SW_Func_Alloc[]←Med_R(PRM_Func[],
                 Cur_Req_Func[], PR_Addr[], PB_Size[], Total_PRRs);
13          break;
14       Case 3: //High Reliability Mode
15          HW_SW_Func_Alloc[]←High_R(PRM_Func[],
                 Cur_Req_Func[], PR_Addr[], PB_Size[], Total_PRRs);
16          break;
17       Case 4: //Hybrid Reliability Mode
18          HW_SW_Func_Alloc[]←HB_R(PRM_func[], Cur_Req_Func[]
               PR_Addr[], PB_Size[], Total_PRRs, Cur_SEU_Rate,
19          ABFT_PRM_Func[], Thresholds[]);
20       Case 5: //Power Saving Mode
21          HW_SW_Func_Alloc[]←Power_Save_Alloc(Cur_Req_Func[]);
22          break;
23       do{
24          Send_Data(HW_SW_Func_Alloc[], Cur_Input_Data[]);
25          Results[]←Rec_Data(HW_SW_Func_Alloc[]);
26          Err_PRR[]←Voter(HW_SW_Func_Alloc[])
27          if (Err_PRR[]) then
28             break;
29          else
30             Send_Data_IOM(Result);// to I/O module
31             Cur_Mode ← Set_Mode;
32             Cur_SEU_Rate ← SEU_Rate();//from I/O module
33             Cur_Input_Data[]←Input_Data();//from I/O module
34             Cur_Req_Func[]←Req_Func();//from I/O module
35       }while (!Ch_Mode(Cur_mode, Cur_SEU_Rate, Thresholds[])
                   and !Tm_Period(Time_P));
36    if (Tm_Period(i)) then
37       Refresh_all(); //scrub entire device
38    else if (Tm_Period(i)==0 and Err_PRR[]) then
39       Refresh(Err_PRR[]); //scrub affected PRR
40       Err_PRR[] ← clear;
41 end while
```

Algorithm 1: The AFT controller's algorithm

PRMs' partial bitstreams from memory using *PR_Addr*[] and *PB_Size*[]; step three loads single or multiple copies, dictated by the reliability mode, of the PRMs' partial bitstreams into the PRRs using the *vapres_mem_load_with_size* [11] application programming interface (API); step four allocates software versions for unloaded PRMs if the required number of PRRs exceeds *Total_PRRs* (we note that this step is only possible if a software version of the PRM is available and requires a priority-based PRM hardware/software allocator, where the ground station sets the PRMs' priorities, and a PRR scheduler—both are future work); and step five records each PRRs' loaded PRM and specifies the PRM functionalities that need to run in software, if any, in *HW_SW_Func_Alloc*[] (if the PRM is not available in software, the PRM is queued in the PRM scheduler to be loaded later into a PRR). The reliability modes allocate PRMs to PRRs as follows: low reliability mode (lines 8 through 10) uses *Low_R*() to allocate one PRR per required PRM; medium reliability mode (lines 11 through 13) uses *Med_R*() to allocate two PRRs per required PRM; high reliability mode (lines 14 through 16) uses *High_R*() to allocate three PRRs per required PRM.

The hybrid reliability mode switching function *HB_R*() (lines 17 through 19) makes the necessary architectural changes using a modified version of the five mode-switching steps and requires three additional inputs: *Cur_SEU_rate*, *ABFT_PRM_Func*[], and *Thresholds*[]. Step one of the original five mode-switching steps is modified to additionally compare *Cur_Req_Func*[] to *ABFT_PRM_Func*[] to identify which PRMs offer ABFT versions of the current required functionality. Then, for ABFT PRMs, *HB_R*() allocates one PRR per required ABFT PRM and for non-ABFT PRMs, *HB_R*() allocates one, two, or three PRRs per required PRM if the current SEU rate *Cur_SEU_rate* is lower than the low to medium switching threshold value, lies between the low to medium switching and medium to high threshold value, or is greater than the medium to high switching threshold value, respectively.

Power saving mode (lines 20 through 22) is set when the *Cur_SEU_Rate*() is less than the power saving mode threshold value in *Thresholds*[] (typically a threshold value indicating a very low or negligible SEU rate). Power saving mode makes the necessary architectural changes using *Power_Save_Alloc*(), which takes as input the current required PRM functionality names *Cur_Req_Func*[], allocates all required PRM functionality to software (if available), and turns off the clock to all PRRs using the MicroBlaze's GPIO.

After the reliability mode switching process completes and all required PRM functionalities are loaded into PRRs or allocated to software, lines 23 through 35 execute a loop to continuously pass input/output data between the IOMs and the PRRs and check the output results for errors. Line 24 uses *Send_Data*(), which takes as input *HW_SW_Func_Alloc*[] and *Cur_Input_Data*[], to send the current input data to the corresponding PRMs' PRRs and/or software functions (input data is sent to the PRRs using the VAPRES *put_fsl*() [11] API). Line 25 uses *Rec_Data*(), which takes as input *HW_SW_Func_Alloc*[], to acquire output data from the corresponding PRMs' PRRs and/or software functions (output data is read from the PRRs using the VAPRES *get_fsl*() [11] API) and writes the output data to *Results*[]. Line 26 uses *Voter*(), which takes as input *HW_SW_Func_Alloc*[], to compare the replicated PRMs' output data *Results*[] to check for errors, and, if an error is detected, writes the PRM functionality names and corresponding PRR locations to *Err_PRR*[]. Line 27 and 28 terminates the loop if there is data in *Err_PRR*[]. Line 30 sends the processed data results to the IOMs for transmission to the ground station using *Send_Data_IOM*(). Line 31 saves the current reliability mode to *Cur_Mode*. Line 32 polls the IOMs for the next SEU rate. Lines 33 and 34 obtain the next set of input data and corresponding required functionality. Finally, line 35 terminates the loop in either of two cases: (1) if a reliability mode switch *Ch_Mode*() is required or (2) a designer-specified time period *Tm_Period*() has passed. *Ch_mode*() takes as input the current operating mode *Cur_Mode*, current SEU rate *Cur_SEU_Rate,* and *Thresholds*[] to check if the current reliability mode is suitable for the current SEU rate by comparing *Cur_SEU_Rate* with the SEU threshold values. *Tm_period*() takes as input the designer-specified scrubbing time period *Time_P* and compares the current time to an internal timer to check if the required time period has expired.

After an error has been detected or *Time_P* time has expired, lines 36 through 40 recover the system from faults. Line 37 prevents SEU accumulation by using *Refresh_all*() to scrub the entire device. Lines 39 through 40 recover the failed PRMs' PRRs using *Refresh*(), which takes as input the failed PRM functionality names and associated PRR locations *Err_PRR*[], and clears *Err_PRR*[] to indicate that the PRMs' functionality has been recovered.

Currently, the entire algorithm is designed as a standalone application and runs sequentially. However, if *Call_Mode*() and *Voter*() were run in parallel with the remainder of the AFT controller's algorithm, the system performance would increase. Parallel execution of *Call_Mode*() and *Voter*() can be done in two ways: (1) if the MicroBlaze runs an operating system, *Call_Mode*() and *Voter*() can be designed as separate software processes (proper data communication synchronization is required) and (2) if hardware versions of *Call_Mode*() and *Voter*() are available, external interrupts can be used when a reliability mode switch is required. In the hardware version, the *Call_Mode*() hardware process would interrupt the AFT controller to execute a reliability mode switch interrupt service routine (ISR) (lines 7 through 22), and the *Voter*() hardware process would interrupt the AFT controller to scrub the faulty PRR(s) using a *refresh*() ISR (line 39).

V. EXPERIMENTAL RESULTS

A. Experimental Setup

We verified the functionality of our AFT PR SoC on a Xilinx Virtex-5 XC5LX110T FPGA) test device [1]. We implemented our AFT PR SoC with only one MicroBlaze processor but since that introduces a single point of failure to the SoC, before deployment we assume the processor is replicated three times using TMR to provide sufficient system reliability. To ensure the SoC's PRRs contained enough resources to load PRMs that performed realistic space application computations, we floorplanned our AFT PR SoC's PRRs constrained to span 40 vertical and 21 horizontal configuration logic blocks, which, given the test device, allowed for a maximum of four PRRs with 1,680 slices each.

To provide a fair evaluation of the resource and power utilization of our AFT PR SoC with respect to a traditional FT PR SoC, we compared the same PR SoC with AFT and without AFT. Both SoCs operated at 100MHz and contained four PRRs, where the AFT PR SoC PRRs adaptively reconfigured FFTs according to the current SEU rate (FFT-AFT PR SoC) and the non-AFT PR SoC PRRs were reconfigured with FFT TMR regardless of the SEU rate (FFT-TMR PR SoC). Without loss of generality, we implemented four different FFT versions, 1-K point, 512 point, 256 point, and 128 point, to serve as different PRM types for evaluation purposes. Since the PR design flow requires a PRR's size to be equal to the largest PRM loaded into the PRR, the smaller FFTs required the same resource amount as the largest 1k-point FFT.

We evaluated the resource utilization for both space SoCs for the ISS case study orbit. The ISS has an orbital period of 92 minutes and operates in a low earth orbit (LEO) with a perigee of 361 km and an apogee of 437 km. We calculated the varying fault rates based on orbital position using the CREME96 tool [3] and Weibull parameters from [15]. Figure 3 shows the estimated ISS orbit SEU rates for the test device for a sample 24 hour period. The largest SEU rate peaks occurred near the South Atlantic Anomaly (SAA) and the smaller peaks occurred near the poles. This 24 hour period showed a good variation in SEU rates, and provided an ideal test input for AFT.

For testing purposes, *SEU_Rate*() was hardcoded with the SEU rates from Figure 3, but actual SEU rates could be obtained during runtime by polling an SEU rate sensor loaded in an IOM. We selected a reasonably low SEU rate threshold of 2.0 SEUs per day for switching between low to medium reliability and a high SEU rate threshold of 8.0 SEUs per day for switching between medium to high reliability. However, these threshold values could be modified by the system designer to reflect application-specific requirements. To determine more accurate reliability mode switching thresholds, the SoC must be either tested in space or tested using fault injection techniques, which is beyond the scope of this paper.

B. *Resource Utilization Results and Analysis*

Both the FFT-AFT PR SoC and FFT-TMR PR SoC required a total of 12,351 slices, which was approximately 71% of the test device. To calculate the resource utilization of each SoC with respect to the current orbit SEU rate, we defined P_{av} as the total number of PRRs available, P_{req} as the number of PRRs required per PRM, P_{used} as the number of PRRs used, P_{free} as the number of free PRRs, P_{usable} as the number of usable free PRRs, and P_{nru} as the normalized PRR resource utilization and leveraged the following three equations:

$$P_{free} = P_{av} - P_{used} \tag{1}$$

$$P_{usable} = \begin{cases} P_{free}\ if(P_{free}\%P_{req} = 0\ and\ \frac{P_{free}}{P_{req}} \geq 1) \\ P_{req}\ \ if(P_{free}\%P_{req} = 1\ and\ \frac{P_{free}}{P_{req}} \geq 1) \\ \quad 0\ (otherwise) \end{cases} \tag{2}$$

$$P_{nru} = \frac{P_{usable}+P_{req}}{P_{av}} \tag{3}$$

For example, in low SEU rate orbital positions the FFT-AFT PR SoC required one PRR per PRM ($P_{req} = 1$), one PRR per PRM was used ($P_{used} = 1$), four total PRRs were available ($P_{av} = 4$), three PRRs were free ($P_{free} = 3$), and three usable PRRs were free ($P_{usable} = 3$), therefore, the normalized resource utilization was 1 ($P_{nru} = 1$). Alternatively, in low SEU rate orbital positions the FFT-TMR PR SoC required one PRR per PRM ($P_{req} = 1$), three PRRs per PRM were used ($P_{used} = 3$), four total PRRs were available ($P_{av} = 4$), one PRR was free ($P_{free} = 1$), and one usable PRR was free ($P_{usable} = 1$), therefore the normalized resource utilization was 0.5, ($P_{nru} = 0.5$). Therefore, in low SEU rates the FFT-AFT SoC achieved a 0.5 higher P_{nru} (i.e., 50% more PRR resource utilization) than the FFT-TMR SoC. Similarly, in medium reliability mode the AFT-FFT PR SoC had 50% more PRR resource utilization than the FFT-TMR SoC (P_{nru}'s of 1 and 0.5, respectively). In high reliability mode, both the AFT-FFT PR SoC and the FFT-TMR had the same PRR resource utilization (P_{nru}'s of 0.75).

Figure 4 depicts the FFT-AFT PR SoC's and the FFT-TMR PR SoC's P_{nru} values calculated with respect to the ISS orbit SEU rates. The results revealed that over the 24 hour period, the FFT-TMR PR SoC achieved average P_{nru}'s of 0.64 (i.e., 64% of the PRR resources were utilized on average), whereas the FFT-AFT PR SoC achieved a significantly higher average P_{nru} of 0.86 (i.e., 86% of the PRR resources were utilized on average).

C. *Power Consumption Results and Analysis*

We measured the power consumption of both the FFT-AFT PR SoC and the FFT-TMR PR SoC using a wattmeter. The FFT-AFT PR SoC had an average base power consumption of 10.8 watts when all PRRs' clocks were disabled, and consumed 11.0 watts when all PRRs were active. The FFT-TMR PR SoC continuously consumed 11.1 watts. Since the wattmeter

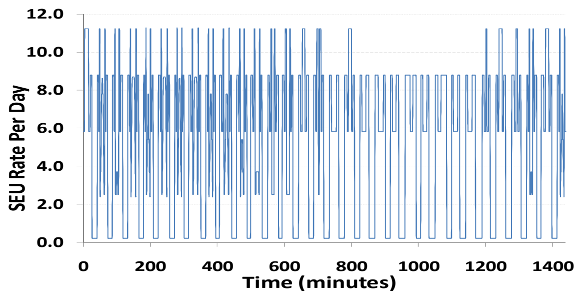

Figure 3: ISS orbit SEU rates over a period of 24 hours for a Virtex-5LX110T FPGA.

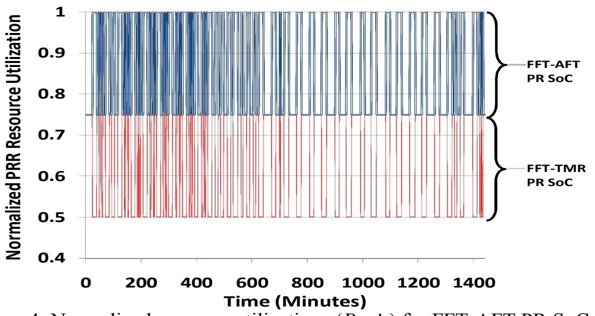

Figure 4: Normalized resource utilizations (P_{nru}'s) for FFT-AFT PR SoC (ranges from 0.75 to 1) and FFT-TMR PR SoC (ranges from 0.5 to 0.75) with respect to the ISS orbit SEU rates.

measured the power consumption of the entire test device, and to reduce the payload as much as possible, space systems typically contain stripped down versions of FPGA devices (e.g., PICO cards [13]), we measured the power consumption of each SoC's FPGA only using Xilinx's Xpower analyzer [25].

We simulated the FFT-AFT PR SoC and the FFT-TMR PR SoC for 0.1 ms and loaded the post place and route trace simulation file to the Xpower analyzer. The FFT-AFT PR SoC had an average base power consumption of 3.45 watts when all PRRs' clocks were disabled, and consumed 3.65 watts when all PRRs were active. The FFT-TMR PR SoC continuously consumed 3.45 watts. Thus, in power saving mode, the FFT-AFT PR SoC can achieve approximately 5.5% power savings in comparison to the FFT-TMR PR SoC.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we designed and implemented an adaptive fault tolerant partially reconfigurable system-on-chip (AFT PR SoC) leveraging VAPRES—the Virtual Architecture for Partially Reconfigurable Embedded Systems. A novel MicroBlaze-based software controller (AFT controller) adapts the AFT PR SoC's fault tolerance to changing space radiation levels and achieves higher resource utilization in comparison to a traditional triple modular redundancy (TMR)-based fault tolerant (FT) PR SoC. Our results indicate the AFT PR SoC can achieve an average of 22% higher resource utilization in the International Space Station (ISS) orbit. Since a system designer can implement a wide variety of applications using the AFT PR SoC's PRRs, the AFT PR SoC is an ideal platform for space SoCs.

Future work includes integrating an operating system in our space SoC to allow parallel software processes to control voting and reliability mode switching, developing a priority based hardware/software scheduler and resource allocator for loading of space SoC components, upgrading the AFT PR SoC's MicroBlaze processor with a LEON3FT fault tolerant processor to provide additional system reliability, using fault injection techniques to test our space SoCs robustness and evaluating the performance penalties of the SoC due to reconfiguration time and refresh mode.

ACKNOWLEDGEMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422.

REFERENCES

[1] C. Bolchini, L. Fossati, D.M. Codinachs, A Miele, C. Sandionigi, "A Reliable Reconfiguration Controller for Fault-Tolerant Embedded Systems on Multi-FPGA Platforms," International Conference on Defect and Fault Tolerance in VLSI Systems (DFT), Oct. 2010.

[2] F. Chen, L. Craymer, J. Deifik, A.J. Fogel, D.S. Katz, A.G. Silliman, R.R. Some, S.A. Upchurch, K. Whisnant, "Demonstration of the remote exploration and experimentation (REE) fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing," International Conference on Dependable Systems and Networks (DSN), June 2000.

[3] CRÈME96 https://creme.isde.vanderbilt.edu/CREME-MC.

[4] A. Doumar, K. Katoh, H. Ito, "Fault Tolerant SoC Architecture Design for JPEG2000 using Partial Reconfigurability," International Conference on Defect and Fault-Tolerance in VLSI Systems (DFT), Sept. 2007.

[5] N. Goel, K. Paul, "Hardware Controlled and Software Independent Fault Tolerant FPGA Architecture," International Conference on Advanced Computing and Communications (ADCOM), Dec. 2007.

[6] S. Di Carlo, P. Prinetto, A. Scionti, "A FPGA-Based Reconfigurable Software Architecture for Highly Dependable Systems," International Conference on Asian Test Symposium (ATS), Nov. 2009.

[7] S. Erdogan, J.L Gersting, T. Shaneyfelt, E.L. Duke, "Using FPGA technology towards the design of an adaptive fault tolerant framework," International Conference on Systems, Man and Cybernetics (SMC), Oct. 2005.

[8] S. Erdogan, T. Shaneyfelf, G. S. Ng; A. Wahab, "Fault Tolerant Hardware for High Performance Signal Processing," Advanced International Conference on Telecommunications (AICT), June 2008.

[9] X. Iturbe, K. Benkrid, T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, A. Perez, "R3TOS: A reliable reconfigurable real-time operating system," International Conference on Adaptive Hardware and Systems (AHS), June 2010.

[10] A. Jacobs, A.D. George, G. Cieslewski, "Reconfigurable fault tolerance: A framework for environmentally adaptive fault mitigation in space," International Conference on Field Programmable Logic and Applications (FPL), Aug. 2009.

[11] A. Jara-Berrocal, A. Gordon-Ross, "VAPRES: A Virtual Architecture for Partially Reconfigurable Embedded Systems," Design, Automation & Test in Europe Conference & Exhibition (DATE), March 2010.

[12] Jet Propulsion Laboratory, "Virtex-4VQ Dynamic and Mitigated Single Event Upset Characterization Summary", April 2009.

[13] Pico Computing Inc., "E-17 Quick Reference", www. picocomputing.com, April 2010.

[14] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin, "Improving FPGA Design Robustness with Partial TMR," International Reliability Physics Symposium (IRPS), March 2006.

[15] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, "Static Proton and Heavy Ion Testing of the Xilinx Virtex-5 Device," Radiation Effects Data Workshop (REDW), July 2007.

[16] G.L. Smith, L. de la Torre, "Techniques to enable FPGA based reconfigurable fault tolerant space computing," International Conference on Aerospace, March 2006.

[17] E. Stott, P. Sedcole, P. Cheung, "Fault tolerant methods for reliability in FPGAs," International Conference on Field Programmable Logic and Applications (FPL), Sept. 2008.

[18] M. Straka, J. Kastil, Z. Kotasek, "Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA," International Conference on Nordic Microelectronics, (NORCHIP), Nov. 2010.

[19] M. Straka, J. Kastil, Z. Kotasek, "Modern fault tolerant architectures based on partial dynamic reconfiguration in FPGAs," International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), April 2010.

[20] I. Troxel, E. Grobelny, G. Cieslewski, J. Curreri, M. Fischer, A.D. George, "Reliable management services for COTS-based space systems and applications," International Conference on Embedded Systems & Applications (ESA), June 2006.

[21] K. Whisnant, R.K. Iyer, Z.T. Kalbarczyk, P.H. Jones, D.A. Rennels, R. Some, "The Effects of an ARMOR-based SIFT environment on the performance and dependability of user applications", International Conference on Software Engineering (ICSE), April 2004.

[22] J. Williams, N. Bergmann, R. Hodson, "A Linux-based Software Platform for the Reconfigurable Scalable Computing Project", International Conference on Military and Aerospace Programmable Logic Devices (MAPLD), September 2005.

[23] J. Williams, A. Dawood, S. Visser, "Reconfigurable Onboard Processing and Real Time Remote Sensing", Transactions on Information and Systems (IEICE), May 2003.

[24] Xilinx inc., "Correcting Single-Event Upsets in Virtex-4 FPGA Configuration Memory", XAPP1088 (v1.0), October 5, 2009

[25] Xilinx inc., "ISE Design Suite Software Manuals and Help", UG681 (v 12.4), Dec. 2010.

[26] Xilinx inc., "ML505/ML506/ML507 Evaluation Platform User Guide", UG347 (v3.1.1), Oct. 2009.