A Programmable Configuration Controller for Fault-Tolerant Applications

Lingkan Gong*†, Tong Wu*, Nguyen T. H. Nguyen*, Dimitris Agiakatsikas*,
Zhuoran Zhao*, Ediz Cetin†, and Oliver Diessel*
* School of Computer Science and Engineering, UNSW Australia
† School of Electrical Engineering and Telecommunications, UNSW Australia

***Abstract*—FPGAs are promising candidates for computational tasks in space applications. However, they are susceptible to radiation-induced errors, the most common failure being due to the corruption of their configuration memory. Module-based partial reconfiguration and frame-based scrubbing are the two most commonly used techniques for detecting and recovering from configuration memory errors. Both methods require user-designed reconfiguration controllers (RC) to read and write FPGA configuration memory data. This paper proposes a Programmable Configuration Controller (PCC) specifically designed for fault-tolerant applications. PCC has a soft Application Specific Instruction Set Processor (ASIP) architecture. The PCC is software programmable using the C language, which allows it to be used in a wide variety of fault-tolerant applications with minimal design and/or hardware overhead. PCC also has instruction extensions to accelerate commonly-used reconfiguration operations such as reading and writing configuration data. Through 5 case studies, we demonstrate that the use of an ASIP architecture for reconfiguration control in applications prone to radiation-induced corruption strikes the right balance between speed, resource utilization and flexibility.**

I. Introduction

Both module-based partial reconfiguration and frame-based scrubbing perform FPGA fault recovery by reading and writing configuration bitstreams from and to the FPGA configuration port, such as the Internal Configuration Access Port (ICAP) in Xilinx FPGAs. The Reconfiguration Controller (RC), which oversees this process, is thus a critical component for FPGA fault recovery. A number of investigations have studied general techniques for designing fast, light-weight, and easy-to-use RCs [1]. For space-based FPGA systems in particular, the requirements for resource utilization (area), speed and flexibility should be motivated yet constrained by the desire to reduce the risks of radiation-induced errors to the RC itself.

Constrained by the system reliability, we believe it is essential to have an RC that balances the tradeoffs between performance, resource utilization and flexibility, instead of focusing on any one single factor while neglecting others. For example, in space applications that use slow flash memories for storing bitstreams, the system only needs to perform reconfiguration at the rate at which the flash memory can be read. Resource utilization is another criterion and a light-weight RC introduces less error-susceptible bits. Removing unused functionality could reduce resource usage but the RC may not remain flexible enough to be reused in a range of applications, or to explore alternative fault recovery algorithms. In particular, the surveys in [2], [3] cite a large number of possible scrubbing algorithms. The design space is further enlarged if we also consider Modular Error Recovery (MER) as a fault recovery strategy. Ideally, the RC should be software programmable so that designers can explore various fault recovery methods or extend existing fault recovery approaches.

To the best of our knowledge, existing RCs only focus on one or two aspects of area, speed and flexibility in general but have not considered them jointly in the context of space applications. HWICAP [4] is a Xilinx, general-purpose IP that provides a bus-based interface to the ICAP and is software programmable to send any command sequence, including single frame R(ead)/W(rite) as well as partial/complete bitstreams. Unfortunately, the software-programmability, while improving flexibility, comes at the cost of large resource overhead and low performance [5]. As another example, the Xilinx Soft Error Mitigation (SEM) controller [6] is a light-weight and high-performance IP dedicated for fault recovery. However, the SEM controller is not flexible since it is based on the PicoBlaze processor [7], which does not have an official C compiler and suffers from an extremely small instruction space (1,024 words). Hence, the controller can not readily be reprogrammed to perform new or different scrubbing functions.

This paper proposes a Programmable Configuration Controller (PCC) to assist in detecting and recovering from radiation-induced errors in space applications. PCC is a soft Application Specific Instruction Set Processor (ASIP) based on the RISC-V instruction specification [8]. It supports all RISC-V integer instructions and benefits from a complete compiler tool chain and a large development community. PCC can run fault detection/recovery software using the general instructions defined by the RISC-V specification while being able to benefit from the high reconfiguration throughput provided by instructions specifically customized for this purpose. The PCC implementation is based on Vscale [9] and PicoRV [10], two versions of RISC-V that have low resource usage. The PCC can be used in either standalone mode or peripheral mode,

This research was supported in part by the Australian Research Council's Linkage (LP140100328) and Discovery (DP150103866) Projects funding schemes.

978-1-5090-5602-6/16 $31.00 © 2016 IEEE

as configured at design time by passing Verilog parameters or VHDL generics, to meet different system design requirements.

This article is organized as follows. Section II summarizes the use cases of RCs for fault-tolerant applications and assesses the suitability of a number of existing RCs for this application. The survey shows that both HW and SW solutions fail to adequately balance performance, resource utilization and flexibility requirements for applications with radiation-induced faults. Aiming to bridge the gap, we propose the ASIP architecture of Section III, while Section IV provides the design details of the Programmable Configuration Controller (PCC). In Section V, we compare the PCC with reference reconfiguration controllers in a comprehensive set of fault-recovery case studies. Section VI presents our conclusions.

II. Related Work

While various reasons could cause an FPGA design to fail in space, the most common failure is due to the corruption of the configuration memory induced by radiation [11]. User and research interest has focused on devising autonomous methods to detect and recover configuration memory errors as they occur. Broadly, the controllers, that oversee such autonomous fault detection and recovery, as well as the golden copy of the recovery bitstreams, may be located either external to the FPGA or within it. Our research focuses on on-chip controllers that fetch externally stored bitstreams and access an internal configuration port to check and overwrite the configuration memory. Hence, reading and writing the configuration memory are fundamental operations for detecting and recovering from configuration errors. For example, in MER, the existence of an error can be identified by reading voter status via the configuration port [12], [13]. After an error is detected, MER approaches typically recover the error by partially reconfiguring the erroneous module [14], [15]. In simple blind scrubbing, the RC refreshes the FPGA device by continuously rewriting the configuration memory [3]. A more efficient scrubbing method relies on Error Correcting Codes (ECC) stored with each configuration memory frame. The RC reads the configuration memory so as to calculate and check the ECC data, which can isolate single bit errors. A corrected configuration frame is written back to the device when an error is identified [2]. Last but not least, fault injection, which is a commonly-used technique for test and debug purposes, is typically implemented by intentionally writing an erroneous frame to the FPGA device [16]. In summary, the RC is a key component for space-based FPGA designs, and there are a variety of different use cases for it.

Various proprietary and academic controllers have been developed to meet the general needs of dynamically reconfiguring FPGAs as well as the specific requirements posed by radiation fault-tolerant applications. As mentioned in Section I, the HWICAP [4] and the SEM controller [6] are vendor IPs that can be used for fault recovery. HWICAP, commonly used with a soft processor such as MicroBlaze, suffers from large resource overheads and slow performance, while the SEM controller fails to meet flexibility needs as it does not allow new scrubbing functions to be developed. Despite these limitations, they are popular due to their ease of use. Several of the use cases we discuss in Section V were originally developed using either HWICAP or the SEM controller.

Several academic development efforts have achieved ICAP throughput approaching the maximum rated capacity of 400 MB/s but invariably compromise on flexibility and/or reliability. Representative efforts include:

- AC-ICAP [17] for Kintex-7, which provides an AXI interface and therefore can be interfaced to a MicroBlaze or user logic, achieves 380 MB/s ICAP throughput using 1286 LUTs, 1193 FFs and 22 BRAMs and supports single frame R/W as well as loading of partial bitstreams, but does not support the loading of arbitrary commands, preventing state capture, for example;
- A self-recovering controller [18], developed for Virtex-4, that has the ability to recover from errors within the controller by loading prestored recovery bitstreams, achieves 380 MB/s throughput, performs single frame R/W and loads partial bitstreams, and supports ECC scrubbing – while fast, this controller's flexibility is compromised by virtue of being PicoBlaze based;
- An open source controller [5], developed for Virtex-6, that can be overclocked to drive the ICAP at up to 838 MB/s using 586 LUTs, 672 FFs and 8 BRAMs, but is inflexible as it only supports loading partial bitstreams;
- Another open source controller [19], also designed for Virtex-6, that only supports loading of protected bitstreams by performing Single Error Correction, Double Error Detection (SECDED) at 320 MB/s or Cyclic Redundancy Checking (CRC) at 395 MB/s using about 590 LUTs, 300 FFs, and 1 BRAM;

A number of researchers, such as [20], have worked on pure hardware-based RCs for better performance but are more expensive to implemente than combined software-hardware approaches. Furthermore, HW-only designs are not programmable and lack the flexibility needed to develop and explore different fault detection and recovery applications.

III. ASIP-based Fault Recovery

As mentioned in Section I, RCs for fault recovery needs to balance performance, resource usage, and flexibility. In this paper, we propose an ASIP architecture to achieve this balance. An ASIP is a processor that has customized instructions for specific applications. In particular, an ASIP has the benefit of a general-purpose processor for users to implement various fault detection and recovery algorithms. It also caters for customized instructions to accelerate reconfiguration operations commonly used for fault-recovery. Based on our literature survey in Section II, we have identified the commonly used reconfiguration operations as follows:

Table I MAPPING OF RECONFIGURATION OPERATIONS TO ASIP

Operations	HW Instruction	SW Function
write configuration read configuration	`wcfg <buf_addr> <nwords>` `rcfg <buf_addr> <nwords>`	– –
write one frame read one frame	– –	prepare header and footer in a buffer; use `wcfg` to write the header and footer; use `wcfg`/`rcfg` to write/read frame data
issue `GCAPTURE` issue `DeSync`	– –	prepare configuration commands in a buffer; use `wcfg` to write commands;
blind scrubbing modular error recovery partial reconfiguration	`wbit <ext_addr> <nwords>`	identify the external bitstream address; use `wbit` to write bitstream from external source;
ECC-based scrubbing	– –	read one frame and check ECC; correct the error and write the corrected frame back;
ICAP-based voter checking	– –	read one frame and extract voter bits;
fault injection	– –	read one frame, flip one bit and write the frame back;

- *Error detection by reading the configuration memory*. As mentioned in Section II, errors can be identified by reading the voter status bits. For Xilinx-based FPGAs, the RC needs to issue a GCAPTURE command so as to copy the voter status from user logic to the configuration memory, and then read the configuration frame where the voter status bits reside. For ECC-based scrubbing, errors can also be detected by reading out the configuration data, calculating the ECC data while reading, and comparing the calculated ECC data with the pre-stored ECC data. For Xilinx-based FPGAs, the RC can, in some cases, obtain ECC results directly from the FRAME_ECC primitive. Therefore, the RC needs to be capable of reading the configuration memory, while being able to perform vendor-specific operations such as issuing GCAPTURE and reading the FRAME_ECC primitive.
- *Error recovery by writing to the configuration memory*. As mentioned in Section II, both MER and scrubbing approaches recover a configuration error by over-writing the configuration memory. The golden configuration data is commonly stored in radiation-hardened storage such as external flash. The golden data can also be calculated at runtime by flipping the erroneous bits of the configuration data. Therefore, the RC needs to be capable of accepting configuration data from external sources, and buffering them for future reference.
- *Error injection by writing the configuration memory*. Fault injection can be used to test the reliability of user-designed components by emulating bit-flip errors in the FPGA's configuration memory. In order to support fault-injection experiments, the controller should be able to read a frame and write it back with some bits flipped.

Table I illustrates the mapping of commonly-used reconfiguration operations to the hardware or the software of an ASIP. The most fundamental operations are reading and writing configuration data to/from the configuration port, a temporary bitstream buffer, and an external bitstream source such as flash memory. We have therefore decided to implement these as customized instructions. All other application-specific operations are performed in software. For example, a `GCAPTURE` command is issued by populating the temporary bitstream buffer with the required command sequence and executing a `wcfg` instruction, which is capable of copying an abitrary number of words from the bitstream buffer to the ICAP. Writing one frame is achieved by three `wcfg` instructions that write the header, frame data and the footer respectively. ECC-based scrubbing is accomplished by reading a frame, checking the ECC and writing the corrected frame back. By mapping these basic operations to customized instructions, the ASIP architecture can meet the performance goals that fault-tolerant applications require. By providing software programmability, an ASIP-based RC can be flexibly reused for a wide range of use cases without consuming additional resources.

IV. PROGRAMMABLE CONFIGURATION CONTROLLER

The Programmable Configuration Controller (PCC) is designed according to the general requirements and design considerations presented in Section III. It currently supports Xilinx FPGAs but its general idea can easily be applied to FPGAs from other vendors.

The PCC can be used in various modes that are configurable at design time by passing Verilog parameters or VHDL generics to the design. Figure 1 depicts the most common configuration of the PCC, the standalone mode. The lightly-shaded blocks are optional modules depending on use cases and modes. The moderately-shaded blocks are present in all configurations. The darkly-shaded blocks are part of the FPGA device silicon (e.g., the ICAP and the FRAME_ECC primitive).

A. RISC-V CPU

Standalone mode is intended to be used in systems where PCC is the only CPU. The main processor of the PCC is

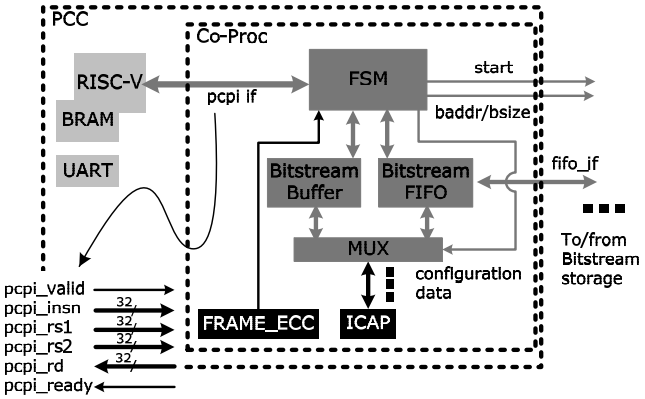

Figure 1. PCC in standalone mode

based on the RISC-V instruction specification and it supports all RISC-V integer instructions and all RISC-V C compilers such as gcc and LLVM [8]. We have excluded the rarely used multiplication/division instructions to reduce resource utilization. The compiler will generate integer instructions to emulate these instructions when they are used. Optionally, the processsor can be configured to include interrupt handling, a UART and a timer. To save resource usage, we implemented these options as customized instructions, which are tightly coupled with the main processor, instead of as traditional peripherals over a shared and resource hungry bus.

The main processor adopts the 3-stage pipeline architecture from Vscale [9], an open-source implementation of the RISC-V specification. To reduce resource utilization, the processor stalls the pipeline when hazards are detected. Furthermore, PCC does not implement any of the privileged instructions and registers that were originally intended to support operating systems. To optimize the design for FPGAs, the instruction and data fetch logic is tightly coupled with the memory resources (e.g. BRAM) available on FPGAs so that instructions/data are always available one cycle after an address is presented to the memory.

B. Extended Instructions for Reconfiguration

The extended instructions, as shown in Figure 2, are implemented by a Finite State Machine (FSM), which controls a bitstream buffer and a bitstream FIFO. The bitstream buffer is used by the `wcfg` and `rcfg` instructions (See Table I) as a temporary storage for configuration data. The buffered configuration data can be used by other operations that are implemented in software, such as to extract voter status bits or to flip the erroneous bit when an ECC check mismatches. The bitstream FIFO is used by the `wbit` instruction to buffer configuration data transferred from external storage such as a serial flash. Functionally, the `wbit` instruction acts as a DMA engine between the FIFO and the ICAP. The instruction extension logic can be conceptually viewed as a co-processor to the main processor while it is actually tightly coupled to the pipeline of the main processor for better performance.

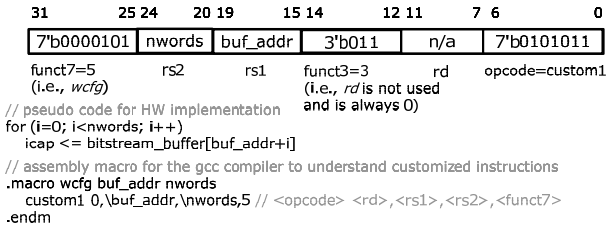

Figure 2. Customized wcfg instruction

Figure 2 illustrates the format and the pseudo-code description of the `wcfg` instruction. The reconfiguration instructions use the R-type instruction format in which the two source operands, rs1 & rs2, come from the register file, and the result, rd, is written back to the register file. The instruction opcode, `custom1`, is selected from the set of reserved opcodes that are used by the RISC-V compiler for user-defined instructions. The `funct3` field indicates whether rs1, rs2 and rd are valid. The `funct7` field is assigned by the user for individual instructions. In the assembly program, users need to define a macro for each extended instruction (see Figure 2). The rest of the program uses the assembly macro and the compiler generates valid instructions accordingly.

The main processor communicates with the configuration co-processor via the Pico Co-Processor Interface (PCPI) [10]. In particular, the decode logic of the main processor dispatches the extended reconfiguration instructions word (i.e., the `pcpi_insn` signal indicated in Figure 1), as well as the operands (i.e., the `pcpi_rs1` and `pcpi_rs2` signals) to the co-processor via a `VALID/READY` handshake. The result (i.e., the `pcpi_rd` signal) is driven by the co-processor and is written back to the main register file.

Standalone mode requires users to design a dedicated flash controller to interface the bitstream FIFO with an external flash. Flash controllers are system specific and are therefore not included as part of the PCC. Instead, PCC exports a simple and native FIFO interface so that users can easily implement logic that transfers configuration data from external storage to the FIFO. Depending on the application's performance requirements, designers can optionally use dedicated DMA to perform bitstream transfer.

C. Peripheral Modes

The PCC can also be used as a slave attachment to the AXI bus and operate as a bus peripheral. Such a system typically involves a main processor, such as a MicroBlaze or a LEON, running the main application, and uses the PCC to perform reconfiguration operations and/or fault-recovery tasks. This mode is also useful during the development of a system during which designers can debug the PCC via the system processor. Depending on whether the PCC includes the RISC-V CPU or not, there are two types of peripheral modes.

Figure 3 depicts the *NO_CPU peripheral mode*. Instead of including the RISC-V processor, the PCC contains the

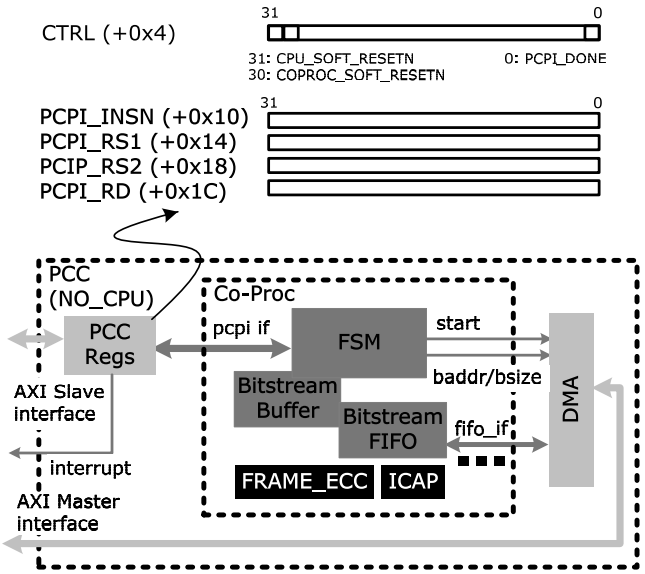

Figure 3. PCC in NO_CPU peripheral mode

PCC_REGs module which has registers accessible by the main processor. The PCC_REGs module mimics the PCPI interface. In particular, if the main processor writes to the PCPI_INSN, PCPI_RS1 and PCPI_RS2 registers, the PCC_REGs module initiates a PCPI handshake and dispatches the instruction to the co-processor. On completion, the PCPI handshake retires and the PCC_REGs module raises an interrupt to the main processor. Since the PCC_REGs module shares the same PCPI interface as the RISC-V, the co-processor of the PCC is the same in all modes. The peripheral mode also includes an AXI master interface to fetch bitstreams from external sources.

If the designer includes the RISC-V CPU, PCC operates in the so-called *co-processor peripheral mode*. The co-processor peripheral mode includes all the AXI master and slave logic, as well as the PCC_REGs module. This mode is intended to be used by systems that require a dedicated CPU to perform fault recovery in the background so as to offload these tasks from the main processor.

V. Case Studies

We demonstrate the benefits and flexibility of the PCC via 5 case studies. The case studies aim to cover all aspects of common reconfiguration operations used by fault-tolerant applications (See Section III). Case studies 1 and 3 demonstrate error detection by reading the configuration frames of the voters and the modules. To recover an error, we can obtain the golden configuration data from external flash (Case Study 2) or by flipping the erroneous bit (Case Study 3). The case studies injects errors as the test stimuli. The fourth study demonstrates the use of PCC to implement a custom recovery strategy that blends frame- and module-based error recovery. The last case study involves using the PCC for traditional partial reconfiguration that is not related to fault recovery, but demonstrates PCC's suitability for general purpose applications. For each case study, we compare the use of PCC with that of a MicroBlaze and HWICAP subsystem (MB_HWICAP) or the SEM controller. We compare the resource usage, performance and development effort for each design.

A. Case 1: ICAP-based Voter Checking

Cast Study 1 demonstrates the use of PCC to check voter status via the ICAP used in Modular Error Recovery (MER) [13] (See Figure 4). The system consists of a number of synthetic compute nodes with each node containing one voter and three identical copies of synthetic computational modules. Each voter monitors the module outputs and updates its voter status bits (either NO_ERROR or the erroneous module ID). The system's Reconfiguration Controller (RC) polls the voter status bits of each voter so as to identify which module is in error, and it does so by reading the configuration frame of the voter. The RC is implemented as either a MB_HWICAP subsystem, or as a PCC running in standalone mode.

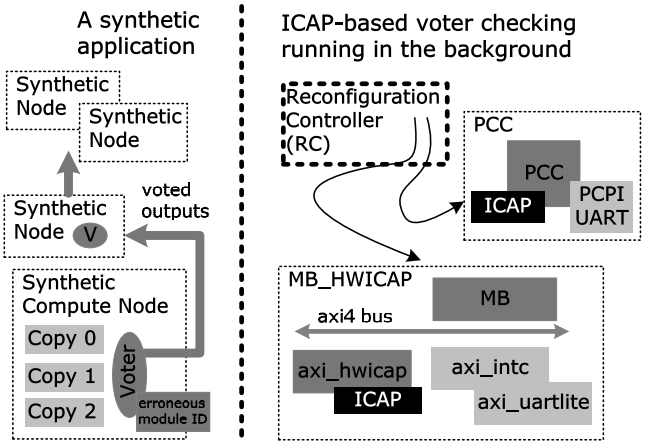

Figure 4. Case study 1: ICAP-based voter checking

Table II Resource utilization of RCs in Case Studies 1-4

RC	Slices	LUTs	FFs
MB_HWICAP (Cases 1,2,4)	3350 (10%)	7452 (5.5%)	8323 (8.1%)
PCC (Cases 1,2,3,4)	573 (1.7%)	1758 (1.3%)	1048 (0.39%)
SEM (Case 3)	187 (0.55%)	568 (0.42%)	478 (0.17%)

Table III Performance of Case Study 1.

RC	Frame Write	Frame Read	GCAPTURE	One Test
MB_HWICAP	73us	70us	5us	16.4ms
PCC	3.41us	3.39us	0.81us	1.51ms
Lower Bound	2.02us	2.02us	0.11us	0.71ms

The voter checking platform was implemented on a Xilinx Artix-7 XC7A200TFBG484-1 FPGA using the Vivado tool chain. A test cycle, which involves injecting an error into one of the compute nodes and checking all voter status bits, requires frequent frame-based configuration access operations. Table II shows the resource utilization of the RCs in Case

Studies 1-4. Table III compares the performance on the frame access between RCs and the theoretical lower bound. Note that for the FPGA used, a frame contains 202 words including the pad words [21], and GCAPTURE requires 11 words. The PCC-based RC achieves more than 20x improvements in performance at significantly lower utilization.

Table III also compares the time to run one full test cycle, which involves 231 frame reads and writes combined. The PCC-based RC was found to be 10x faster than MB_HWICAP. Since the application itself requires frequent frame reads and writes, using fast configuration access instructions has a significant impact on the overall performance. For this voter-checking application, we are only interested in a small number of voter status bits contained in the configuration frame. The PCC uses a `rcfg` instruction to burst the frame data to the bitstream buffer at maximum throughput and only reads the bits of interest to the processor main memory. On the other hand, the MB_HWICAP wastes numerous cycles transferring *all* frame data from the ICAP to the processor main memory over a resource hungry bus, only to use a few bits of interest.

The ASIP architecture of PCC introduces a number of overheads. From a hardware perspective, reading a frame is implemented by two `wcfg` and one `rcfg` instructions (See Table I) and, to simplify the processor design, each customized instruction stalls the pipeline and introduces extra latencies. From a software perspective, a common software coding practice is to encapsulate customized instructions as C functions callable by other C code. Such encapsulation introduces a number of extra instructions. PCC therefore fails to achieve the lowest possible latency. However, in subsequent case studies, we can see that the loss of performance is worthwhile given the extra flexibility provided by software programmability.

B. Case 2: Modular Error Recovery

Case Study 2 evaluates the use of PCC for error recovery by partial reconfiguration of the erroneous modules. It uses the same platform as Case Study 1 but, in order to support modular recovery, Case Study 2 adds extra flash controllers to both RCs (See Figure 5) so as to access the golden bitstreams stored in off-chip flash memory. For the PCC-based RC, we used the Xilinx DataMover DMA to connect the FIFO interface of the PCC to the AXI_QSPI flash controller. On the other hand, the MicroBlaze-based RC adds DMA and a flash controller to the AXI bus outside of the RC.

Table IV PERFORMANCE OF CASE STUDY 2.

RC	QSPI->CPU->ICAP	QSPI->DMA->ICAP
MB_HWICAP	3.56MB/s	11.2MB/s
PCC	–	24.4MB/s
Upper Bound	25MB/s	25MB/s

Table IV compares the throughput of both RCs when used for modular reconfiguration. In this application, the bottleneck of the platform is the off-chip flash memory (Spansion

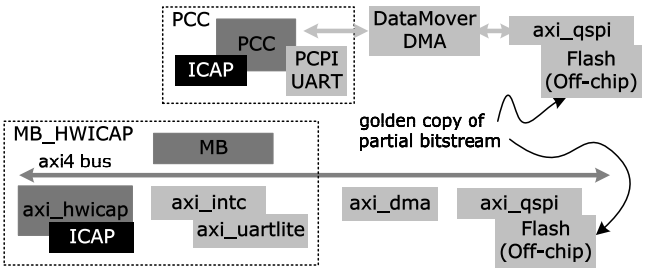

Figure 5. The RCs of Case Study 2

S25FL256S), with a maximum throughput of 25MHz which is dedicated to bitstream storage. Since the PCC exposes the bitstream FIFO as an interface accessible to other components, we are able to directly attach the DataMover DMA and the flash controller to the PCC so as to form a dedicated datapath between the flash memory and the ICAP. The PCC-based RC achieves throughput that is close to the upper bound. For the MB_HWICAP, we have tested the throughput both without and with DMA, and it was no surprise to see a big performance improvement when DMA was used. However, in this application, DMA has to be configured in the "key hole" mode so as to transfer the bistream to the FIFO of the HWICAP. DMA can only transfer 16 bytes at a time and the MicroBlaze-based RC therefore only achieves 50% of the maximum throughput.

C. Case 3: ECC-based Scrubbing

Apart from single frame and modular reconfiguration, PCC can also be used to recover configuration errors by scrubbing. The platform of Case Study 3 is the RUSH on-satellite computer, which is a customized FPGA platform to study fault-tolerance applications in space [22]. The RUSH FPGA design (See Figure 6) contains 9 triplicated computational nodes, as well as an RC that runs ECC-based scrubbing in the background. The RC is either the SEM controller running in repair mode [6], the MB_HWICAP, or the PCC configured in standalone mode. During scrubbing, the RC reads out configuration data frame-by-frame and checks the FRAME_ECC primitives to detect and localize a single bit error. If FRAME_ECC reports an error, the corrected frame is written back to the configuration memory. We added an AXI_GPIO module to the MB_HWICAP to read the FRAME_ECC output. The SEM controller and the PCC are capable of accessing the FRAME_ECC as a sub-component.

Table V PERFORMANCE OF CASE STUDIES 3 AND 4.

RC	Check a Frame	Fix an Error
MB_HWICAP (Cases 3,4)	36us	76us
SEM (Cases 3)	1us	610us
PCC (Cases 3,4)	3.6us	7.5us

The resource utilization of the SEM controller is illustrated in Table II, whereas the resource utilization of the PCC and

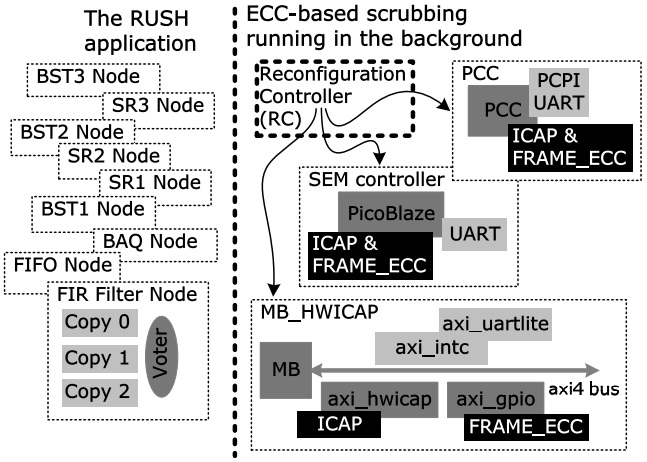

Figure 6. Case study 3: ECC-based scrubbing

the MB_HWICAP is not much different from those of Case Study 1. Table V compares the performance of scrubbing for the RCs. In particular, the SEM controller takes 18.3 ms to scan the entire Artix-7 XC7A200T FPGA [6], which contains 18,300 configuration frames (i.e., 1us per frame). Frame reading is slower for PCC (3.6us per frame) since the customized instructions have to read one extra pad frame for each configuration frame needed. This doubles the number of frames read by the PCC and increases the length of one scrub cycle. On the other hand, it is much faster for PCC to fix an error (7.5us) compared with the SEM controller (610us). Since the SEM controller is a black box, we are not able to analyze its exact operations. For the same reason, we are also unable to extend the SEM IP to study user-developed scrubbing algorithms such as the one in Case Study 4.

D. Case 4: In-house Fault-Recovery Algorithm

Case Study 4 demonstrates the application of PCC to implement and test custom error recovery algorithms such as the Frame- and Module-based configuration memory Error Recovery (FMER) [23]. FMER recovers configuration memory errors that occur in TMR modules using modular reconfiguration. All other resources are periodically scrubbed, thereby ensuring robust detection and repair of errors that occur outside the TMR modules.

The FMER system is implemented on the RUSH FPGA platform and the development effort for this case study involved reusing and integrating hardware and software components from Case Studies 2 and 3. In particular, we developed a tool to extract a list of frames that are not included in TMR modules, and the RC use ECC-based scrubbing, as in Case Study 3, to scrub them on a continuous basis. We added a *star* network so as to connect all voters and to transfer the IDs of erroneous modules from the voters to the RC. When an interrupt is raised by a voter to indicate an error in a TMR module, the RC pauses scrubbing and reconfigures the erroneous module as in Case Study 2. Since the RUSH board uses a parallel flash for storing the golden bitstreams, we modified the RCs of Case Study 2 to use the AXI_EMC flash controller. While these modifications are straightforward to implement on either the PCC or the MB_HWICAP, we are unable to implement FMER using the SEM controller since it cannot be extended.

We found the scrub performance to be the same as Case Study 3 (see Table V). We found the reconfiguration throughput for both RCs to be approximately the same: the PCC had a mean throughput of 13.7MB/s, whereas the MB_HWICAP achieved 12.4MB/s. The performance of both solutions was bounded by the throughput of the parallel flash (15.4MB/s).

E. Case 5: Partial Reconfiguration

Apart from using the PCC for fault-tolerant applications, it can also be used as the RC for general purpose reconfigurable designs. Case Study 5 is derived from the Xilinx partial reconfiguration reference design [24]. As illustrated in Figure 7, the system has a reconfigurable peripheral attached to a microprocessor bus, while two modules, an adder and a multiplier, are mapped to the reconfigurable peripheral. Software running on the main MicroBlaze processor reconfigures the peripheral by programming the HWICAP. The reference design was originally implemented using Xilinx PlanAhead tool targeting Virtex-5 FPGAs with an SD card. We re-targetted it to a Nexys-Video board that has an Artix-7 FPGA and an SPI flash, and upgraded the IPs and tools to Vivado.

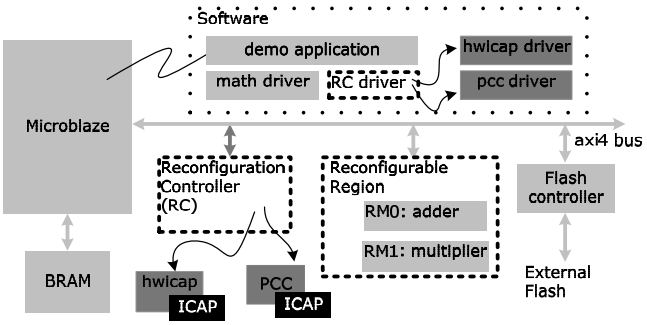

Figure 7. Case study 5: Partial reconfiguration using PCC

Table VI RESOURCE AND PERFORMANCE OF CASE STUDY 5

RC	Slices	LUTs	FFs
HWICAP	261 (0.78%)	339 (0.25%)	957 (0.36%)
PCC	387 (1.15%)	1015 (0.75%)	1022 (0.38%)
PCC.NO_DMA	231 (0.73%)	646 (0.43%)	552 (0.20%)

For Case Study 5, we replaced the HWICAP module with a PCC operating in the NO_CPU peripheral mode. Since this mode uses the standard AXI master and slave interfaces, it can readily be integrated with a blocked design using the Vivado tools. From a software perspective, the application can be easily modified for PCC since the PCC and the `hwicap` drivers share similar C function prototypes and calling conventions. For example, the C functions that perform partial

reconfiguration are the `XHwIcap_DeviceWrite` function for `hwicap`, and the `Pcc_DeviceWrite` function for the PCC. Both versions take the same bitstream buffer and size as arguments. While the `XHwIcap_DeviceWrite` function copies the bitstream, word by word, to the HWICAP, the `Pcc_DeviceWrite` function initiates DMA-style bitstream transfer by dispatching a `wbit` instruction to the PCC co-processor via the PCPI interface. The `hwicap` driver has 3000 Lines of Code (LOC) whereas the PCC version has 1500 LOC.

The PCC, when configured in NO_CPU peripheral mode, consumes 50% more slices than HWICAP (See Table VI). This increase is primarily due to the inclusion of DMA logic and the AXI master interface, which are not available in HWICAP. As a reference, the PCC without DMA is slightly smaller than HWICAP. With DMA, the time to reconfigure either of the peripherals, whose partial bistreams have 123,280 bytes, reduced from 33.8ms, for HWICAP, to 7.6ms for PCC.

VI. Conclusions and Future Work

Due to the unique characteristics of fault-tolerant FPGA applications used in space, the design of the RC needs to carefully balance performance, resource usage, and flexibility requirements. To the best of our knowledge, this paper is the first work to use a soft ASIP architecture to balance such requirements while targetting reconfiguration control for FPGA design reliability. The PCC, an implementation of a soft ASIP architecture, is C programmable and supports rapid development of a variety of fault-tolerant applications. PCC also has hardware instruction extensions to accelerate commonly-used reconfiguration operations such as reading and writing configuration data.

Through 5 comprehensive case studies, we have demonstrated that PCC can be used in both single frame and modular reconfiguration applications (Cases 1 & 2), in conventional and custom scrubbing applications (Cases 3 & 4), as well as in conventional partial reconfiguration applications (Case 5). The PCC can be used in standalone configuration (Cases 1, 2, 3 & 4), and NO_CPU peripheral mode (Case 5). It is capable of interfacing with an external flash controller (Cases 2 & 4), without a flash (Cases 1 & 3) and of using a system flash (Case 5). In all cases, the development effort involves changing a few hardware parameters, and writing C code. Therefore, we believe that PCC meets the flexibility requirements set by space applications. Our case studies also demonstrate that a PCC-based RC provides lower reconfiguration latency, higher throughput and uses less resources compared with the MB_HWICAP solution. While PCC uses more resource than the SEM controller, it achieves comparable performance and provides far more programming flexibility.

This paper provides the baseline design of a reliable reconfiguration controller. Looking ahead, we are studying techniques to enhance the reliability of PCC itself. We are looking at general fault-tolerant approaches such as TMR, ECC-protected BRAM, as well as methods that are specific for ASIP designs.

References

[1] K. Papadimitriou, A. Dallas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 36:1 – 36:24, 2011.

[2] I. Herrera-Alzu and M. Lopez-Vallejo, "Design Techniques for Xilinx Virtex FPGA Configuration Memory Scrubbers," *IEEE Trans. on Nuclear Science*, vol. 60, no. 1, pp. 376–385, 2013.

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

[4] Xilinx, *AXI HWICAP (PG134)*, 2015.

[5] K. Vipin and S. A. Fahmy, "A High Speed Open Source Controller for FPGA Partial Reconfiguration," in *Field-Programmable Technology (FPT), International Conference on*.

[6] Xilinx, *SoftSoft Error Mitigation Controller (PG036)*, 2015.

[7] ——, *PicoBlaze 8-bit Embedded Microcontroller User Guide (UG129)*, 2011.

[8] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, 2014. [Online]. Available: https://riscv.org/specifications/

[9] A. Magyar, *RISC-V in Verilog*, 2015. [Online]. Available: https://riscv.org/2015/09/risc-v-in-verilog/

[10] C. Wolf, *PicoRV32 - A Size-Optimized RISC-V CPU*, 2016. [Online]. Available: https://github.com/cliffordwolf/picorv32/

[11] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, and R. Bell, "On-Orbit Results for the Xilinx Virtex-4 FPGA," in *2012 IEEE Radiation Effects Data Workshop*, 2012, pp. 1–8.

[12] F. Veljkovic, T. Riesgo, and E. de la Torre, "Adaptive Reconfigurable Voting for Enhanced Reliability in Medium-grained Fault Tolerant Architectures," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1–8.

[13] D. Agiakatsikas, N. T. H. Nguyen, Z. Zhao, T. Wu, E. Cetin, O. Diessel, and L. Gong, "Reconfiguration Control Networks for TMR Systems with Module-based Recovery," in *Field-Programmable Custom Computing Machines (FCCM), IEEE Symposium on*, 2016, pp. 88–91.

[14] C. Bolchini, A. Miele, and M. D. Santambrogio, "TMR and Partial Dynamic Reconfiguration to Mitigate SEU Faults in FPGAs," in *IEEE Int. Symp. on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2007, pp. 87–95.

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

[16] H. Quinn and M. Wirthlin, "Validation Techniques for Fault Emulation of SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 62, no. 4, pp. 1487–1500, 2015.

[17] L. A. Cardona and C. Ferrer, "AC ICAP: A Flexible High Speed ICAP Controller."

[18] A. Ebrahim, K. Benkrid, X. Itrube, and C. Hong, "A Novel High-Performance Fault Tolerant ICAP Controller," in *NASA/ESA Conference on Adaptive Hardware and Systems*.

[19] S. D. Carlo, P. Prinetto, and P. Trotta, "A Portable Open-Source Controller for Safe Dynamic Partial Reconfiguration on Xilinx FPGAs," in *Field Programmable Logic and Applications (FPL), International Conference on*.

[20] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2005, pp. 223 – 228.

[21] *7 Series FPGAs Configuration User Guide (UG470)*, Xilinx Inc., 2013.

[22] E. Cetin, O. Diessel, L. Gong, and V. Lai, "Towards Bounded Error Recovery Time in FPGA-based TMR Circuits using Dynamic Partial Reconfiguration," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2013, pp. 1–4.

[23] D. Agiakatsikas, E. Cetin, and O. Diessel, "FMER: A Hybrid Configuration Memory Error Recovery Scheme for Highly Teliable FPGA SoCs," in *Field Programmable Logic and Applications (FPL), International Conference on*, 2016, pp. 1–4.

[24] *PlanAhead Software Tutorial: Partial Reconfiguration of a Processor Peripheral (UG744)*, Xilinx Inc., 2009.