IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs

DOI:
10.1109/MCSoC2018.2018.00018

Document Version
Accepted author manuscript

IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs

***Abstract*—FPGA devices have been used widely in many industrial domains, but only limitedly in secure and safety-critical applications, which have special requirements for the physical implementation, such as module isolation. This is partly due to limited functionality available with current FPGA vendors' tools and flows. To extend FPGA's appearance in secure and safety-critical applications, we propose an alternative flow for isolation design called the *Isolated Partial Reconfiguration Design Flow (IPRDF)* in this paper. Systems designed by the proposed *IPRDF* are not only fully isolated but also support partial reconfiguration of insulated modules. This allows building secure and dependable systems that can use partial reconfiguration to mitigate from single-event upsets (SEUs) and that are more tolerant to aging and device imperfections. Further, this also allows information assurance applications to benefit from hardware module isolation and run-time reconfigurability. Case studies on isolated Triple Modular Redundancy (TMR) and single-chip cryptographic (SCC) designs are presented to demonstrate capabilities and advantages of the proposed *IPRDF* methodology.**

I. Introduction

Field Programmable Gate Array (FPGA) devices can be used not only to substantially accelerate applications, but also are able to achieve these performance advantages with fairly low power overhead [1]. Moreover, with the reconfigurable capability and short time-to-market, FPGAs are replacing ASICs as promising solutions in many industrial domains.

Furthermore, there are strong needs to use FPGAs in other application domains such as in automotive, aerospace, defense, cyber-security and in hazardous environments (e.g., in space [2] or high-energy physics [3]) which often require secure and safety-critical system implementations. However, these applications are challenging to implement with FPGAs. For instance, when FPGAs are working in space, an ionized particle hit can cause a single-event upset (SEU), resulting in a bit-flip in one or more configuration cells or a communication signal and consequently in catastrophic behavior [4]. Moreover, in information assurance applications, hardware modules must be independently and solitarily implemented in a single chip to satisfy isolation, reliability and security concerns [5].

Finally, with the trend of using FPGAs as accelerators in cloud environments [6], [7], physical insulation of reconfigurable modules may become important for granting multiple users simultaneous access to the same FPGA device. For example, recent researches [8], [9] have demonstrated how side channel attacks using delays on long wires can allow leaking information from FPGA cryptographic modules *without physical access*. However, a physical isolation fence between wires of different modules can reduce side-channel effects significantly [8], and therefore can eliminate the risk of data leakage through this sort of attack.

Xilinx, a major FPGA vendor, addresses these concerns by providing an Isolation Design Flow (IDF) [5], which comprises the following requirements:

- Each hardware module has to be isolated and must be in its own level of hierarchy.
- A fence must be used to separate isolated hardware modules within a single chip. Further, it is not allowed to use any primitive or routing resources in any directly adjacent tile (e.g., a CLB or BRAM). Therefore, a fence is at least one tile wide.
- Input/output buffers (IOBs) must be instantiated *inside* isolated modules for proper isolation of the IOBs. Thus, having full control over the routing of the signals from IOBs to the module is essential to establish off-chip trusted communications.
- On-chip communication between isolated hardware modules is achieved through the use of Trusted Routing, which has to follow restrictions:
  - There is no entry or exit point in the fence between isolated regions.
  - There is one source and one destination for each routing path (only joint-to-joint routing).
  - The entry and the exit points must stay in the source and destination regions, respectively.
  - Its entirety stays contained in the source/destination regions
  - It does not touch a fence tile from another isolation region.

Xilinx IDF supports the implementation of isolated modules which satisfy all mentioned requirements as well as an automatic mechanism to verify module isolation using the Xilinx Vivado Isolation Verifier (VIV). However, even the latest Vivado tool suite (version 2018.1) does not provide any partial reconfiguration (PR) capability together with IDF.

Although isolation design is not possible with the Xilinx Partial Reconfiguration (PR) flow as clocks and IOBs must remain in the static logic part [10] when using 7-Series and earlier devices, the redundancy needed for safety-critical systems and the requirements in the physical implementation of these systems would strongly benefit from partial reconfiguration (PR). For example, the fences around modules or the redundant instances of modules for TMR put more pressure on resource utilization which can be in some cases mitigated with the help of PR. Furthermore, cryptographic military systems often use multiple different cyphers (e.g., DES, AES, Blowfish) in a round-robin fashion which can be implemented in a resource efficient manner using PR. Moreover, PR is an effective countermeasure against single-event upsets (SEUs) by writing correct configuration to the memory swiftly. Therefore, by integrating PR flow into IDF, we are able not only to mask a SEU, but also to recover the malfunctioned area at run-time.

To implement this, we propose an alternative flow, named

TABLE I: Isolation Design Flows' features and supports.

Features	Xilinx IDF [10]	Altera SDF [11]	Academic Customized IDF [12], [13]	*IPRDF*
FPGA families	Xilinx	Altera	Xilinx	Xilinx
Isolated modules	✓	✓	✓	✓
Fences between modules	✓	✓	✓	✓
IOB assignments	automatic	automatic	?	manual
Secured communication	✓	✓	✓	✓
Partial reconfiguration	✗	✗	✓	✓
Support for module relocation, multiple module instantiation	✗	✗	✓	✓
Direct communication between PR modules	✗	✗	✗	✓
Off-chip trusted communications in PR regions	✗	✗	✗	✓

Isolated Partial Reconfiguration Design Flow (IPRDF), which applies partial reconfiguration (PR) design practices into the isolation design flow (IDF). Moreover, we provide a design rule check to ensure that our approach fulfills all requirements for a standard isolation design flow as specified in [5]. In addition to Xilinx IDF, systems designed by our flow are partially reconfigurable. This enables to design self-aware and fault-tolerant mechanisms, which gives systems higher probability to detect and mask errors by moving relocatable modules to other regions for alleviating defects in the FPGA fabric. Thus, systems designed by the proposed *IPRDF* use less resources (i.e. are cheaper), are potentially less vulnerable, more reliable, and more suitable in secure and safety-critical applications than static counterparts.

IPRDF includes several phases, which are realized by a combination of commercial Xilinx Vivado [14], open source tools such as GoAhead [15] and BitMan [16] as well as our own tools and scripts adding the isolation capability to reconfigurable systems. In detail, placement and routing for static and partial designs are constrained by GoAhead, and are physically implemented by Vivado. Moreover, full bitstreams are generated by Vivado, but are later manipulated by BitMan to compose relocatable partial bitstreams.

Additionally, we present two case studies: 1) an isolated Triple Modular Redundancy (TMR) system and 2) a single-chip cryptographic (SCC) design both on a ZedBoard using a Xilinx *XC7Z020* FPGA to demonstrate capabilities of the proposed *IPRDF* tool flow.

The contributions of this work include:

- A design flow which guarantees isolated modules, trusted communication channels and separated IOBs for partial reconfigurable modules (Section III).
- Trusted regions which are partially reconfigurable and able to host multiple relocatable modules in time and space manners (Section III).
- Error detection and recovering techniques with considering isolation as required for TMR (Section IV).
- Two case studies on *IPRDF* (Section IV and Section V).

Further sections include an overview on related work in Section II and a conclusion in Section VI.

II. Related Work

A. Isolation Design Flows

The major FPGA vendors have introduced proprietary isolation flows such as Xilinx Isolation Design Flow (IDF) [5], and Altera Separation Design Flow (SDF) [11]. However, as mentioned in Section I, they lack partial reconfiguration capability.

Related research papers [12], [13] used the Xilinx IDF to design relocatable modules. Those works, however, are not targeting design isolation as needed to implement secure or reliable systems. Instead, some IDF mechanisms were used to prevent static system routing to cross partial regions. This property allowed those approaches to relocate modules. However, none of those related approaches could support off-chip trusted communications for partial regions.

Comparisons of the proposed *IPRDF* and other state-of-the-art tool flows are presented in Table I. As we can realize, the limitation of *IPRDF* is that IOB assignment has to be carried out manually[1].

B. Partial Reconfiguration Tools

There are several partial reconfiguration (PR) design flows, available both from industry such as Xilinx [10], Altera [17] and from academia, for instance, Torc [18], RapidSmith [19], OpenPR [20], and GoAhead [15]. OpenPR and GoAhead can generate blocker macros that allow to prohibit the Xilinx vendor router to use a defined set of wires only (e.g., this allows it to implement an IDF conform fence around a module). When physically implementing a module, blocker macros will occupy all possible connections to and from modules. However, we include tunnels into these blockers to carry out the top-level routing. The here presented *IPRDF* is a frontend for the GoAhead tool because it is the only academic PR tool that is currently supporting latest FPGAs from Xilinx.

C. Designing for Reliability

Wirthlin in [4] summarized common design practices for high reliable FPGA systems including hardware redundancy, configuration scrubbing, error-correction coding, flip-flop mitigation, and system-level mitigation of FPGA single-event effects (SEEs). That work highlights the importance of combinations of hardware redundancy, especially Triple-Modular Redundancy (TMR), and configuration scrubbing as a recovering mechanism in systems being used in satellites [2]. Moreover, Abramovici et al. proposed the idea of rotating functional units via PR for testing and repairing in [21]. In Section IV, we will demonstrate a combination of TMR, isolation design and partial reconfiguration that can act as a recovering scheme for highly available and secure systems.

[1]However, this process is relatively easy to carry out and is normally done before the PCB design (which implies that designing secure systems include aspects beyond the actual FPGA design).

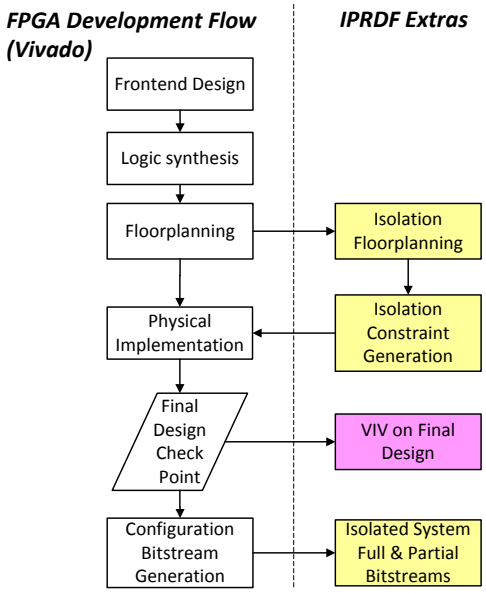


Fig. 1: Isolated Partial Reconfiguration Design Flow *(IPRDF)*.

III. The *IPRDF* Flow

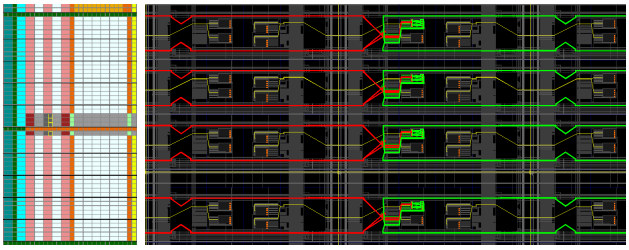
In this section, *IPRDF* is presented step-by-step. The flow is described in Subsection III-A. Static and Partial designs are presented in Subsection III-B and Subsection III-C.

A. Overview

We are using standard Xilinx Vivado for our front-end design and logic synthesis. This allows us to take advantages from all input specification methods that are available in Vivado including RTL, schematic entry, or even High Level Synthesis (HLS). *IPRDF* requires to carry out more steps in the floorplanning stage which will consecutively affect physical implementation as well as bitstream generation. The flow in comparison with the default Xilinx Vivado flow is presented as in Figure 1 from a developer's point of view.

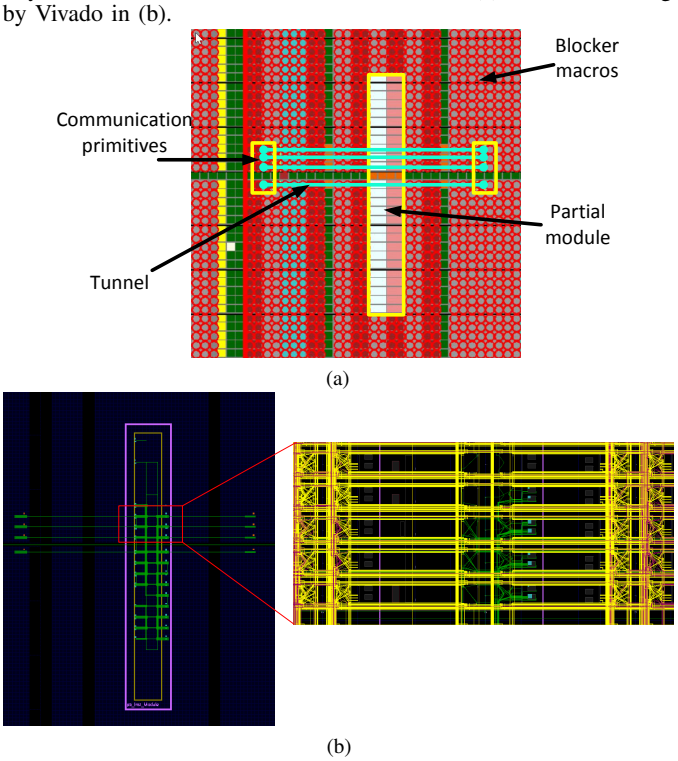
IPRDF's steps are described as follows:

- Isolation Floorplanning: based on the resource utilization retrieved from synthesis reports, we define regions on the FPGA, either to host reconfigurable modules (in the static system) or to implement reconfigurable modules (partial modules). In module floorplanning, an automatic placement exploration [22] starts finding all possible positions. Bounding boxes according to all those positions are generated without user's operation. Static system's floorplanning is done manually but assisted by our tool flow by an automatic check that ensures that those regions provide the necessary number of resources, even if some resources are not allowed to be used due to module isolation.
- Isolation Constraint Generation: position and bounding box information is then used for static and partial designs. Physical constraints for placement and routing are generated by the GoAhead tool. These constraints are written into some TCL files, which are then used by Vivado to guide the physical implementation stage. *IPRDF* adds rules to this process to match the IDF requirements (see Section I). This is implemented through scripts that are written for GoAhead.
- Final Design Check Point: Design Check Points from previous steps will be ran through the Xilinx Vivado Isolation Verifier (VIV) to ensure that our designs comply with the isolation design rules.
- Configuration Bitstream Generation: *IPRDF* results in full bitstreams of static and module designs. To compose partial bitstreams for modules, we use the tool BitMan [16].



(a) (b)

Fig. 2: An example of 16-bit bus for system communication with Physical Constraint Generation on GoAhead in (a) and after routing by Vivado in (b).



(a)

(b)

Fig. 3: Module placement, communication tunnels and blockers for the selected partial module, on GoAhead in (a) and after routing by Vivado in (b).

B. Static Design

The implementation of the static design starts with a floorplanning step where we define the placement of the static system components, communication infrastructure and reserved areas for the partial modules. With these parameters, we instruct GoAhead to create top-level routing and placement constraints as TCL scripts for Vivado.

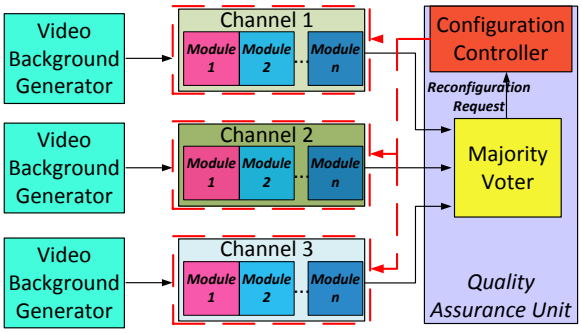


Fig. 4: Block diagram of the TMR system.

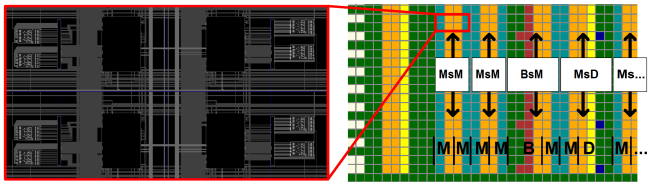


Fig. 5: Partial slots with various FPGA primitive slots such as *MsM*, *BsM* and *MsD*. The left screenshot is from the Vivado and the right screenshot is taken from the GoAhead's floorplanning GUI.

The goal at this stage is to define a region which hosts partial modules. In order to leave as many resources as possible for the actual application, we aim at maximizing this area. In the case of using *IPRDF* for TMR systems, we define 3 regions of identical size (and to be more precise, regions where the relative layout of primitive columns (e.g., CLBs, BRAMs) is identical[2]). A blocker macro is then generated to prohibit any routing or logic resource to be used by reconfigurable modules.

The blocker macro will prevent all FPGA primitives and routing resources to be used in the selected partial region. Therefore, we will leave holes in the blocker macro (called tunnels) that are used to constrain module interface signals to specific wire resources on the FPGA following isolation rules. See Figure 2 for a 16-bit bus example.

C. Module Design

The implementation of the partial module design also starts with a floorplanning step which includes placing communication primitives around the partial module. These macros act as sink/source connection points and substitute the surrounding static system. Blockers will be generated to prohibit all partial module's primitives being placed around the selected area acting as a fence to implement strict module bounding boxes as well as the isolation fence. Routing tunnels are included for the communication to and from the primitives. The position of these tunnels match exactly the tunnels as used in the static design to implement the communication between static and partial areas.

The result of this stage is shown in Figure 3. As we can see, blockers are placed around the partial module to ensure all primitives must be used inside the selected area.

[2]It is not mandatory that all TMR regions are identical because it is possible to generate different module implementations for each region. However, by using identical regions, we could even share the place and route result including the final partial bitstream.

As a modules is implemented on a separated design from the static system, the final results generated by Vivado is a full configuration bitstream. This data is passed to BitMan that cuts out the configuration data that corresponds to the module only. We repeat these steps for all modules to build a partial module library.

At run-time, BitMan manipulates those partial bitstreams to a desired position inside a partial region of the static system.

IV. Case Study I: Triple Modular Redundancy

In this example, a TMR system, as in Figure 4, will be designed on a ZedBoard to demonstrate our *IPRDF*'s capability. This system includes 3 video background generators, 3 video streaming channels, and a *Quality Assurance Unit*, which contains a *Majority Voter* and a *Configuration Controller*. All these components are implemented isolated from each other by physical fences.

Partial regions are tiled into multiple adjacent slots that are two resource columns (CLB, BRAM, or DSP) wide. Modules are one or more slots wide. Implemented modules include a video overlay generator, a DES encryption, and a SHA1 hash function. A module could have multiple alternatives as it might be placed on different primitives (e.g., one alternative providing a RAM column in the left and another providing RAM in the right half of the module). For TMR operation, all channels must host the same modules in the same order to guarantee fully redundant execution over all channels.

Outputs from video streaming channels will be routed to the *Majority Voter* inside the *Quality Assurance Unit*. This *Majority Voter* will guarantee that any SEU in any channel would not impact the final system's output. Moreover, a single difference in a channel's output will trigger the *Configuration Controller* to dynamically partially reconfigure this specific channel to mitigate any SEU effect with a low guaranteed latency. The *Configuration Controller* can stay either on-chip by utilizing reconfigurable ports such as *ICAP* and *PCAP*, or off-chip in a host machine and using *JTAG* port to reconfigure the FPGA fabric[3]. This redundant system with repair mitigates against multiple upsets which may occur and potentially impact the TMR outputs.

Specific information of this example's implementation is provided in Subsection IV-A. Subsection IV-B describes how the error detection and recovering schemes work. In Subsection IV-C, the fully implemented system is presented, and achieved goals for this isolated design are discussed.

A. System Implementation

FPGA resources on the *XC7Z020* device are aligned column-wise. We represent the relative layout of primitive columns by a *Resources String* which is simply a string of symbols which are denoting the particular column type. For example, starting from the bottom left corner, we can model the FPGA with the device resource string *M M M M B M M D M...*, as shown in Figure 5, with:

- *L*: a CLB column providing *SLICE_L* primitives (supports only logic and arithmetic).

[3]In this case study, we are using the *PCAP* port that is controlled by the available ARM core only. For full fault tolerant operation, it would need another port (e.g., *JTAG*) to remove a single point of failure or a watchdog mechanism.

TABLE II: Available primitives on various resource slots and required elements for different modules.

Resource Slot	Region Size		Available Resources		
	Columns	Rows	LUTs	BRAMs	DSPs
MsM	2	46	736	0	0
BsM	2	46	368	8	0
MsD	2	46	368	0	16
Module	**Module Size**		**Required Resources**		
	Columns	Rows	LUTs	BRAMs	DSPs
Video Overlay Generator	2	46	207	0	0
2-round DES in CRC mode	2	46	226	0	0
2-Stage SHA1	2	46	235	0	0
3-Stage SHA1	2	46	295	0	0

- *M*: a CLB column providing *SLICE_M* primitives (supports logic, arithmetic and distributed memory).
- *B*: a column providing *BRAM* primitives.
- *D*: a columns providing *DSP48* primitives.
- *s*: two switch matrices between primitives (any *L M B* or *D* type).

A resource slot (our smallest atomically reconfigurable unit) is defined by a primitive column followed by two switch matrices and another primitive column. To incorporate this in our string matching abstraction, we model our resource slots with symbols like *MsM* or *BsM* or *MsD* to indicate the different combinations of primitives that may exist for a resource slot, as illustrated in Figure 5. A partial module with the type *MsM* can only be loaded in *MsM*-slots. Therefore a logical module may need different implementations (e.g., *MsM*, *BsM*, or *MsD*-compatible versions) if relocation is used. In this case study, each channel provides 6 resource slots.

To establish horizontal fences, we have reserved one CLB row at the top and another at the bottom of a resource slot. This action results in less available primitives within each resource slot to host a module, and this is an expected overhead when using module insulation.

Numbers of FPGA primitives in various slots and the required resources for different functional modules are listed in Table II.

As long as sufficient resources are found in the bounding box, horizontally physical fences at the top and bottom of the module can be established without much effort. However, routing of communication tunnels must be carefully analyzed and strictly constrained to a predefined set of wires that will carry out the top-level module communication through the fence. A physical fence following IDF rules [5] must leave all logical and routing primitives unused.

This means that at least one resource column must be left totally empty at each side of the module's borders. Therefore, when creating the communication infrastructure, we will use *double*-wires, which span a distance of 2 resource columns, or *quad*-wires, which jump from the current column to another one that is 4 columns away.

This ensures that we can bypass the switch matrices of the fence while still implementing all top-level signals. When using *double*-wires, this allows us to route up to four bit signals per CLB row and respectively 3×4 bit signals in the case of *quad*-wires. Considering the usable $50 - 2$ CLB rows, this allows wide interfaces of up to $(50 - 2) \times (4 + 3 \times 4) = 768$ bits in total per signal direction for a module that is one clock region in height on a Xilinx Zynq FPGA.

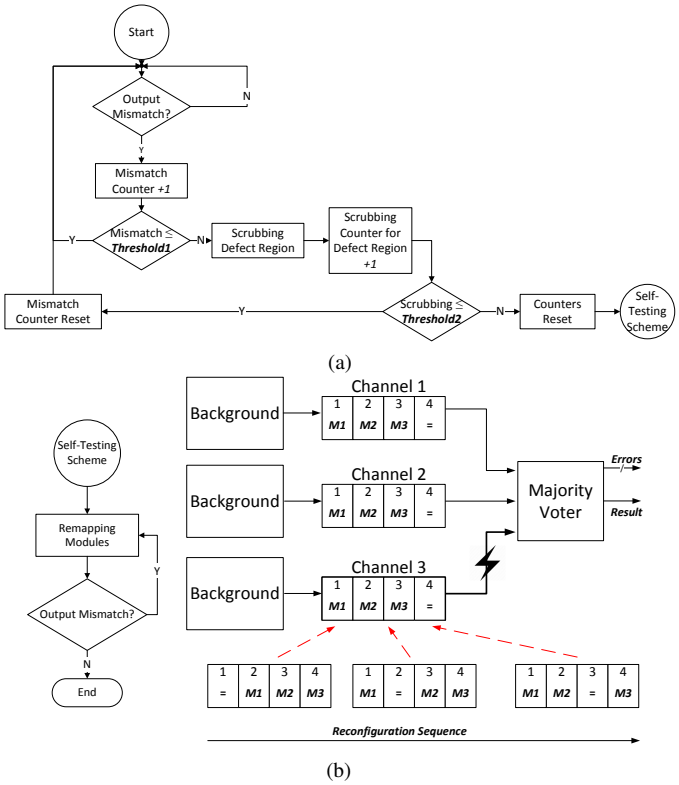


Fig. 6: Two different error detection and recovering schemes. (a) is to prevent impacts from single-event upsets (SEUs), and (b) is to reduce impacts from permanently physical damages such as ageing or device imperfections.

B. Error Detection and Recovering Schemes

At run-time, errors can be caused by SEUs, or physical ageing and are mitigated differently. Therefore, we have developed two schemes to mask them, as shown in Figure 6. In this work, we used the *Majority Voter* as an error detection unit. It will detect any mismatch happening over all channels' outputs, specify the malfunctioning channel, and send requests to the *Configuration Controller* for reconfiguration actions. These recovering schemes are implemented as a software application on the embedded ARM CPU, and PCAP is used for FPGA reconfiguration.

1) SEU Recovering Scheme: In the first flow, as shown in Figure 6a, any difference in a channel's output is detected by the *Majority Voter*, and the error causing channel is recognized. A *Mismatch* counter will be incremented. This allows distinguishing between SEUs on the datapath (transient errors) from SEUs hitting configuration SRAM cells. Former issues can be mitigated by the majority result from the *Voter* while later ones must be repaired by reconfiguring the modules of the impacted channel.

However, permanent FPGA defects cannot be recovered by configuration scrubbing regardless of reconfiguration efforts. Consequently, the *Scrubbing* counter for the defect channel

will be incremented until it reaches a threshold and triggers the second flow, called *Self-Testing Scheme*.

2) Self-Testing Scheme: The example in Figure 6b shows three channels for TMR each hosting three modules that are one slot wide. Each channel provides one spare slot that is bypassed (symbolized with '='). For *Self-Testing*, we generated a reconfiguration sequence such that the first configuration bypasses the first slot, the second configuration bypasses the second slot, etc., until we find a working sequence that is eventually recovering the defective slot.

Self-Testing is carried out for a single channel (e.g., channel 3 in Figure 6b) while using the other channels as a reference for testing correct operation. The actual configurations are composed by BitMan from relocatable modules which in some cases may involve implementation alternatives to deal with the heterogeneous layout of resource columns. Note that in this case study, each channel has 6 slots, and *Self-Testing* is being conducted in parallel to the operation of the system.

In general, the modules in the channels may have an internal state that would be out of synchronization after partial reconfiguration and it needs somehow a mechanism to resynchronize all TMR instances. In this case study working on a video stream, after each row of pixels (in our case, 1024 pixels), all modules start with the same initial state. We therefore wait after reconfiguration for at least this time before evaluating the *Majority Voter* output.

C. Result

Details of an implemented module designed by *IPRDF* are shown as in Figure 7. The generated blocker macros ensure that all module's primitives are placed inside the bounding box. Ultimately, no routing violation was found by the Vivado Isolation Verifier (VIV) when using our design methodology.

Moreover, with our design methodology, a module can be implemented in different resource slots. Thus, it is more flexible to place and relocate modules across the FPGA fabric in order to mitigate physical vulnerabilities that may happen during its lifetime. Figure 7 shows implemented alternatives of the *Video Overlay Generator* in three resource slots providing different resource footprints. These three implemented alternatives are sufficient to place this two column-wide module to any slot inside any reconfigurable region of the system.

The whole demonstration system is designed with *IPRDF* as shown in Figure 8. The static parts include the *Video Background Generator* and the *Quality Assurance Unit* which are physically isolated from other partial regions and from each other. In addition, partial channels are also separated by horizontal and vertical fences between system elements.

For recovering schemes, we tested the correctness of transient fault masking by injecting errors into channels' inputs via push buttons. Moreover, the mitigation technique from SEU in configuration data was verified by flipping random bits from a LUT table through partial reconfiguration.

Finally, to generate permanent errors, we used a feature from BitMan that prohibits a defined resource (a LUT for our experiments) from reconfiguration. Considering modules that are one-slot wide and a channel which has s slots hosting m modules, the worst case time for the *Self-Testing* procedure is:

$$\binom{s}{m} \times (t_{config} \times s + t_{test}),$$

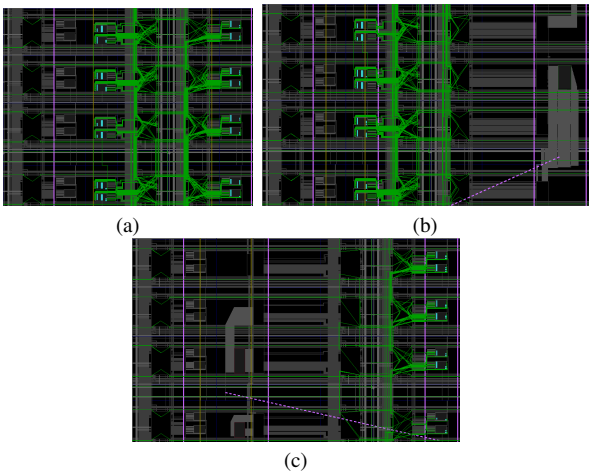


Fig. 7: Various FPGA primitive columns on the *XC7Z020* FPGA device in (a), and implemented options of a *Video Overlay Generator* module, with the *MsM*-compatible option in (b), and the *MsD*-compatible option in (c).

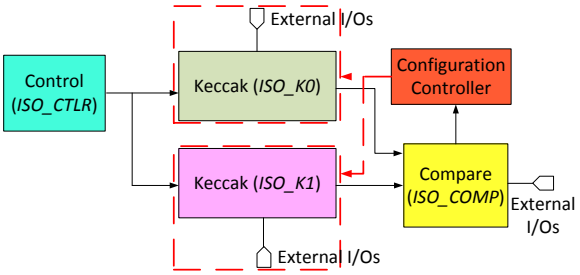


Fig. 9: The single-chip cryptographic (SCC) system's block diagram.

with t_{config} being the time to reconfigure one slot and t_{test} be the time for testing one configuration of the test sequence.

In this case study, reconfiguration overhead per slot was 0.4 ms on average, and the testing scheme took about 0.02 ms per configuration. The worst case was 14.5 ms, where resource verification needed to be conducted for 6 slots per channel.

The whole error mitigation is running in bare-metal on the ARM using PCAP partial reconfiguration.

V. Case Study II: Single-chip Cryptographic Design

To directly compare our *IPRDF* against the Xilinx IDF, we took the example of the Single-chip Cryptographic (SCC) design from the Xilinx's XAPP1256 application note for IDF [23], and we implemented its modules not only isolated but also partially reconfigurable by using our *IPRDF* methodology. The isolation of cryptographic modules satisfies information assurance requirements while the partial reconfiguration enables hardware module replacement or operation maintenance at run-time.

The example design consists of two redundant Keccak cryptographic hash modules[4] (*ISO_K0* and *ISO_K1*), whose outputs are sent to a comparator (*ISO_Compare*) block, and a processor control (*ISO_Controller*) module is used to supply clocks and resets, as shown in Figure 9. This case study is utilizing Double Module Redundancy (DMR) technique to

[4]Keccak is the superset of the SHA-3 standard.

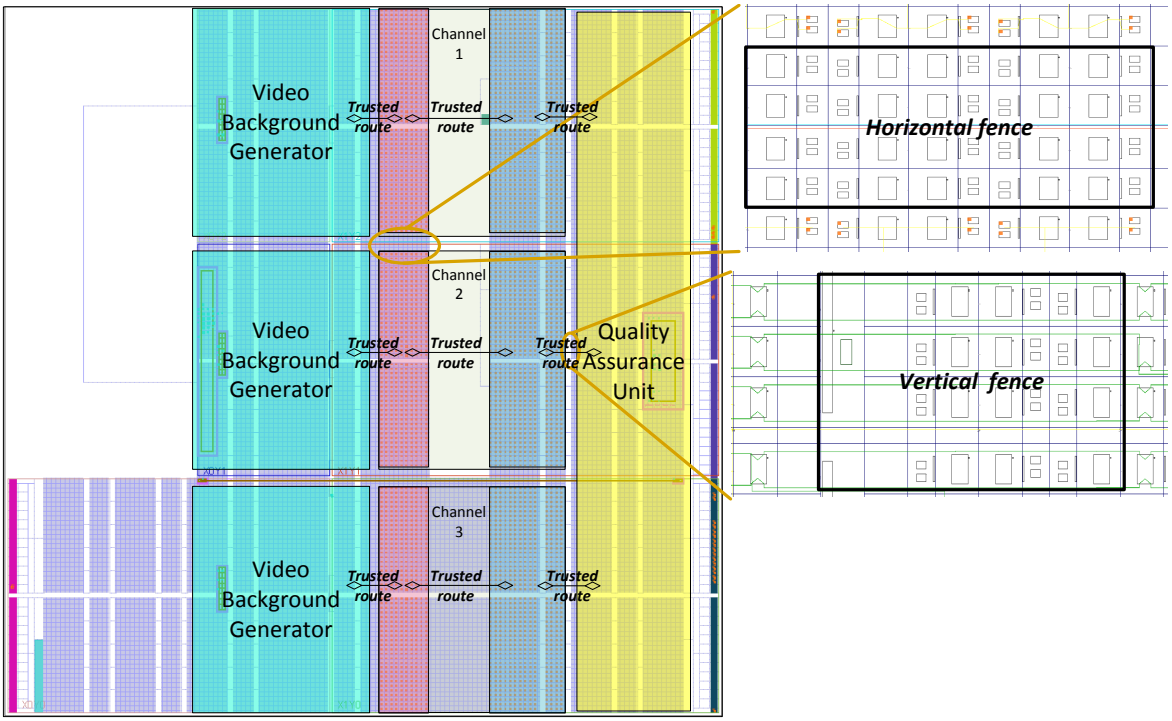


Fig. 8: System layout of the TMR design implemented on a *XC7Z020* FPGA. Each channel, which has 6 resource slots, can host the modules from Table II, as long as their resource primitives match the targeted resources. There are no wires in the horizontal fence between the isolated regions and only trusted routes are crossing the vertical fence.

guarantee system's functional correctness. Moreover, as the main difference to the Case Study I in Section IV, it requires that module IOBs stay inside partially isolated partitions for off-chip communication, which is officially not supported for 7-Series devices according to the latest Xilinx PR documentation [10] at the time of writing this work.

In addition to XAPP1256, we have developed SHA-2 and AES-based hash modules as alternative solutions in order to demonstrate PR on this case study. These modules can be loaded to change cryptographic algorithms without shutting down the whole system.

A. System Implementation

We have revised the Floorplan for the Xilinx IDF reference to reserve two partially reconfigurable regions for Keccak cryptographic modules. One module, *ISO_K0*, is placed in the top-right of the chip layout whereas the other, *ISO_K1*, is at the bottom-left corner. The *ISO_Controller* and *ISO_Compare* stay in the static part of the system as shown in Figure 10. All IOBs for module off-chip communication are reserved inside the isolation partitions, either static or partially reconfigurable, and are connected directly to the modules for full control over the routing of the signals from IOBs to the module.

B. Result

The final design is shown in Figure 10, and physical fences are realized by our *IPRDF* methodology. Moreover, trusted routing is used between isolated partitions for secured communication.

Off-chip inputs and outputs for each Keccak hash module are instantiated and assigned into its isolated partition for trusted communication requirements. Moreover, they are reconfigurable along with the partial modules, which is not feasible with Xilinx IDF and PR flow. This partially reconfigurable capability allows isolated partitions to host different cryptographic algorithms at run-time. Table III shows available resources of each partial region and sizes of their partial bitstreams.

TABLE III: Available resources in *ISO_K0* and *ISO_K1* partial regions and size of partial bitstream to reconfigure each region.

Partial Region	Slice LUTs	BRAMs	DSPs	IOBs	Bitstream Size (kBytes)
ISO_K0	13600	30	60	50	1375
ISO_K1	13248	31	42	50	1425

The outputs of modules in *ISO_K0* and *ISO_K1* are passed to the *ISO_Compare* module for quality assurance purpose. Any difference in these outputs triggers an alarm.

This final design was verified successfully for isolation-compatibility by the Xilinx Vivado Isolation Verifier (VIV). In addition, the demonstration of this partially reconfigurable SCC design could be seen on a YouTube video by following this link: REMOVED FOR BLIND REVIEW!

VI. Conclusion

In this work, we have proposed an alternative design flow, named *IPRDF*, to build fully isolated and reconfigurable systems. Two case studies have been presented to demonstrate details on how to use *IPRDF* for implementing fully isolated designs. The first case study uses this to implement a safety-critical TMR system that provides mitigation strategies for transient faults, configuration SEUs as well as for permanent

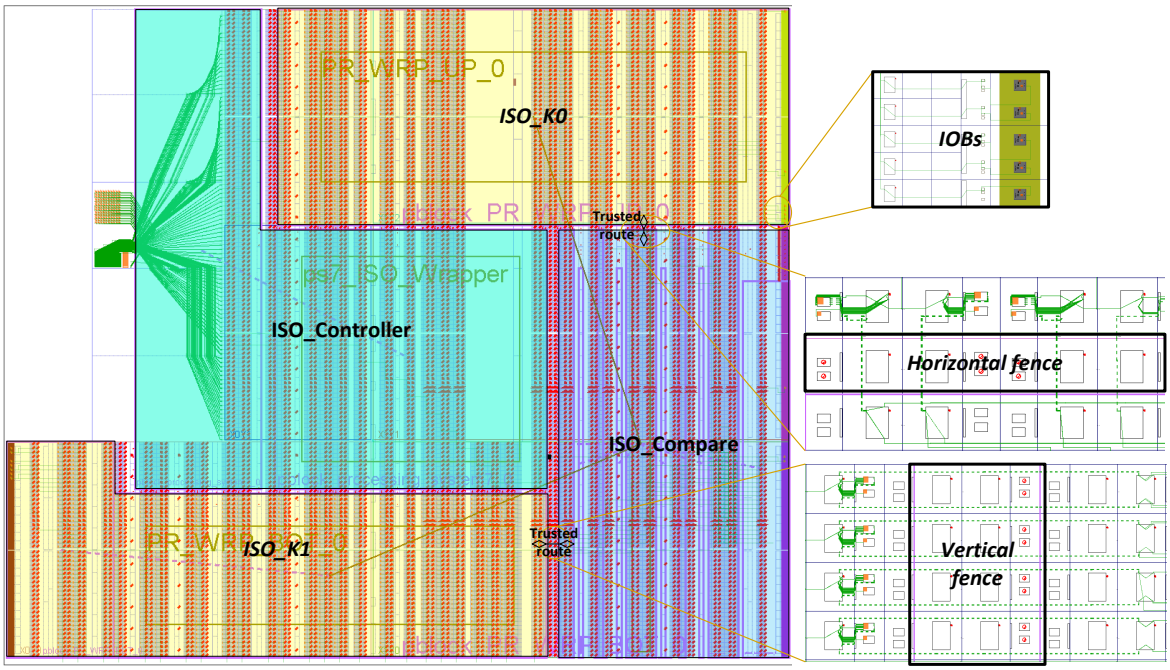


Fig. 10: System layout of the SCC design implemented on the *XC7Z020* FPGA. There are 2 partially reconfigurable regions *ISO_K0* and *ISO_K1* which could host Keccak hash modules, as in this example following the Xilinx's XAPP1256 [23], or other cryptographic ones at run-time.

FPGA defects using partial reconfiguration. The second case study enhances a single-chip cryptographic (SCC) system from [23] with partial reconfiguration capabilities which would allow changing ciphers at low resource cost.

It should be mentioned that module insulation is a requirement for implementing modules in certain security concerned systems (e.g., military applications) and that our entire tool flow is generating physical constraints for the Xilinx vendor tools. As a consequence, no IP details (neither code nor netlist) has to be presented to our tool flow, because our physical implementation scripts can be generated independently, even before the application. This means that the *IPRDF* is not adding any security threat to the already established insulation flow for static only systems.

In practice, the security may be even higher as different modules are developed and physically implemented entirely separated from each other. With this, we enabled partial reconfiguration in secure and safety-critical systems.

References

[1] S. Choi et al., "Energy-efficient Signal Processing Using FPGAs," in *FPGA*, 2003.

[2] H. Quinn et al., "The Cibola Flight Experiment," *TRETS*, 2015.

[3] K. Red, "Single Event Upsets in SRAM FPGA based readout electronics for the Time Projection Chamber in the ALICE experiment," Ph.D. dissertation, The University of Bergen, Bergen, Norway, 2009.

[4] M. Wirthlin, "High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond," *Proceedings of the IEEE*, 2015.

[5] Xilinx, "XAPP1222 - Isolation Design Flow for Xilinx 7 Series FPGAs or Zynq-7000 AP SoCs (Vivado Tools)," 2016.

[6] S. Byma et al., "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *FCCM*, 2014.

[7] Xilinx, "Reconfigurable Acceleration in the Cloud," 2017. [Online]. Available: https://www.xilinx.com/products/design-tools/cloud-based-acceleration.html#alibaba

[8] I. Giechaskiel et al., "Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires," in *ASIACCS*, 2018.

[9] C. Ramesh et al., "FPGA Side Channel Attacks without Physical Access," in *FCCM*, 2018.

[10] Xilinx, "UG909 - Vivado Design Suite User Guide Partial Reconfiguration," Apr. 2018.

[11] Altera, "AN 567: Quartus II Design Separation Flow," 2009.

[12] L. Gantel et al., "Module relocation in Heterogeneous Reconfigurable Systems-on-Chip using the Xilinx Isolation Design Flow," in *ReConFig*, 2012.

[13] J. Rettkowski et al., "RePaBit: Automated Generation of Relocatable Partial Bitstreams for Xilinx Zynq FPGAs," in *ReConFig*, 2016.

[14] Xilinx, "UG908 - Vivado Design Suite User Guide: Programming and Debugging," 2017.

[15] C. Beckhoff et al., "GoAhead: A Partial Reconfiguration Framework," in *FCCM*, 2012.

[16] K. D. Pham et al., "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*, 2017.

[17] Altera, "Partial Reconfiguration," 2017. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/partial-reconfiguration.html

[18] N. Steiner et al., "Torc: Towards an Open-source Tool Flow," in *FPGA*, 2011.

[19] C. Lavin et al., "RAPIDSMITH - A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs," Brigham Young University, Tech. Rep., 2014.

[20] A. A. Sohanghpurwala et al., "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *IPDPSW*, 2011.

[21] M. Abramovici and C. E. Stroud, "BIST-Based Delay-Fault Testing in FPGAs," *Journal of Electronic Testing*, 2003.

[22] N. B. Grigore and D. Koch, "Placing Partially Reconfigurable Stream Processing Applications on FPGAs," in *FPL*, 2015.

[23] Xilinx, "XAPP1256 - Zynq-7000 AP SoCs or 7 Series FPGAs Isolation Design Flow Lab (Vivado Design Suite)," 2016.