

Multi-bit Fault Injection for FPGAs with SPFI

Grzegorz G. Cieslewski, Adam Jacobs, Alan D. George, and Ann Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC),

ECE Department, University of Florida

SRAM-based FPGA devices are an attractive option for on-board data processing on space systems due to higher computational capabilities and a lower power envelope than traditional processing devices. However, FPGAs present unique reliability verification challenges because single-event upsets (SEUs) can trigger both data errors and configuration memory errors, which may cause deviations from the expected system functionality. To evaluate system reliability, traditional fault-injection testing uses numerous test vectors to observe effects induced by a change of a single configuration memory bit at a time. This single-bit fault-injection (SBFI) methodology provides high fault coverage and testing repeatability, but requires long fault-injection time due to the large number of test vectors required to verify the entire design's functionality. Long injection times are further exacerbated as configuration memory size and design complexity increase. To shorten injection time, we propose our Simple, Portable Fault Injector platform for FPGAs (SPFI-FPGA) used in tandem with a novel multi-bit fault-injection (MBFI) methodology which modifies multiple configuration memory bits, referred to as a *batch*, during each test. We optimize our proposed methodology to efficiently detect the faulty bits within a batch, adaptively select the optimal batch size, and optimize the fault-injection sequence based on the design's placement. Using relevant case studies, SPFI-FPGA augmented with MBFI methodology reveals up to $49.7\times$ speedup in fault-injection time.

I. Introduction

State-of-the-art space systems employ new sensor technologies that collect more data than can be transmitted back to the base station. The bottleneck is caused by limited transmitter power and bandwidth, target visibility, and high transmission latency. On-board data processing can mitigate these limitations by performing more data processing on the system before transmitting the smaller, processed results. Since traditional CPU-based technologies often cannot meet the increasing high-performance computing demands for space systems, reconfigurable computing, dominated by field-programmable gate arrays (FPGAs), provides an attractive and powerful alternative, offering advantages in performance, energy efficiency, size, and adaptability. However, since FPGAs store system functionality in configuration memory, they present unique reliability and fault-testing challenges with respect to single-event upsets (SEUs).

In order to mitigate SEUs, space systems typically employ radiation-hardened devices. However, the radiation-hardening process results in smaller, slower, and more expensive devices than commercial-off-the-shelf (COTS) counterparts. Even though COTS devices provide unprecedented levels of flexibility, they are susceptible to SEUs caused by high-energy particles, which can cause data corruption and device latchup. Unfortunately, SRAM-based FPGAs are particularly susceptible to SEUs as compared to non-reconfigurable devices because SEUs can additionally cause configuration memory errors, which can result in deviations from the expected system functionality. In order to leverage the benefits afforded by FPGAs and maintain high system reliability, traditional and innovative fault-tolerant (FT) design methodologies are required as well as an effective fault-testing methodology.

Fault testing is the process of evaluating system or device reliability. Previous work proposes numerous fault-testing methodologies, ranging from simulation and fault injection to physical radiation testing, but no single method has provided an optimal solution with respect to fault-testing speed and fault coverage. The predominant injection methodology for non-reconfigurable devices modifies device memory elements, such as cache and registers, and observes the resulting changes in device functionality. FPGA fault injection is similar in basic principle, but since the configuration memory stores the system's functionality, the configuration memory accounts for the majority [1]

of all susceptible bits, and should therefore be the focus of FPGA fault-testing.

Traditional injection methodologies use single-bit fault injection (SBFI), where a single configuration memory bit is altered during each test. However, since injection time increases as the configuration memory size increases, FPGA injection time is becoming prohibitively long, making comprehensive fault-injection testing nearly impossible. Alternative methods decrease injection time using statistical sampling to select a subset of configuration memory bits to test, as opposed to the entire configuration memory, and use confidence intervals to show bounds on the susceptibility estimate for the entire device. However, an accurate susceptibility prediction with a tight confidence interval requires a large number of samples, which may still result in long injection times.

In this paper, we propose a new Simple, Portable, Fault Injector platform for FPGAs (SPFI-FPGA) in conjunction with a novel multi-bit fault-injection (MBFI) methodology, which significantly decreases injection time as compared to traditional reliability verification techniques. SPFI-FPGA performs fault injection using a mix of full reconfiguration (FR) and partial reconfiguration (PR) that minimizes the time required to modify configuration memory and improves fault-injection speed. Our MBFI methodology mitigates the effects of increasing configuration memory size and large test-vector sets by modifying multiple bits, called a *batch*, in configuration memory during each test. We optimize our methodology to efficiently detect faulty bits within a batch, adaptively select the optimal batch size, and optimize the fault-injection sequence based on the design’s placement. SPFI-FPGA currently supports the Xilinx Virtex-4, Virtex-5 and Virtex-6 FPGA families, however the methodologies proposed in this paper are applicable to a wide range of FPGA devices. Using relevant case studies, our SPFI-FPGA platform augmented with MBFI methodology reveals up to $49.7\times$ speedup in injection time as compared to SBFI.

The remaining sections of this paper are organized as follows. Section 2 surveys previous work related to fault injection and testing. Section 3 provides an overview of SPFI-FPGA, SBFI and testing methodologies. Section 4 details our proposed MBFI methodology and optimizations to decrease the injection time. In Section 5, we evaluate testing performance (injection time) of SPFI-FPGA augmented with our MBFI methodology. Finally, Section 6 provides conclusions and outlines directions for future work.

II. Background and Related Research

SRAM-based FPGAs are commonly used in space systems. FPGAs improve performance and reduce component overhead for many applications as compared to non-reconfigurable space-qualified parts. In addition, an FPGA’s reconfigurable architecture facilitates quick, on-site design changes, which alleviates the design-time impacts of late-stage design bugs and system functionality additions, and provides the ability to repurpose system functionality after deployment. In this section, we first briefly review FPGA architectural fundamentals and reconfiguration methods, which form a basis for understanding our fault-testing methodology. Next, we discuss how radiation affects FPGAs and various existing fault-injection tools to study these effects. We note that, for the remainder of this paper, we will focus on Xilinx FPGAs (and Xilinx-based terminology) because these FPGAs are predominately used in space systems with support for PR. However, even though we focus on Xilinx FPGAs, our analysis and injection methodologies can be generalized to any FPGA.

A. FPGA Architecture

A Xilinx FPGA’s architectural resource types include configurable logic blocks (CLBs) and specialized cores (BlockRAMs, multipliers, etc.) that are linked by a programmable on-chip interconnect. CLBs consist of relatively small components including look-up tables (LUTs), multiplexers, flip-flops (FFs), and supporting structures composed of basic logic gates, which enable implementation of complex logic functions. The on-chip programmable interconnect is a grid of switchboxes linked by wire segments of various lengths. Each switchbox integrates a large number of programmable switches allowing for custom routing of signals between CLBs and specialized cores. An FPGA’s configuration memory is reconfigured using a full bitstream, which contains the logic and routing configuration as well as initial values for the BlockRAMs and FFs. The majority of a full bitstream’s bits are responsible for switchbox configuration, followed by the BlockRAMs and other components [2, 3] An FPGA can be reconfigured via FR or PR. FR uses a full bitstream file that alters all of the configuration memory, while PR uses a partial bitstream file that alters only a portion of the configuration memory. A *frame*, the smallest programmable unit that a partial bitstream can alter, contains 1312 bits. Since modifying a single configuration memory bit requires reprogramming

an entire frame, injection time can be lengthy for SBFI methodology. FPGA configuration memory is logically divided into the top and bottom halves, which are mirror images of each other, and each half is divided into a series of rows and columns that form a coarse grid. The frame address, used to uniquely identify each frame, reflects this organization and consists of a top/bottom bit, block type, row, column, and minor addresses. Minor addresses describe the location of a frame inside each grid rectangle. Different FPGA models will have different ratios of rows, columns, and block types to reflect various device sizes and specializations. The device geometry describes the details of a particular FPGA model and provides mapping information detailing each frame's function (e.g., IOB, CLB, DSP, BlockRAM) and allows for a coarse classification of the bits in each frame. FPGAs can be reconfigured using a variety of interfaces [4, 5]. The JTAG port is the most commonly used reconfiguration and debugging interface since it is simple and available on most devices. However, JTAG speed is limited due to serial communication and the ability to daisy-chain multiple devices. Since this speed limitation is not suitable for systems that require rapid configuration for fast system bootup, the SelectMAP port is a faster alternative. SelectMAP offers a higher configuration speed due to a higher clock rate and a dedicated parallel connection.

B. Radiation Effects

While all space systems are vulnerable to radiation-induced faults, an FPGA's reconfigurable nature makes it more susceptible. SEUs, which cause transient faults in FFs and BlockRAMs, are similar to upsets on fixed-logic platforms and can lead to data corruption or single-event functional interrupts (SEFIs), where a device or design can enter an unexpected operating state. However, configuration memory faults are unique to FPGAs, as the function of the logic and interconnection network is controlled by the configuration memory. Configuration memory faults can lead to unpredictable application behavior due to broken connections, the formation of new connections, or changes to the LUTs that describe the logical functions. Radiation effects on reconfigurable FPGAs can be emulated by modifying the configuration memory's bits. Previous work showed that the vast majority of errors on FPGAs were attributed to SEUs in the configuration memory [1] and not errors in the embedded FFs, as the FFs represents only a small fraction of the total configuration memory

bits. For example, in the Virtex-4 SX55, the configuration memory's size (not including BlockRAM) is 15.4 million bits and the total number of FFs attached to slices is approximately 49,000 bits, thus making configuration memory upsets over 300 times more likely [6] than FF upsets. In order to investigate the effects of faults on FPGAs, effective fault-injection tools are required.

C. Fault-Tolerance Methods

To mitigate SEU effects in space systems, fault-tolerant FPGA designs use a variety of fault tolerance methods to improve reliability. The most popular and overhead-intensive method is triple-modular redundancy (TMR). TMR uses three identical replicas of the same module (design) working in lock step. The outputs of these models are compared using a majority voter to mask errors affecting any single module. TMR has a very high overhead (greater than 200%) because of the module replication and addition of the voter. There are two primary TMR variations: external and internal. External TMR uses three independent FPGAs connected to a reliable fourth device, which acts as the majority voter. Internal TMR places all replicated modules and the majority voter onto single FPGA. Whereas external TMR is more reliable than internal TMR because of a lack of a single-point of failure, external TMR requires more devices, power, and physical space [7]. To repair fault modules, TMR is usually augmented with a configuration scrubbing technique, which continually checks configuration memory for SEU-induced errors. The scrubber verifies configuration memory by reading the contents of the configuration memory and verifying embedded checksums or comparing it to the full bitstream file. When the scrubber detects a configuration memory error, it corrects the affected bit and signals the affected TMR module to resynchronize with the rest of the system.

Some lower-overhead, fault tolerance alternatives to TMR include (but are not limited to) self-checking pairs (SCP) [8], error-correcting codes (ECC), and algorithm-based fault tolerance (ABFT) [9, 10]. Many of these alternative techniques can detect errors quickly, but may require additional processing or re-computation to correct the errors.

D. Fault-Injection Platforms

Several injection platforms have been proposed in literature in recent years. Most of the proposed work is focused on reducing injection time by specializing hardware, which can limit the tool's portability and reusability.

Johnson, et al. [11] proposes an upset simulation methodology that uses a specialized platform, SLAAC-1V, consisting of two identical FPGAs (Virtex XCV1000s) executing in parallel and a voter to compare the FPGAs' outputs. One FPGA is configured using a corrupted bitstream, the other FPGA is configured with the original bitstream, and both FPGAs are supplied with the same input vectors. A difference in the FPGAs' outputs indicates that the corrupted bitstream causes a change in the configuration memory that enables an undesirable effect. This platform's custom, parallel architecture allows for high-performance testing and does not require *golden standard* (expected, fault-free outputs) to verify correct functionality tested FPGA.

The Xilinx Research Test Consortium (XRTC) platform [12] uses a base motherboard from SEAKR Engineering with a daughter-card containing an FPGA as the device-under-test (DUT). The motherboard contains two FPGAs that monitor the DUT's functionality and verify the DUT's outputs against a golden standard. Fault injection is performed through the JTAG port using an external computer or, in later versions, is integrated into the motherboard over the SelectMap port [6]. Even though the XRTC platform offers high performance, obtaining the golden standard requires a two-step testing approach where the first execution establishes the golden standard and the second execution determines the effects of the fault. However, like the SLAAC-1V platform, the XRTC platform uses specific hardware and custom boards and cannot be used with any other FPGA devices without a significant redesign.

Violante, et al. [13] proposes a System-on-Chip (SoC) platform and a split design that divides the FPGA (both physically and logically) into the unit-under-test (UUT) and supporting logic consisting of an embedded PowerPC, a timing unit, and an Internal Configuration Access Port (ICAP) controller. The tested logic is placed in the UUT and constrained to a portion of the FPGA to allow for PR. The supporting logic performs the fault injection using user-supplied test vectors (comprised of input data and the corresponding golden standard), which are stored on the

FPGA, and collects the results. This fault-testing method outperforms both the SLAAC-1V and the XRTC platforms in terms of performance due to the high-speed internal ICAP controller for PR and storage of the test vectors on the FPGA, which eliminates slow off-chip transfers. Although this tool is suitable for testing individual design components, the split design limits applicability and does not allow for standalone system testing. Additionally, available on-chip memory limits test vector sizes and the UUT is constrained to only resources not used by the supporting logic.

The Virtex-II SEU Emulator (V2SE) [14], which is similar to the XRTC platform, uses the SelectMap port for high-speed fault injections in combination with COTS and custom-designed hardware. Whereas the SelectMap port allows for fast reconfiguration, it is not as popular as the JTAG port and thus is not present on all devices.

III. SPFI-FPGA

SPFI-FPGA [15] is a flexible fault-injection platform for Virtex-4 and Virtex-5 FPGAs that tests the behavior of designs subjected to configuration memory faults. SPFI-FPGA is a subplatform of the larger SPFI platform that is targeted at system-level testing for various devices, including FPGAs, CPUs (e.g., PowerPC), and many-core devices (e.g., TILE64) [16]. SPFI provides maximum portability across a wide range of systems and enables high-performance, in-system testing. While SPFI-FPGA's performance is important, we avoid custom approaches that improve SPFI-FPGA's performance at the expense of limiting the platform's portability.

A. SPFI-FPGA System Architecture

Figure 1 shows SPFI-FPGA's high-level system architecture divided into three major software components: campaign generator (CG); management engine (ME); and test generator program (TG). The CG and ME execute on a host computer attached to the FPGA while the TG can be split between the host computer and the FPGA. Based on user parameters and the design's bitstream files, the CG generates a fault-injection campaign (FIC) file containing information necessary for the ME to identify the injection locations. During injection testing, the ME communicates with the FPGA over the programming interface (PI) to inject and remove faults, while the TG communicates

over the testing interface (TI) to assess the functionality of the FPGA.

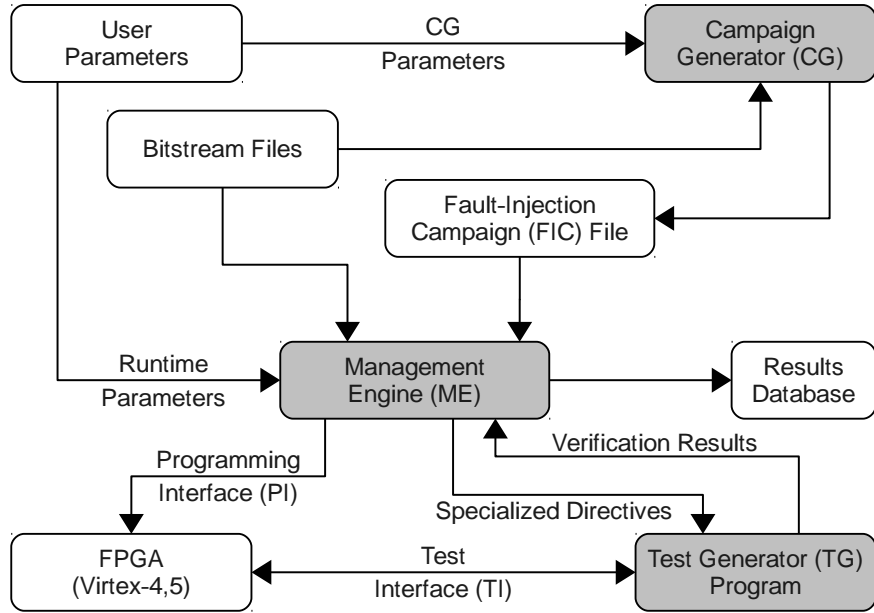


Fig. 1: Block diagram of SPFI-FPGA’s high-level system architecture. Primary software components, highlighted in grey, execute on a PC attached to the FPGA (functionality of the TG program may be partitioned between PC software and hardware wrapper on the FPGA).

The CG is the primary interface between user and the ME. The CG creates the FIC file based on user-specified parameters and automatic analysis of the design bitstream files. The FIC file contains injection locations (configuration memory addressing information for the injection bits) and an optional frame offset, which allows for fast localization of injection bits in the bitstream file by the ME at the time of injection.

User-specified parameters include fault-injection area, resource type, and number of bits to be tested. Users specify rectangular injection areas using two frame addresses to denote the rectangle’s diagonal vertices and the type of the resource to be tested. If the user-specified area contains more configuration bits than the number of bits in the campaign, the CG chooses which bits to test based on a random sampling of the injection area. In cases where the user wants to test a particular set of bits, the CG can be bypassed and the user can directly create the FIC file. Automatic bitstream analysis increases SPFI-FPGA’s portability by eliminating the need for a device database describing the geometry of each supported device. The device geometry describes the mapping

of a frame address to the approximate logical location on the device, which provides information about each frame's function (e.g., IOB, CLB, DSP, BlockRAM) and is used to decode the frame addressing scheme in the full bitstream file. The ME performs bitstream analysis by correlating the full bitstream file and the debug bitstream file generated by the Xilinx BitGen tool [17]. Similarly to the full bitstream file, the debug bitstream contains all of the frames required to initialize the device, however each frame in the debug bitstream file is clearly delimited and tagged with the frame address. The functional classification of each frame is performed using a two-pass approach. During the first pass, the ME uses the block type portion of the frame address to distinguish the frames containing the BlockRAM contents from other frames. The second pass counts the number of minor frames in row/column intersection. Each type of resource has a distinct number of minor frames required for programming, which allows the ME to perform functional classification. More detailed classification of bits in the bitstream requires detailed knowledge of the proprietary structure of the FPGA which is not publicly available.

The ME orchestrates SPFI-FPGA's main operation and is responsible for campaign management, FPGA monitoring, fault insertion and removal, and data logging. The ME's primary inputs are the design's bitstream files (full and debug bitstreams), the FIC, and user-specified ME runtime parameters. The ME runtime parameters allow users to control the type of campaign management used as well as various timeout mechanisms.

The ME's campaign management determines the order in which the faults are tested and orchestrates functionality verification. Ordering of the fault locations in the FIC file does not determine the runtime injection order. The ME employs different injection methodologies that change the injection order to improve the performance of testing. The TG performs functionality verification, when requested by the ME, using specialized directives stored in environmental variables. These directives specify the temporary storage location and basic state information about the ME (injection time, current fault locations, and iteration number). In addition to specialized directives, the ME can pass optional arguments to the TG as directed by the user. The TG returns the functionality verification results to the ME, which directs the ME's next steps as dictated by the current injection methodology. The injection results collected by the ME and the TG are logged in a results database,

which the user can use for detailed off-line statistical analysis.

The PI facilitates FPGA monitoring and programming and the ME provides a flexible application programming interface (API) for a variety of PIs. Since the JTAG port is available and accessible on most platforms, it serves as the primary PI, however a high level of abstraction in SPFI-FPGA’s architecture provides an easy and modular way for incorporating specialized PIs, such as SelectMap. The ME inserts and removes faults from bitstreams by automatically generating partial bitstream files with data frames that modify the appropriate configuration memory.

The TG verifies the FPGA’s functionality and provides the verification results as feedback to the ME. To maximize flexibility, the TG is a user-defined, plug-in application that communicates with the FPGA using the test interface (TI). The user must carefully define the TG since improper TG design can result in misclassification of injected faults and introduction of false-positives and false-negatives. The TG’s complexity can vary widely due to design choices and system architecture. The simplest version can leverage the FPGA system’s built-in self-test (BIST) feature to enable verification with little TG interaction. After the BIST completes, the test results are transferred back to the TG, which are forwarded on to the ME. The more complex TG version requires the TG to generate a set of test vectors to be transferred to the FPGA for self-testing. The results produced by the FPGA are transferred back to the TG to verify correctness. The most complex version of TG, divides the functionality between the PC host computer the design wrapper on FPGA. The wrapper is responsible for generating/storing test vectors and interfacing with the tested design, while the computer component of the TG is responsible for interfacing between wrapper and the ME. Such divided design can greatly reduce verification time as the test vectors can be stored directly on the FPGA.

B. Single-Bit Injection Methodology

Figure 2 depicts our single-bit fault-injection (SBFI) methodology, which leverages SPFI-FPGA and focuses on portability and correctness as well as performance and repeatability. Similarly to previous FPGA injection techniques [11–13], SBFI tests the vulnerability of a single configuration memory bit at a time. Since the main portability consideration is the type of configuration interface,

the JTAG port is the primary programming interface (testing methodology is the same regardless of programming interface).

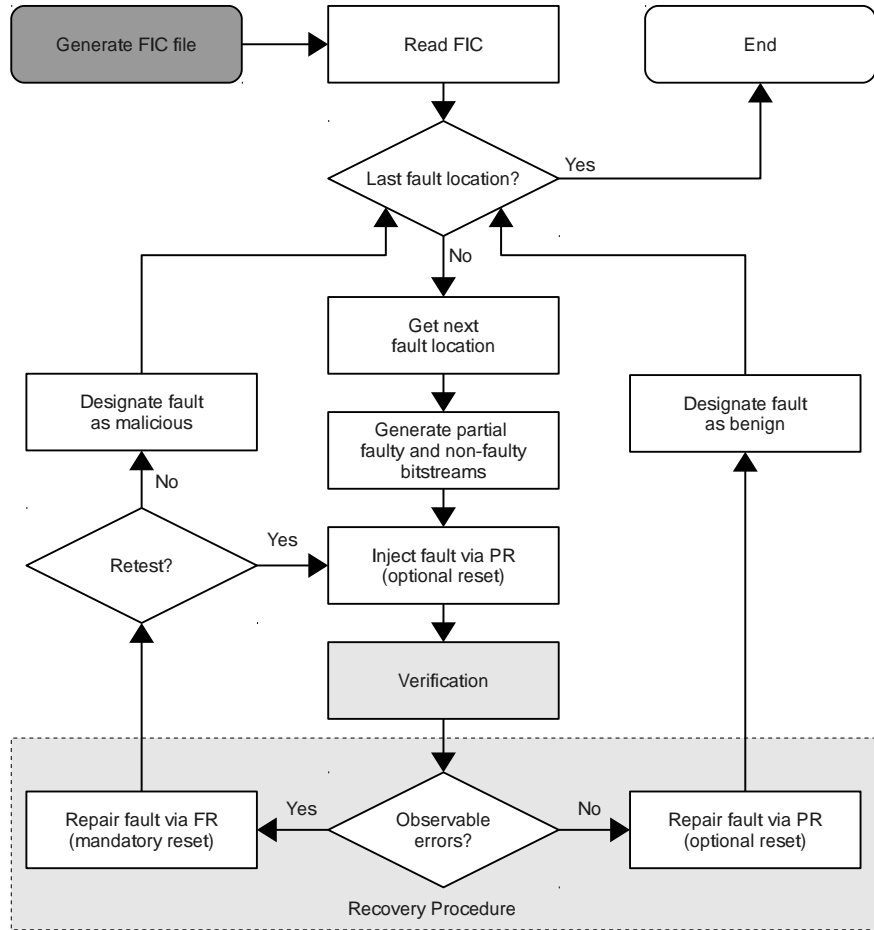


Fig. 2: SBFi methodology. Dark-grey action is performed by the CG, white actions by ME, and light-grey actions by TG.

Our SBFi methodology provides high-performance injection using a mixture of PR and FR techniques. Before the injection starts, we use CG to generate the FIC file. At runtime, the ME reads all of the fault locations from the FIC and recovers all the necessary frames from the debug bitstream file. The injection process starts when the ME selects a fault location to be tested. The fault is inserted by inverting the specified configuration memory bit into the configuration frame with corresponding address. Next, the ME generates two partial bitstreams: a *faulty* partial bitstream with the inverted configuration memory bit; and a *non-faulty* partial bitstream with the correct configuration memory bit. Due to limitations of Xilinx bitstream format, both partial bitstreams

contain a whole frame in order to inject a single fault. Using the faulty partial bitstream the ME uses PR to inject the fault into the FPGA. To increase fault-testing coverage, the FPGA’s internal execution state can be reset after injection using a reset option in the user-specified ME runtime parameters. Resetting the FPGA before verification enables the design’s startup behavior to be tested with respect to the injected fault; alternatively if the design is not reset, changes to the startup behavior might not be detected. However, in cases when the startup behavior is lengthy and will be performed infrequently in the final design, the user may opt to not reset the FPGA to improve testing performance. The TG verifies the FPGA’s operation and, based on the outcome of the operation (observable errors detected in the output), an appropriate recovery procedure is taken. If the TG does not observe an error, the FPGA is reconfigured with the non-faulty partial bitstream to repair (remove) the faulty bit, and the injected fault is designated as benign. If the TG detects an observable error, FR is used to repair and reset the FPGA to a nominal state. The faulty bit is then retested to remove any bias introduced by false positives that can occur due to multiple PRs, lack of testing coverage in the TG, or inconsistent behavior of other devices attached to the FPGA. If the TG detects errors during both tests, the injected fault is designated as malicious. The process loops to the fault location selection and is repeated until all the faults specified in the FIC are tested.

The performance (execution time) of our SBFI methodology can be approximately modeled as

$$t_{sbfi} = t_{pr} + t_v + p_e(2t_{fr} + t_{pr} + t_v) + (1 - p_e)t_{pr} + t_c \quad (1)$$

where: t_{sbfi} denotes the average execution time for one injection; t_{fr} and t_{pr} denote the full and partial reconfiguration times, respectively; t_v denotes the TG’s verification time; susceptibility denoted as p_e is the probability that the injected fault will be manifested as an observable error; and t_c denotes the constant software overhead per injection. The t_{sbfi} is strongly dependent on the TG’s performance as well as the PI’s speed.

C. Fault-Testing Methodologies

We propose two fault-testing methodologies based on the tested design’s architectural layout and source of input stimulus. *Module-level injection* (Figure 3a) tests small modules that require

input data to be explicitly provided, whereas *system-level injection* (Figure 3b) tests standalone systems that interface directly with external components.

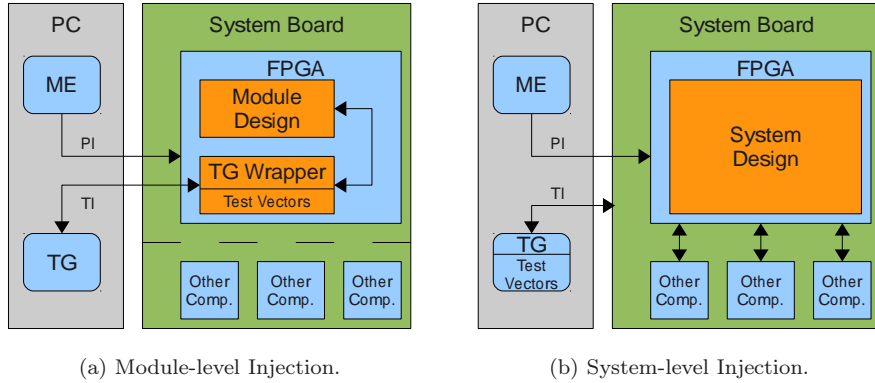


Fig. 3: SPFI-FPGA testing methodologies.

Module-level injection is most suitable for small designs (module designs) that do not interact with any external components and occupy only a fraction of the FPGA. Such module designs typically have reduced complexity requiring fewer test vectors as compared to larger, more complex designs. The remaining FPGA area can be used to offload most or all of the TG’s features from the attached host computer (PC, laptop) in the form of a TG wrapper. The TG wrapper consists of test vectors stored in the remaining BlockRAMs, TG interface, logic required to pass the test vectors to the tested design module, and comparison logic required to verify the module outputs against a golden standard. The functionality allowing the TG to interface with the ME as well as the interface to the TG wrapper remains on the host computer. Module-level injection increases injection performance by reducing or removing the communication delay between the TG and the FPGA via the TI.

System-level injection is most suitable for complex system designs that occupy the majority of the FPGA and are integrated with other external components (e.g., SDRAM, network, ADCs). For these large designs, a full-featured TG wrapper cannot fit on the remaining FPGA area, all of (or the majority of) the TG executes on the host computer and communicates with the system design via the TI, which might not be directly attached to the FPGA but instead to an intermediary device (microcontroller, etc.). In addition of providing the input vectors and verifying the FPGA’s output, the TG could be used to initialize the FPGA during boot-up, perform BIST, or trigger

the computation on the system. System-level injection allows observing the response of the entire system to the FPGA faults and can be used for testing complete space systems prior to deployment.

D. Fault-Injection Bottlenecks

We designed SPFI-FPGA to be portable across a variety of FPGA types and systems; however increasing portability generally decreases injection performance due to many bottlenecks affecting the injection process. The JTAG port is SPFI-FPGA's main performance bottleneck. Even though JTAG is available on most platforms and provides additional FPGA debugging features (boundary scan, device reset) that are not available for other PIs (e.g., SelectMAP, SlaveSerial), JTAG has limited communication speed (serial interface), limited clock rate, and in many cases the performance characteristics of the software/hardware interface between host computer and the JTAG programming cable are poor.

Additionally, the TG limits SPFI-FPGA's performance. A comprehensive and representative test-vector set for a complex module can be very large and require lengthy verification time that can be many orders of magnitude greater than injection time [18]. In the module-level testing methodology, storing test vectors onboard the FPGA can mitigate injection time.

The error rate, regardless of the testing platform, can also create a performance bottleneck. Error rate influences SPFI-FPGA's performance because the taken recovery procedure, and thus recovery time, is dictated by the TG's verification results. When the TG detects an error, the FPGA is fully reconfigured and the location is retested, whereas if the TG does not detect an error, the FPGA is partially reconfigured to remove the faulty bit. Since FR is very time consuming as compared to PR, the recovery procedure for observable errors can take an order of magnitude longer to execute. Fortunately, this bottleneck is not as prominent when testing the susceptibility of fault-tolerant designs, as the error rate is usually very small, causing only a few full reconfigurations.

Most FPGA's are equipped with an on-board FLASH subsystem that stores the initial full bitstream for rapid FPGA reconfiguration (fast FR). Therefore, to reduce the time required to perform FR when the TG detects an error, SPFI-FPGA leverages this subsystem to improve overall injection performance. Fast FR can be triggered using two mechanisms: a specialized JTAG signal;

or a command issued to the subsystem. Commands to the subsystem are issued via a user-provided custom plug-in program, which interfaces with the FPGA.

IV. Multi-Bit Fault Injection

The main goal of any fault-injection system is to provide timely and accurate susceptibility characterization of a given design and device (high injection performance), while maintaining portability. Unfortunately, many bottlenecks affect the achievable injection performance. One method for achieving a high injection rate is to modify a portion of the injection platform to remove bottlenecks. In particular, a modified PI can greatly improve the injection performance by decreasing FR and PR times, however, these modifications require major changes to the injection platform components, resulting in customized hardware that is tied to the particular platform (or narrow set of platforms) being tested and thus decreases portability. Alternatively, injection performance can be increased by decreasing the total number of required injection operations while maintaining the quality and fidelity of the injection results. Our methodology is inspired by multiple-bit upsets (MBUs) [19], which are caused by cosmic radiation, and occur when multiple faults are introduced due to an interaction with a single cosmic ray.

In this section, we propose a MBFI methodology that reduces the number of fault injections by coalescing multiple, single-bit injections into a single multi-bit injection, called a *batch* and uses both FR and PR and leverages SBFI methodology concepts. MBFI methodology yields identical results as compared to SBFI while significantly increasing the fault-injection performance. Subsection IV A describes the details of the proposed methodology and the modifications to SPFI-FPGA to support this methodology. Subsection IV B presents an optimization strategy that alters the order in which faults are injected to improve performance. In Subsection IV C, we show an algorithm for dynamic batch optimization and in Subsection IV D we discuss how to effectively apply our methodology to TMR designs.

A. Multi-Bit Injection Methodology

Figure 4 depicts our MBFI methodology as integrated into SPFI-FPGA. Initially, similarly to SBFI, the CG creates an FIC file. To ensure that bits in a batch belong to different CLBs, the

ME preprocesses the fault locations stored in the FIC file by first partitioning the fault locations by frame address (excluding minor address bits) into bins, where each bin's members are constrained by the same row/column address combination. Since fault locations in the FIC are selected randomly, some row/column addresses may occur more frequently than others, causing uneven partitioning which can lead to suboptimal batch selection. After preprocessing is completed, the ME begins injection by generating a batch. To generate a batch, the ME selects a single bit from each bin until the desired number of fault locations (batch size) is reached. Since each batch generation removes fault locations from the bins, if the batch size exceeds the number of non-empty bins, the resulting batch size is simply decreased to match the number of non-empty bins. To mitigate the uneven partitioning resulting from initial selection and removal of fault locations during the batch generation, the ME sorts bins in order of descending number of members before starting each batch generation process. The combination of preprocessing and sorting allows the batch generation process to maintain the user designated (static) batch size through as much of the FIC as possible.

Following the batch generation process, the ME creates both the faulty and non-faulty partial bitstreams. The faulty partial bitstream combines all of the required frames associated with all of the faulty bits specified in the batch, and the faulty bits are inverted. The number of faulty bits affects the reconfiguration time, since the reconfiguration time increases linearly with the number of configured frames and the number of frames in the bitstream is equivalent to the number of faults injected.

The TG's verification procedure is similar to single-bit injection, except in the case where an observable error is detected. We use a binary search algorithm with branch and bound, depicted in Figure 5, to detect faulty bits that caused the error. The binary search recursively divides the batch into two sub-batches. Each sub-batch is separately tested using multi-bit injection and branch and bound prunes the sub-batches that do not produce an error. This search process continues until bits that caused the error are determined. The example in Figure 5 shows a batch of size 16 with a single faulty bit. It takes 9 tests in total to detect the faulty bit.

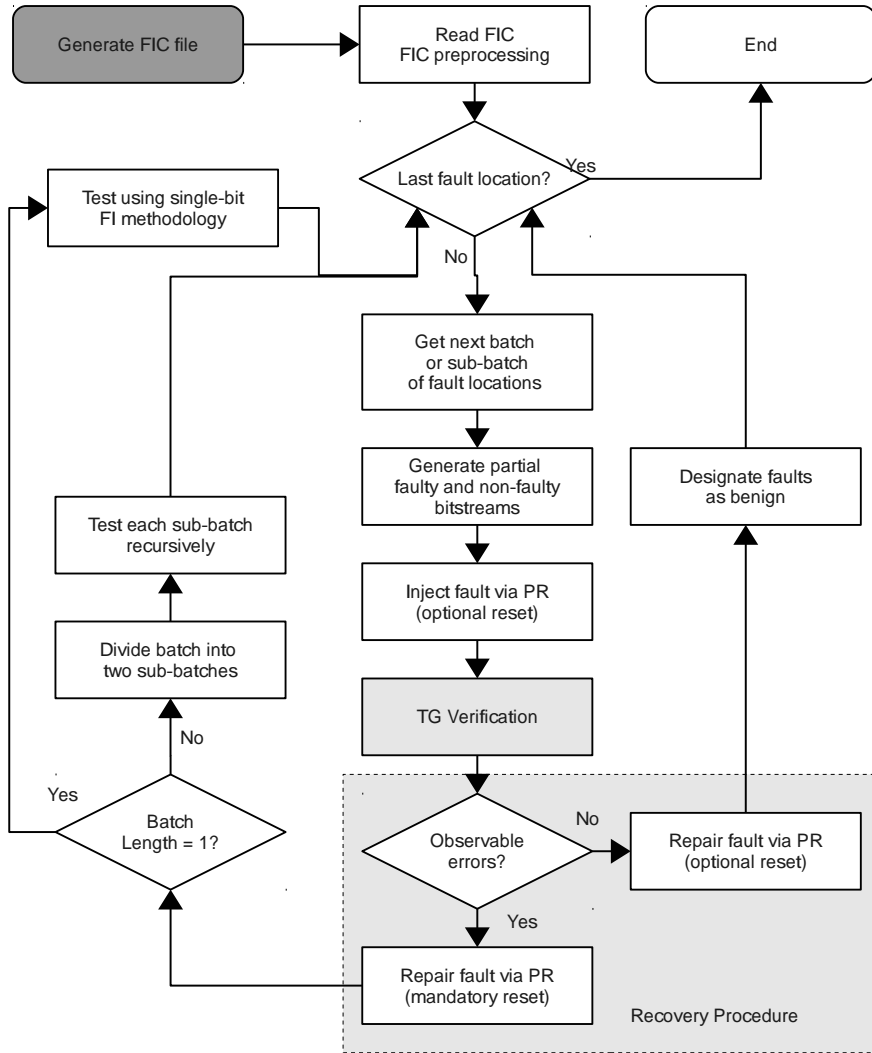


Fig. 4: Multi-bit injection methodology. Dark-grey, white, and light-grey components are part of the CG, ME, and TG, respectively.

B. Campaign Sequence Optimization

Bitstreams contain considerably more ‘0’ bits than ‘1’ bits because no design can take full advantage of the interconnect fabric in an FPGA. Our experimental results revealed that the majority of observable errors are caused by ‘1’ to ‘0’ transitions as opposed to ‘0’ to ‘1’ transitions. Since the majority of the FPGA’s configuration bits are responsible for routing signals, ‘1’ to ‘0’ transitions are more likely to affect signal routing. In Xilinx FPGAs, switchboxes not used for routing contain only ‘0’ bits and switchboxes that are used for routing contain some ‘1’ bits. Therefore, ‘0’ to ‘1’ transitions can create short circuits and ‘1’ to ‘0’ transitions can create open circuits (i.e., ‘0’ to ‘1’

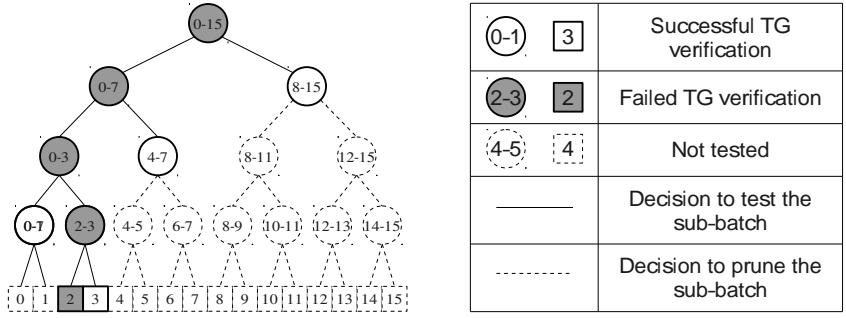


Fig. 5: Example decision tree used by binary search algorithm to determine faulty bits in a batch. The numbers inside nodes indicate range of bits in the current sub-batch. Circular nodes indicate batch tests while square nodes indicate single-bit tests.

transitions can create new, but unused signal routing, and ‘1’ to ‘0’ transitions can break required signal routing).

Based on this observation, campaign management can improve fault-testing performance by optimizing the sequence of tested bits based on the outcome of the bit upset (‘1’ to ‘0’ or ‘0’ to ‘1’). If ‘0’ to ‘1’ transitions are grouped at the beginning of the campaign and ‘1’ to ‘0’ transitions are grouped at the end, the fault distribution can be skewed from a uniform scattering throughout the campaign to a low-concentration area where faults are less likely to occur (at the beginning of the campaign) and a high-concentration area (‘1’ to ‘0’ transitions at the end of the campaign). Campaign sequence optimization (CSO) increases the fault-testing performance for larger batch sizes. CSO is implemented in SPFI-FPGA by augmenting the FIC preprocessing and batch generation process. In addition to partitioning the fault locations into bins, each bin is sorted where the ‘0’ to ‘1’ transitions are selected first during the batch generation process. During the batch generation process, the ME selects ‘0’ to ‘1’ transitions first, until none are available, then it starts selecting the ‘1’ to ‘0’ transitions.

C. Dynamic Batch-Size Optimization

The main difficulty the user experiences using MBFI methodology is determining a static batch size to obtain the best injection performance. However, there is no single, optimal batch size since the optimal batch size is dependent on the type and complexity of the design being tested. Suboptimal

batch sizes can result in decreased injection performance, but determining the optimal batch size is difficult and requires a detailed timing model of SPFI-FPGA and prior knowledge of the design’s characteristics. Determining the optimal batch size is further complicated by designs employing FT techniques, since a detailed knowledge of the design’s sub-modules and the susceptibilities of each sub-module are needed. Additionally, smaller than optimal batch sizes increase the number of batch tests, and larger than optimal batch sizes increase the number of false-positive batch tests. However, false-positive batch tests do not affect the outcome of the fault testing and are pruned out during the binary search. False-positive tests are batch tests that fail, but no individual faulty bit causes a fault. False positives are especially prominent in FT designs where a combination of faults is required in order for an observable error to occur. Additional complications in determining the batch size are also introduced when using CSO where susceptible bits are not distributed uniformly across the campaign. Whereas using different batch sizes for each part of the campaign could yield nearly optimal performance, but prior knowledge of susceptibilities is required for ‘0’ to ‘1’ transitions and ‘1’ to ‘0’ transitions. Therefore, in order to address these difficulties in determining an optimal batch size, we propose a dynamically variable size where the batch size changes during the FIC.

A dynamic batch-size optimization (DBSO) can improve fault-injection performance and allow for better usability of SPFI-FPGA. Instead of statically choosing a batch size at the beginning of an FIC and waiting until the end of injection to determine the fault-injection performance, the ME dynamically varies the batch size. During runtime, the ME records average batch testing time for each batch size and after each batch, the ME calculates the new batch size using Algorithm 1. Since the batch testing times are collected during the runtime, the average batch test time varies significantly during the beginning of the FIC since few testing times have been collected. To address these variations, we use a greedy hill-climbing algorithm combined with a random walk algorithm for selecting the next batch size [20].

Algorithm 1 shows the proposed DBSO process. The algorithm takes as input the current batch size s , user-defined parameter α , and an array $t_b[x]$ that contains the average batch testing times for batch size x and calculates and returns the next batch size s_{next} . Step 1 (hill climbing) uses local

Algorithm 1 Dynamic Batch Size Optimization.

Input: $s, t_b[], \alpha$ **Output:** s_{next}

```
1: if  $t_b[s - 1] > t_b[s] > t_b[s + 1]$  then {Step 1 - Hill Climbing}
2:      $s_{next} = s + 1$ 
3: else if  $t_b[s - 1] < t_b[s] < t_b[s + 1]$  then
4:      $s_{next} = s - 1$ 
5: else {Step 2 - Weighted Random Walk}
6:     if  $rand() < \alpha$  then
7:          $s_{next} = s - 1$ 
8:     else
9:          $s_{next} = s + 1$ 
10:    end if
11: end if
12: return  $s_{next}$ 
```

gradient information about $t_b[x]$. If $t_b[s + 1] < t_b[s] < t_b[s - 1]$, the algorithm increases s_{next} by one. If $t_b[s - 1] < t_b[s] < t_b[s + 1]$, the algorithm decreases s_{next} by one. If both line 1 and 3 fail, the algorithm moves to Step 2 (weighted random walk), where s_{next} is either decreased or increased by one randomly based on the probability α of decreasing the batch size. This random walk enables the algorithm to escape local minimums and increases the number of time measurements on batches next to s . SPFI-FPGA uses $\alpha = 0.5$, which favors increasing the batch size during the initial batch tests due to uninitialized values in $t_b[]$. In these cases, increasing α stabilizes the algorithm at a small performance penalty.

D. Placement-Aware Fault Injection

COTS FPGA-based space systems rely on TMR to mitigate effects of SEUs. In many cases, multiple cores are located on the same FPGA to minimize size, weight, and power requirements using tools like the Xilinx TMRTool [21] or BYU Tool [22]. However, complete internal TMR is impossible to achieve for a single FPGA because of the single points of failure inherent in the FPGA fabric. Performing injection on a true TMR design should show that no single SEU can affect the output of the design, however when using MBFI methodology it is possible to corrupt bits in more

than one module of the TMR core, which induces a false positive batch test and has a negative impact on injection performance.

Fault-injection performance for a TMR design on a single FPGA can be significantly improved by leveraging the underlying structure of TMR designs. The placement-aware fault-injection (PAFI) technique leverages the TMR structure as placed on the FPGA, where three modules are physically separated on different parts of the chip. Without knowing the structure of the TMR design, SPFI-FPGA can only create campaigns that uniformly test bits across the TMR design as depicted in Figure 6a. However, if the structure of the TMR layout is known prior to fault-injection, the FIC can be split into three FICs where each part of campaign tests only one module of the TMR system as depicted in Figure 6b. Separately testing each single module significantly reduces false positive batch tests and allows for larger batch sizes, thereby reducing the injection time and significantly improving testing performance.

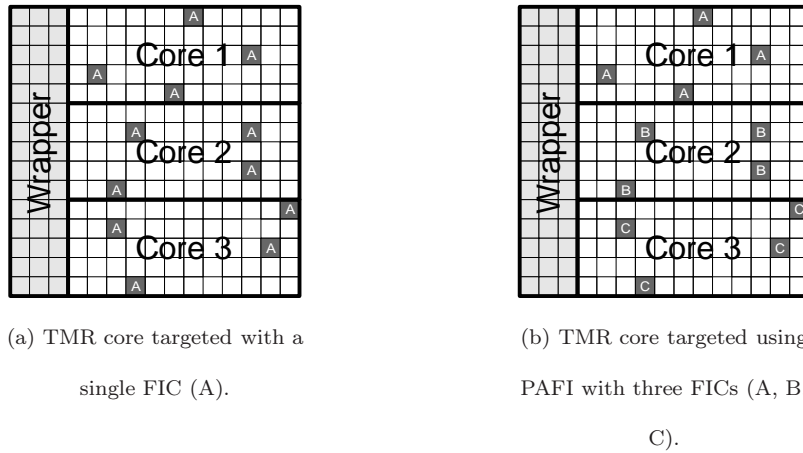


Fig. 6: PAFI applied to a TMR core. Each square in the grid denotes a bit that can be tested. Dark squares denote bits targeted while symbols (A, B, C) inside of dark squares indicate the FIC to which the bits belong.

V. Results

We use three diverse case studies on three different FPGA test boards to evaluate the performance of our SPFI-FPGA platform augmented with MBFI methodology and to showcase the platform’s effectiveness, portability, and efficiency in detecting faults. Each case study is designed

to evaluate a different combination of key factors (e.g., susceptibility, verification time, and fault-testing methodology) that effect SPFI-FPGA’s performance. The first study performs a fast Fourier transform (FFT) on a Xilinx ML401 development board with a Virtex-4 LX25 FPGA. This study uses the module-level testing methodology, has the fastest verification time, and highest fault susceptibility out of the three studies. The second study runs a MicroBlaze system that performs LU decomposition on a Xilinx ML501 board with a Virtex-5 LX50 FPGA. This study uses the system-level testing methodology, which has the slowest verification time, while the fault susceptibility falls between the other two studies. The third study performs a matrix multiply (MM) on a NASA SpaceCube [23] platform with a Xilinx Virtex-4 FX60 FPGA. This study uses a hybrid approach between the module- and system-level testing methodologies, the verification time is slower than the FFT study but faster than the MicroBlaze study (i.e., the MM study has a midrange verification time), and the lowest fault susceptibility out of the three studies. Each study evaluates two types of cores: a base core with no fault-tolerant features (NFT) and a coarse-grain TMR core that is suitable for space systems.

Each study uses a similar experimental system setup. The FPGA test board is connected with the PI and the TI to a Linux-based host computer that runs a portion of SPFI-FPGA (the ME and the TG) (Figure 3). Depending on the testing methodology, the TG’s functionality can be partitioned between the host computer and the FPGA (Section 3.3). The PI uses a Xilinx Platform Cable and the TI uses an FTDI 232R (FFT and MicroBlaze studies) or an FTDI 2324H (MM study) USB-to-serial converter cable to interface with FPGA logic.

Table 1: Case-study attributes.

Case Study	FPGA Type	FR Time [s]	PR Time (min-max) [ms]	Max Batch Size	NFT Core Suscept.	TMR Core Suscept.	Verif. Time [s]
FFT	V4 LX25	3.6	110-180	40	5.5%	0.10%	0.03
MicroBlaze	V5 LX50	5.6	80-180	40	1.6%	0.03%	2.01
MM	V4 FX60	6.2	80-280	130	1.5%	0.01%	0.29

Table 1 summarizes attributes for each study. The FR and the PR times for each platform vary based on the JTAG performance and the type of FPGA. The PR times are dependent on the batch size since the number of frames in the partial bitstream is equivalent to the batch size. The maximum batch size varies based on the area occupied by the core on the FPGA and is equivalent to the initial number of bins generated during batch selection (Section IV A). The FFT and MicroBlaze studies’ maximum batch sizes are 40 and the MM study’s maximum batch size is 130. We measured the susceptibilities (p_e) and verification times for each core during our experiments. Each core’s susceptibility is constant and depends on the design’s complexity and how the design is mapped to the FPGA. The fault susceptibility is estimated as a ratio of malicious faults recorded during FIC over the total number of fault injections performed (e.g., for the FFT NFT core, $552/10000 = 0.0552$). As expected, the NFT cores show much higher susceptibility than the TMR cores. The average verification time is obtained based on the data collected by SPFI-FPGA during each experiment.

We evaluate the performance of SPFI-FPGA by comparing the total FIC injection times for SBFI to MBFI using three different optimization strategies: CSO, DBSO, and PAFI. For each study, we first measure the baseline injection times using SBFI for the NFT and TMR cores. Next, we collect the injection times for the NFT and TMR cores using MBFI and MBFI with CSO using a static batch size. We vary the batch size for each experiment from two up to the maximum batch size possible. Additionally, we perform another set of experiments on the TMR core using PAFI and measure the injection times for MBFI and MBFI with CSO. To evaluate injection time for DBSO, we perform one experiment for each combination of optimizations (NFT, NFT & CSO, TMR, TMR & CSO, TMR & PAFI, and TMR & PAFI & CSO). We report performance in terms of speedup, which is calculated as the ratio of baseline SBFI injection time over MBFI injection time.

To compare the different optimization strategies, we use two randomly generated FIC files per study: one for the NFT cores and one for TMR cores. To obtain a fair estimate of the core’s susceptibility and injection times while maintaining a reasonable injection time, we inject 10,000 faults per experiment.

A. FFT Case Study

FFT is a key signal-processing algorithm used in space-based applications, such as signal filtering, synthetic aperture radar (SAR) [24], and software-defined radio (SDR) [25]. FFT is highly amenable to FPGA implementation, resulting in compact and efficient use of the FPGA fabric. In this study, the FFT core performs a 1024-point complex fixed-point FFT, where each point consists of two 32-bit values. We generate the FFT core using the Xilinx CoreGen tool, which optimized the design for the ML401 board. We choose the module-level testing methodology (Figure 3a) for the FFT study as the core created is not designed to interface with external logic and requires a wrapper to provide test vectors. Additionally, the core occupies a small area of the FPGA that allows for limited storage of the test vectors in the remaining BlockRAMs, as shown in Figure 7. Only two randomly generated test vectors are used due to the limited number of BRAMs available for the wrapper on the LX25 FPGA. The FFT’s TMR core integrates the voting logic into the wrapper. The host’s TG partition is lightweight and only sends verification requests to the wrapper module. The wrapper module performs the verification and returns the number of errors encountered during the verification process. The FFT core has shortest verification time as it uses module-level testing methodology and does not require sending the test vectors over the TI.

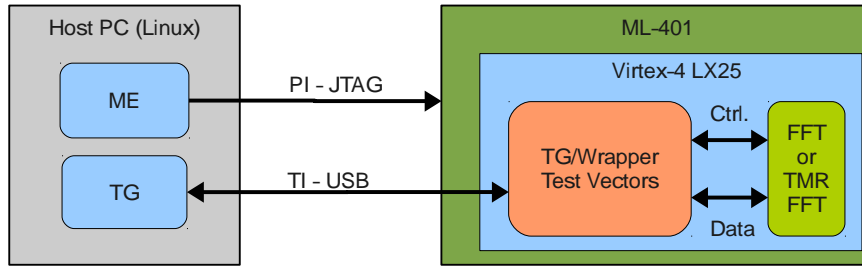


Fig. 7: Experimental setup of the FFT study.

Figure 8 depicts injection speedups attained by SPFI-FPGA using MBFI methodology versus increasing batch size for the FFT NFT, TMR, and TMR with PAFI cores. MBFI methodology yields only minor speedups for the FFT NFT core due to the combination of the FFT NFT core’s high susceptibility and fast verification time. In this situation, the cumulative FR and PR times dominate the total injection time and thus prohibit large speedups. MBFI methodology achieves moderate speedups for both FFT TMR cores due to these cores lower fault susceptibility as compared to the

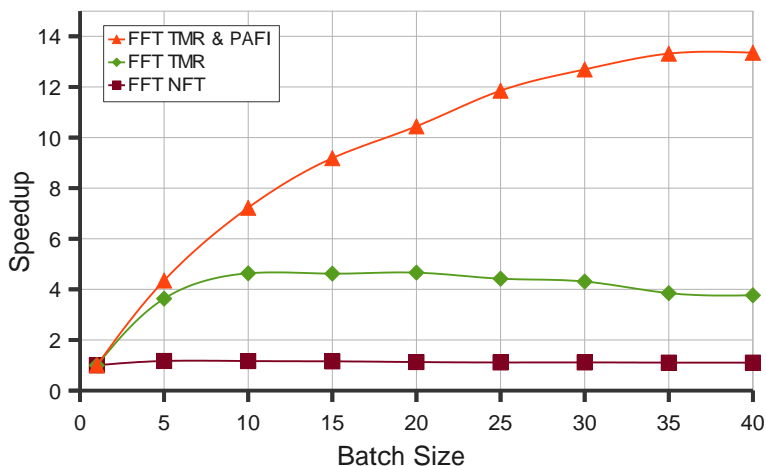


Fig. 8: Injection speedup for FFT study.

FFT NFT core, and thus requiring fewer reconfigurations.

Table 2 summarizes the FFT study results for each core version both with and without using CSO. Columns two and three show the best speedups (corresponding to maxima in Figure 8) and the associated runtimes when not using the DBSO algorithm. Columns four and five show the speedups and associated runtimes when using the DBSO algorithm. Column six compares these speedups using the ratio of the DBSO time over the best time, which shows the efficiency of the DBSO algorithm. The maximum speedups attained for the FFT NFT core without and with CSO are $1.2\times$ and $1.1\times$, respectively. In both of these experiments, the DBSO algorithm performs at 99% efficiency. The maximum speedups attained for FFT TMR core without and with CSO are $4.7\times$ and $4.6\times$, respectively, and the DBSO algorithm performs at 76% and 102% efficiency, respectively. Finally, the maximum speedups attained for the FFT TMR core with PAFI without and with CSO are $13.3\times$ and $13.1\times$, respectively. The best speedups for experiments with CSO show a slight performance drop due to additional computational overhead required during the batch selection process. The maximum speedups for experiments without and with CSO are nearly identical, however for the TMR core, the DBSO algorithm is more efficient when using CSO. Results of this study show that using MBFI methodology with CSO and DBSO significantly decreases injection time of the TMR cores that have verification time shorter than the PR time.

Table 2: Summary of experimental results for FFT study.

FI Options	Best Speedup	Best Time [s]	DBSO Speedup	DBSO Time [s]	Efficiency
NFT	1.2	5373.0	1.2	5402.6	99%
NFT & CSO	1.1	5534.5	1.1	5586.5	99%
TMR	4.7	962.1	3.6	1264.4	76%
TMR & CSO	4.6	1017.3	4.7	994.6	102%
TMR & PAFI	13.3	410.6	13.1	420.4	99%
TMR & PAFI & CSO	13.1	422.1	13.0	426.1	99%

B. MicroBlaze Case Study

The MicroBlaze is a flexible 32-bit soft-core processor optimized for embedded applications commonly used on Xilinx FPGAs. The MicroBlaze is often used as a part of SoCs to manage the control flow or to perform calculations that are difficult to efficiently implement on an FPGA, like the LU decomposition algorithm that is primarily used to solve systems of simultaneous equations. The MicroBlaze core is created using the Xilinx Platform Generator tool, which allows customization of the MicroBlaze’s memory and IO components. We configure the MicroBlaze core with 24kB of RAM, a serial IO port, a hardware-accelerated floating-point unit, and an optimized barrel shifter. The LU decomposition algorithm [26, 27] uses single-precision floating point arithmetic and processes a 5×5 matrix A and outputs two 5×5 matrices L and U such that $A = L \cdot U$. In the MicroBlaze study, depicted in Figure 9, we choose the system-level testing methodology (Figure 3b) as the MicroBlaze core created is suitable for stand-alone use and interfaces directly with the TI. The MicroBlaze TMR core has a small wrapper with a voter to compare the MicroBlaze’s three serial port outputs. The TG sends 100 test vectors each containing an input matrix A and receives and compares the results of the computation against the golden standard (matrices L and U). The MicroBlaze study has the longest verification time as the entire TG executes on the Linux-based host computer, does not offload any functionality to the wrapper, and the test vectors need to be transferred over the TI. Additionally, the serial LU decomposition implementation on MicroBlaze consumes longer time

to calculate results than standalone cores in FFT and MM studies.

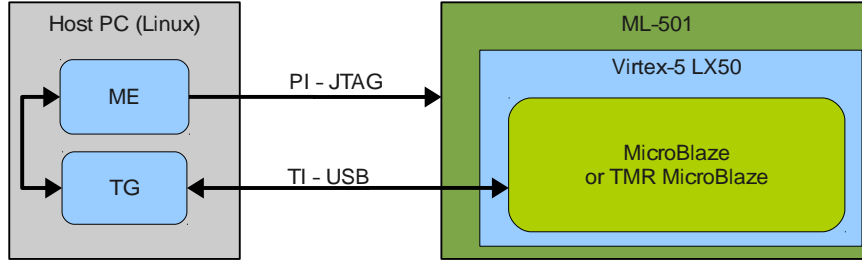


Fig. 9: Experimental setup of the MicroBlaze study.

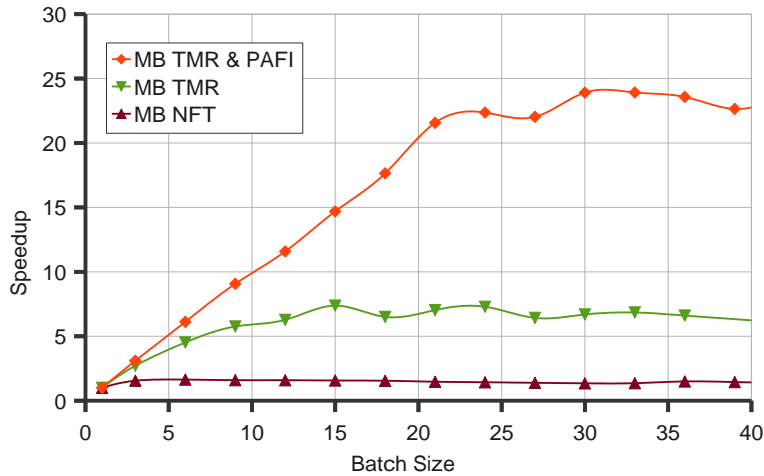


Fig. 10: Injection speedup for MicroBlaze study.

Figure 10 depicts injection speedups attained by SPFI-FPGA using MBFI methodology versus increasing batch size for the MicroBlaze NFT core, TMR, and the TMR with PAFI cores while executing the LU decomposition algorithm. These results show similar trends but higher speedups as compared to the FFT cores. MBFI methodology yields a small but better than the NFT FFT speedups for the MicroBlaze NFT core. The MicroBlaze TMR and TMR with PAFI cores yield significant speedups over SBFI methodology. The MicroBlaze study's speedups are higher than the FFT study because of decreased susceptibility and increased verification time.

Table 3 summarizes the MicroBlaze study results (the table layout is similar to the FFT study results in Table 2). The maximum speedups attained for the MicroBlaze NFT core without and with CSO are $1.6\times$ and $1.7\times$, respectively, and the DBSO algorithm performs at 99% and 96% efficiency, respectively. The maximum speedups for the MicroBlaze TMR core without and with

Table 3: Summary of experimental results for MicroBlaze study.

FI Options	Best Speedup	Best Time [s]	DBSO Speedup	DBSO Time [s]	Efficiency
NFT	1.6	19821.7	1.6	20083.9	99%
NFT & CSO	1.7	18697.3	1.7	19416.5	96%
TMR	7.4	3474.0	6.5	3955.9	88%
TMR & CSO	8.2	3122.0	7.1	3613.9	86%
TMR & PAFI	23.9	886.1	22.7	1031.5	86%
TMR & PAFI & CSO	23.8	982.6	23.4	1001.9	98%

CSO are $7.4\times$ and $8.2\times$, respectively, and the DBSO algorithm performs at 88% and 86% efficiency, respectively. Finally, maximum speedups for the MicroBlaze TMR core with PAFI and without and with CSO are $23.9\times$ and $23.8\times$, respectively, and the DBSO algorithm performs at 86% and 98% efficiency, respectively. In this study, CSO improves the speedup obtained by the DBSO algorithm's in all experiments, and maintains efficiency of DBSO of more than 86%.

C. Matrix Multiply Case Study

The MM core performs a 16-bit fixed-point matrix multiplication on two randomly generated 25×25 matrices. MM is a common kernel in signal and image processing applications, such as beamforming [28] and hyper-spectral imaging (HSI) [29, 30]. Although FPGA area constraints limit the size of a single matrix multiplication, larger matrices can be dividing into smaller blocks, which are processed individually and then coalesced into the final results matrix. The MM algorithm is parallelized over n processing units, which allows for the calculation of the dot product in a single clock cycle, thereby reducing the computational time from $O(n^3)$ to $O(n^2)$ clock cycles. The core uses embedded BlockRAM and DSP resources available on the Virtex-4 FX60 FPGA. We use a hybrid testing methodology in the MM study, depicted in Figure 11, as the core created requires a wrapper to interface with external logic; however there is not enough resources left on the FPGA to store the test vectors. The verification time of MM experiments is longer than the FFT study as

the test vectors need to be transferred over the TI, but it is shorter than MicroBlaze study as the MM core has much higher throughput due to MM core's parallel nature.

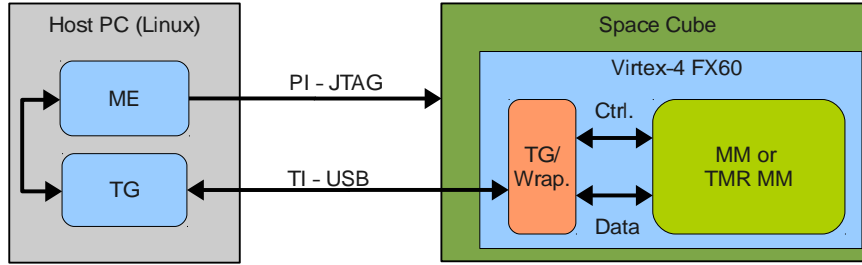


Fig. 11: Experimental setup of the MM study.

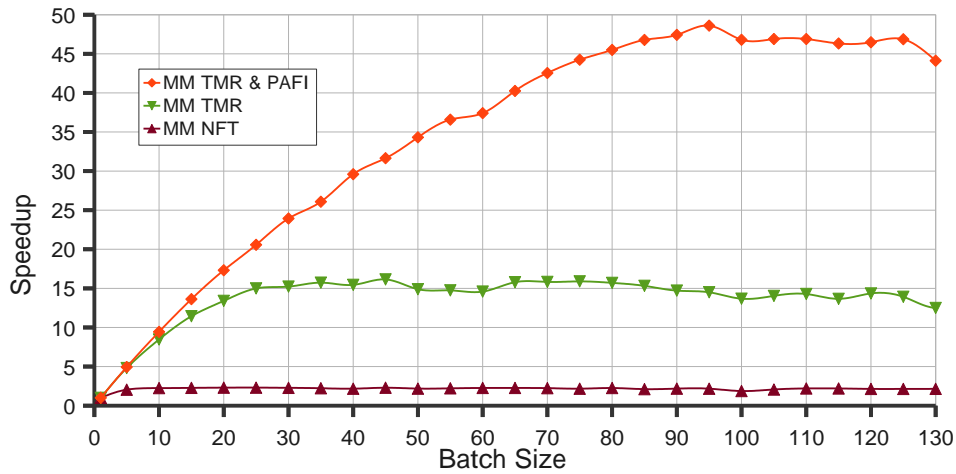


Fig. 12: Injection speedup for MM study.

Figure 12 depicts injection speedups attained by SPFI-FPGA using MBFI methodology versus increasing batch size for the MM NFT core, TMR, and the TMR with PAFI cores. As compared to the FFT and MicroBlaze studies, the MM study reveals the best speedups for the MM NFT and TMR cores due to their low susceptibility and midrange verification time.

Table 4 summarizes the MM study results (the table layout is similar to the FFT study results in Table 2). The maximum speedup for the MM NFT core without and with CSO is $2.3\times$, and the DBSO algorithm performs at 96% and 97% efficiency, respectively. The maximum speedups for the MM TMR core without and with CSO are $16.1\times$ and $18.3\times$, respectively, and the DBSO algorithm performs at 81% and 83% efficiency, respectively. The maximum speedups for the MM TMR core with PAFI and without and with CSO are $48.6\times$ and $49.7\times$, respectively, and the DBSO algorithm

performs at 96% and 93% efficiency, respectively. This study shows that experiments with CSO improve the DBSO algorithm’s achievable speedup with the exception of the MM TMR core with PAFI where it is marginally lower, while maintain efficiency of more than 83%.

Table 4: Summary of experimental results for MM study.

FI Options	Best Speedup	Best Time [s]	DBSO Speedup	DBSO Time [s]	Efficiency
NFT	2.3	3025.9	2.2	3126.6	96%
NFT & CSO	2.3	3152.0	2.3	3073.3	97%
TMR	16.1	623.9	13.0	772.0	81%
TMR & CSO	18.3	551.5	15.0	656.7	83%
TMR & PAFI	48.6	289.7	46.7	302.4	96%
TMR & PAFI & CSO	49.7	297.3	46.1	305.1	93%

VI. Conclusions

SRAM-based FPGA devices are an attractive option for on-board data processing on space systems. However, FPGAs present unique reliability verification challenges because SEUs can trigger configuration memory errors. In this work, we have presented SPFI-FPGA, a new fault-injection platform that uses a mix of PR and FR to test the behavior of designs subject to configuration memory faults. We have shown that the SPFI-FPGA platform can be effectively used on a variety of modern Xilinx FPGAs and that the testing methodologies we have developed allow testing on a variety of designs. Furthermore, the ideas presented in this paper are not limited to Xilinx FPGAs but are applicable to a wide range of devices.

In addition to SPFI-FPGA, we presented a novel MBFI methodology for FPGAs that modifies multiple configuration memory bits during each test. We have shown multiple optimization techniques (CSO, DBSO, and PAFI) which coupled with MBFI methodology allow for very effective testing of FPGA designs. During the course of our experiments, we were able to achieve maximum injection speedup of up to $49.7\times$ for MM TMR core employing PAFI. The investigated DBSO al-

gorithm in conjunction with CSO enabled efficient selection of optimal batch size, and allowed for effective use of MBFI methodology. We achieved over 90% efficiency in most of our studies while using the combination of DBSO algorithm and CSO. Use of FPGAs on space-based platforms is highly effective for increasing computational power of the system per unit energy, but appropriate measures must be taken to determine reliability of the design. High testing speedups demonstrated together with efficient dynamic batch selection can dramatically decrease the time required to test designs in preparation for space deployment.

Future work in this direction may explore further optimizations to the proposed methodology. In particular, an alternative campaign optimization strategy would involve separating bits belonging to the switchboxes from bits belonging to other components in the FPGA fabric. Campaigns optimized with this information in mind could yield even better results than the statistical approach based on transitions used in the paper. Unfortunately, such an approach would require in-depth knowledge of bitstream structure, which is vendor-proprietary and thus not publicly available.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Xilinx, and Aldec.

References

- [1] Johnson, E., Caffrey, M., Graham, P., Rollins, N., and Wirthlin, M., “Accelerator validation of an FPGA SEU simulator,” *Nuclear Science, IEEE Transactions on*, Vol. 50, No. 6, 2003, pp. 2147 – 2157.
- [2] Xilinx, Inc, *Virtex-4 Family Overview*, 2010. Product Specification, DS112 (v3.1).
- [3] Xilinx, Inc, *Virtex-5 Family Overview*, 2009. Product Specification, DS100 (v5.0).
- [4] Xilinx, Inc, *Virtex-4 FPGA Configuration User Guide*, 2009. User Guide, UG071 (v1.11).
- [5] Xilinx, Inc, *Virtex-5 FPGA Configuration User Guide*, 2010. User Guide, UG191 (v3.9.1).
- [6] Allen, G., Swift, G., and Carmichael, C., “Virtex-4VQ static SEU characterization summary,” Tech. rep., Xilinx Radiation Test Consortium, 2008.
- [7] Bridgford, B., Carmichael, C., and Tseng, C., *Single-Event Upset Mitigation Selection Guide*, Xilinx, Inc, 2008. Application Note, XAPP987 (v1.0).

- [8] Laprie, J.-C., Arlat, J., Beounes, C., and Kanoun, K., "Definition and analysis of hardware- and software-fault-tolerant architectures," *Computer*, Vol. 23, No. 7, 1990, pp. 39 –51.
- [9] Huang, K.-H. and Abraham, J., "Algorithm-Based Fault Tolerance for Matrix Operations," *Computers, IEEE Transactions on*, Vol. C-33, No. 6, 1984, pp. 518 –528.
- [10] Wang, S.-J. and Jha, N., "Algorithm-based fault tolerance for FFT networks," *Computers, IEEE Transactions on*, Vol. 43, No. 7, 1994, pp. 849 –854.
- [11] Johnson, E., Caffrey, M., and Wirthlin, M., "Single-Event Upset Simulation on an FPGA," in "Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA) Conference," , 2002.
- [12] Petrick, D., Powell, W., Howard Jr., J., and K., L., "Virtex-II Pro PowerPC SEE Characterization Test Methods and Results," in "Proceedings of the Military and Aerospace Programmable Logic Devices (MAPLD) Conference," , 2005.
- [13] Violante, M., Sterpone, L., Ceschia, M., Bortolato, D., Bernardi, P., Reorda, M., and Paccagnella, A., "Simulation-based analysis of SEU effects in SRAM-based FPGAs," *Nuclear Science, IEEE Transactions on*, Vol. 51, No. 6, 2004, pp. 3354 – 3359.
- [14] French, M., Graham, P., Wirthlin, M., Wang, L., and Larchev, G., "Radiation Mitigation and Power Optimization Design Tools for Reconfigurable Hardware in Orbit," in "Proceedings of Earth-Sun System Technology Conference," , 2005.
- [15] Cieslewski, G. and George, A. D., "SPFFI - Simple Protatable FPGA Fault Injector," in "Proceedings of the Military and Aerospace Programmable Logic Devices (MAPLD) Conference," , 2009.
- [16] Wulf, N., Cieslewski, G., Gordon-Ross, A., and George, A., "SCIPS: An emulation methodology for fault injection in processor caches," in "Aerospace Conference, 2011 IEEE," , 2011, pp. 1 –9.
- [17] Xilinx, Inc, *Command Line Tools User Guide*, 2010. User Guide, UG628 (v12.2).
- [18] Sterpone, L. and Violante, M., "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs," *Nuclear Science, IEEE Transactions on*, Vol. 54, No. 4, 2007, pp. 965 –970.
- [19] Quinn, H., Graham, P., Krone, J., Caffrey, M., and Rezgui, S., "Radiation-induced multi-bit upsets in SRAM-based FPGAs," *Nuclear Science, IEEE Transactions on*, Vol. 52, No. 6, 2005, pp. 2455 – 2461.
- [20] Poole, D. L. and Mackworth, A. K., *Artificial Intelligence: Foundations of Computational Agents*, Cambridge University Press, New York, 2010. Also full text online at <http://artint.info>.
- [21] Xilinx, Inc, *XTMR Tool User Guide*, 2004. User Guide, UG156.
- [22] Pratt, B., Caffrey, M., Graham, P., Morgan, K., and Wirthlin, M., "Improving FPGA Design Robustness with Partial TMR," in "Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE

- International,” , 2006, pp. 226 –232.
- [23] Flatley, T., “Advanced hybrid on-board science data processor - SpaceCube 2.0,” , 2010. Earth Science Technology Forum.
- [24] Jacobs, A., Cieslewski, G., Reardon, C., and George, A. D., “Multiparadigm Computing for Space-Based Synthetic Aperture Radar,” in “ERSA,” , 2008, pp. 146–152.
- [25] Coulton, P. and Carline, D., “An SDR inspired design for the FPGA implementation of 802.11a baseband system,” in “Consumer Electronics, 2004 IEEE International Symposium on,” , 2004, pp. 470 – 475.
- [26] Golub, G. H. and Loan, C. F. V., *Matrix Computations*, The Johns Hopkins University Press, 3rd ed., 1996.
- [27] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical recipes in C (2nd ed.): the art of scientific computing*, Cambridge University Press, New York, NY, USA, 1992.
- [28] Amira, A., Bouridane, A., and Milligan, P., “Accelerating Matrix Product on Reconfigurable Hardware for Signal Processing,” in “Proceedings of the 11th International Conference on Field-Programmable Logic and Applications,” Springer-Verlag, London, UK, FPL ’01, 2001, pp. 101–111.
- [29] Jacobs, A., Conger, C., and George, A., “Multiparadigm Space Processing for Hyperspectral Imaging,” in “Aerospace Conference, 2008 IEEE,” , 2008, pp. 1 –11.
- [30] Chang, C.-I., Ren, H., and Chiang, S.-S., “Real-time processing algorithms for target detection and classification in hyperspectral imagery,” *Geoscience and Remote Sensing, IEEE Transactions on*, Vol. 39, No. 4, 2001, pp. 760 –768.