Injecting FPGA Configuration Faults in Parallel

Shane T. Fleming, David B. Thomas
Dept. of Electrical and Electronic Engineering, Imperial College London, UK
{sf306, dt10}@ic.ac.uk

***Abstract*—When using SRAM-based FPGA devices in safety-critical applications testing against bitflips in the device configuration memory is essential. Often such tests are achieved by corrupting configuration memory bits of a running device, but this has many scalability, reliability, and flexibility challenges. In this paper, we present a framework and a concrete implementation of a parallel fault injection cluster that addresses these challenges. Scalability is addressed by using multiple identical FPGA devices, each testing a different region in parallel. Reliability is addressed by using reconfigurable system-on-chip devices, that are isolated from each other. Flexibility is addressed by using a pending-commit structure, that continually checkpoints the overall experiment and allows elastic scaling. We test and showcase our approach by exhaustively flipping every bit in the configuration memory of the CHStone benchmark suite and a VivadoHLS generated k-means clustering image processing application. Our results show that: linear scaling is possible as the number of devices increases; the majority of error inducing bitflips in the k-means application do not significantly impact the output; and that the Xilinx Essential bits tool may miss some bits that can induce errors.**

I. Introduction

A high-energy particle striking the configuration memory of an SRAM-based FPGA device can flip bits of the configured circuit causing an unwanted localised reconfiguration. For this reason when using FPGA devices in safety critical applications designs must be robust to such errors and thoroughly tested against them.

Exhaustive in-hardware fault testing, where bits of a live configuration are flipped from their intended state, is a slow process. For each bitflip the circuit needs to be re-executed, the output stored, and the device returned to its original state. Returning the device to it's original state usually just requires flipping the bit back and resetting the circuit. But there are cases where a full reconfiguration is required and occasionally the device will lock-up so hard a complete power cycle is required incurring significant extra time [1]. These problems all compound the fact that in-hardware testing is inherently slow, so as the size of the circuit under test (CUT) increases, these problems quickly result in exhaustive testing becoming intractable.

The solution to this problem has generally been to only flip randomly sampled configuration RAM (CRAM) bits[2]; however, as devices and applications become more complex and their use becomes more widely adopted the importance of exhaustive testing increases. We believe more extensive fault testing is necessary for the following reasons:

1 **More complex devices**: It has become commonplace for FPGA devices to contain hardened ARM cores, which share memory, complex buses, and configuration bits with applications in the FPGA fabric.

2 **More complex applications**: High-level synthesis (HLS) is allowing for more complex applications to be developed with more dynamic control-flow structures. This has lead to the development of increasingly irregular applications with more statically unpredictable behaviours.

3 **Widespread adoption of devices**: Machine learning has driven the increasing adoption of FPGA devices in areas such as information retrieval and self-driving cars. With more devices in use the chance of an error being observed increases, which is especially important in safety critical applications.

Rare catastrophic faults may be unlikely but they will eventually occur, especially as the number of devices in use increases 3. Also as both the complexity of the applications 2 and the device 1 increases, interactions between components become more intricate. This increases the potential chance of a catastrophic fault as faults in one component can have non-obvious effects on other components. It is for these reasons that we believe that more emphasis should be placed on more exhaustive testing instead of random sampling.

In this paper we present an approach for building a scalable and reliable fault injection testing platform using reconfigurable SoC devices. We also present an open-source low-cost concrete implementation of our approach, called **ParFlip**, that uses multiple Digilent Zedboards Xilinx Zynq based devices in parallel. ParFlip fully automates fault injection experiments – the user provides a bitstream and a list of CRAM bits to be tested. A central management system then performs the following: it breaks the problem into smaller chunks; it allocates chunks to the multiple currently available FPGA devices; it tracks the health of each device as testing is performed; it recovers any devices that becomes unstable; it takes unstable devices out of service if they become persistently unstable; and it collects results in a transactional fashion, where results are only committed once they are stable. Our approach is able to linearly reduce testing time with the number of FPGA devices added to the system, this is demonstrated in Section V.

This paper makes the following contributions:

1) An approach for parallel and reliable in-hardware fault injection using multiple reconfigurable SoC devices.

2) ParFlip, a concrete implementation of our approach using multiple Digilent Zedboards (Xilinx Zynq based) boards.

3) Results for an exhaustive injection campaign on a k-means image clustering application generated via VivadoHLS, where every output image produced by an error inducing bitflip was saved in a `2.2` GB data set, to enable others to study the effects of CRAM faults.

4) Analysis of the k-means clustering dataset where the resulting error output images were analysed using structural similarity index, enabling us to identify faults related to addressing, interface logic, and to categorise the most extreme faults.

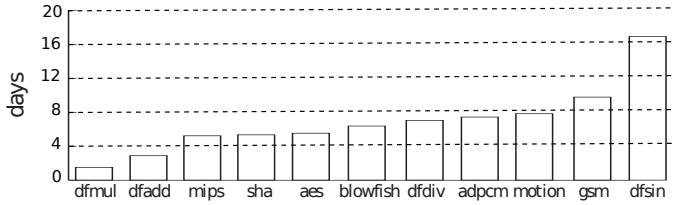

Fig. 1. Testing times for the CHStone benchmark suite on a single Zedboard device

5) The identification of statically predictable bits in the configuration bitstream of Zynq devices that when flipped cause entire system lockup, enabling them to be filtered.

In our concrete implementation we also use ParFlip to perform verification experiments on the Xilinx Essential Bits tool, a tool that can be used to reduce the total number of bits required for testing. Our results hint that there may be CRAM bits that cause observable errors that are not included in the essential bits report, i.e. there are non-essential bits that when flipped induce an observable unintended outcome (wrong result or wrong execution time).

II. Motivation

To test circuits against single static CRAM faults exhaustively on a Zynq (ZC7020) requires the injection of up to 25,697,632 bits. We define a **single static** CRAM error as an unwanted bitflip of one CRAM bit that is present before the circuit has started it's execution, *i.e. it is present at* $t = 0$.

In general not all 25 million bits will not need be flipped, as only a subset are used to configure the circuit under test (**CUT**) – these bits are referred to as the **essential bits** of the CUT, where we have N_E total essential bits for our design. The time per bitflip is the sum of: T_I, the time to flip a CRAM bit; T_E, the time required to execute the circuit; T_C, the time to compare the output; and T_R, the time to recover the device back to its original state. The total time to test every essential bit, T, for a CUT is given by:

$$T = \sum_{j}^{N_E} T_I^j + T_E^j + T_C^j + T_R^j \tag{1}$$

If the circuit has a large number of essential bits, or requires considerable time to execute, then exhaustively injecting errors into the CRAM (i.e. flipping every essential bit) can take a long time. For example, we estimate the `dfsin` circuit of the CHStone HLS benchmark suite [3] would take over 16 days for complete testing on a single device with fully automated testing.

Unfortunately, the time for each bitflip is not the only scalability issue. Sometimes flipped bits cannot be flipped back easily, which means they take much more time to recover from; we refer to such bits as **stuck bits** and **danger bits**. Recovering from a non-stuck or non-danger bit is fast, as it is just the time taken to flip the bit back to its original configuration, T_I. However, recovery from a stuck bit takes longer as a reconfiguration of the entire CRAM with the CUT is required, which requires much more additional time (750ms for a Xilinx ZC7020 device from within the embedded Linux). Danger bits are somewhat worse again as they cause the entire system to lock up when flipped, freezing the processing system completely. The only way to recover from a danger bitflip is to perform a full power cycle and reconfiguration, which can take in the order of minutes, particularly when there is a software component running on the device.

If we take the probability of a stuck bit occurring as P_S, the probability of a danger bit occurring as P_D, the time to recover from a stuck bit as T_{RS}, and the time to recover from a danger bit as T_{RD} we get the following equation for T_R:

$$T_R = (1 - (P_S + P_D))T_I + P_S T_{RS} + P_D T_{RD} \tag{2}$$

Many danger bits may be encountered over the course of a fault-injection campaign – on average we found that $\approx 5\%$ of critical bits are danger bits for a ZC7020 device – dramatically increasing testing time. Luckily most danger bits can be filtered statically from the circuit, however, they cannot be perfectly filtered as some critical bits may be removed that are non-danger bits. This means that filtering danger bits comes at the sacrifice of exhaustive testing coverage and we found on average $\approx 0.3\%$ of non-danger critical bits will be filtered out along with the danger bits (ZC7020 device).

Figure 1 shows estimated testing time for the CHStone benchmark suite in days on a single Zedboard device tested in a non-naive fashion with automated experiment management and recovery. These results were obtained by bit-flipping a small fraction ($\approx 0.1\%$) of each circuit's essential bits, selecting bits in a uniformly random fashion, and extrapolating to the total number of essential bits for each circuit. Summing the testing times in Figure 1 we estimate that it would take a total of 11 weeks to exhaustively test every essential bit (single static error) of the CHStone benchmark suite.

The results in Figure 1 are for CUTs with no protection, but fault injection is often used to test a fault protection strategy, which generally have considerable overheads. Such techniques usually double or triple the logic of the circuit and thus increase the size of the circuit and testing time, they also often increase the critical path of CUT increasing it's execution time further increasing testing time.

We identify four major challenges to performing in-hardware FPGA fault injection:
[Scalability]: Complete coverage of the CRAM bits is unfeasible for large circuits.
[Reliability]: Injecting faults into a system can often result in the system becoming unstable and unpredictable.
[Checkpointing]: Experiments that run for long periods of time need to be interruptible and should be able to be paused and resumed at will.
[Flexibility]: As experiments often require a long time they should dynamically scale as hardware resources become available or unavailable. [1]

Simulation-based approaches [4][5] are able to address all the challenges outlined above: scalability, is addressed by instantiating multiple parallel instances of the problem; reliability is not a concern, as all unstable behaviour is contained within the simulation environment; checkpointing is easy, as the complete state of the system can be known at all times; and flexibility can be addressed by executing it in an elastic fashion. However, such simulation-based approaches rely on injecting errors by corrupting signals in the register transfer

[1]The devices used in this work were shared with an undergraduate lab

level (**RTL**) description of the circuit, and then performing an RTL simulation on the faulty description. Operating at the RTL means that they fail to capture any device-level information and since $\approx$ 90% of the configuration bits in modern FPGA devices are used for routing, a CRAM fault is far more likely to occur in a regions not tested by such approaches.

To accurately simulate configuration bits at the device-level requires in-hardware fault injection, where addressing the challenges outlined above is considerably harder. Existing work addresses the scalability issue by either using random sampling of the configuration bits, sacrificing fault coverage for reduced testing time [2][6], or by trying to increase the rate at which errors can be injected into the system [7][8]. There are two main methods for injecting errors directly into the CRAM of an FPGA device: internally or externally. External injection, typically via a JTAG connection [7][8], is desirable as the device is decoupled from the fault injection mechanism, making it easier to deal with reliability challenges. However, external injection is considerably slower than internal injection methods. For internal injection methods, the fault injection mechanism is located on the device and can be: a custom circuit running in the FPGA logic [6]; code running on a soft-core processor in the FPGA logic [9]; running on the in-package ARM cores of a reconfigurable SoC device. While internal injection methods are faster they come at the cost of being more difficult to manage, as injections that cause the CUT to become unstable may also directly or indirectly affect the fault injection mechanism.

A. Our Approach

Our approach uses an internal injection method per node. We address the **[Scalability]** challenge by using multiple parallel identical FPGA devices, each testing a different portion of the circuit at the same time. The other challenges are addressed by the use of reconfigurable SoC devices that contain an FPGA, where the CUT resides, and a small ARM core, that can be used as an in-built local host processor for managing the local experiment. This local host enables each device to run in isolation from each other and supports networking infrastructure enabling it to connect to a cluster-management machine, called the SoC-controller, via a local area network (Section IV).

Using networking in this way allows nodes to be seamlessly added and registered as available with the SoC-controller, addressing the **[Flexibility]** challenge. When the SoC-controller allocates a chunk of the overall problem to each node in the system, that chunk is moved into a pending state until the node has indicated that it has completed where it is then committed. If the node has become unresponsive the chunk is split in two and returned to the work-pool (Section III-A). The isolation between nodes and the pending-commit structure addresses the **[Reliability]** challenge. If a single node becomes unstable it will not affect the overall state of the experiment; the SoC-controller can detect that the node has become unstable and attempt to recover it, or if it can't recover it, it takes it out of service. The pending-commit structure also addresses the **[Checkpointing]** challenge as the entire system can be stopped at any point and resumed from the last committed results.

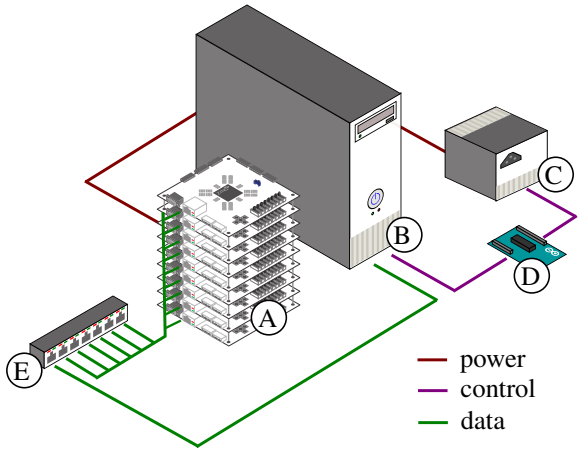

Fig. 2. The physical architecture of ParFlip showing how the cluster of reconfigurable SoC devices is connected to a central desktop.

III. Details of Our Approach

Historically, before reconfigurable SoC devices, building such a cluster would have been much more challenging, as each FPGA device in the cluster would need to be connected to a host (dekstop/server) machine. These host machines were limited to the number of FPGA devices that they could host, meaning that as the size of the cluster increased there would be a linear increase in the number of hosts, resulting in extra complexity in their management. Having a lightweight host built into each of the reconfigurable SoC device enables the cluster to be scaled in a easy and cheap manner.

Figure 2 depicts an example implementation of our approach, called ParFlip, which consists of:

- (A) A number of reconfigurable SoC based **nodes**, in this case Digilent Zedboards (XC7Z020-CLG484) and an SD card.
- (B) A cluster management desktop machine, called the **SoC-controller**.
- (C) Power supplies for the nodes, in this case a single ATX.
- (D) An Arduino (microcontroller development board) for controlling the resettable segments, *i.e.* (C).
- (E) A network switch.

Experiments are managed and coordinated by the SoC-controller (B), which is a central server such as a desktop machine. A network switch (E) connects a rack of reconfigurable SoC devices (nodes) (A) to the SoC-controller. Each node contains an identical FPGA device and they all run Ubuntu (version 12.4) Linux operating system on an ARM-based processing system from an SD card. For all nodes, ssh is enabled and is used to transfer data and run experiment scripts.

Faults injected into the CRAM can sometimes result in failure in the processing system, causing complete system lock-up. The only way to recover from such errors is to power cycle the board, restarting the OS and reconfiguring the FPGA fabric. Power cycling is achieved by assigning nodes to **resettable segments**, a set of nodes that must be power cycled together. Each of these resettable segments can be reset independently

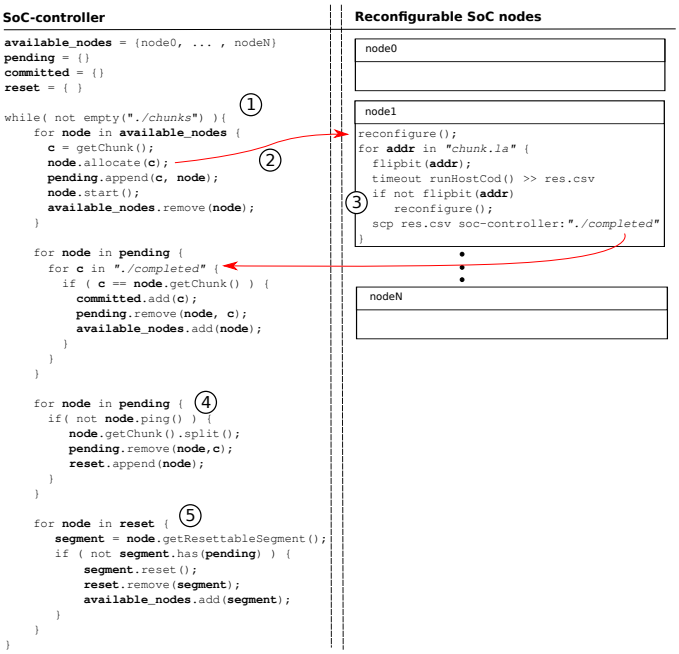

Fig. 3. Rough overview for the ParFlip management software of the soc-controller and nodes

and is controlled via an Arduino micro-controller connected to the SoC-controller via a serial connection. The SoC-controller continually monitors the health of the nodes and if required will issue a reset command to a resettable segment, power cycling all the nodes in a segment.

The granularity of the resettable segments impacts the overall scalability of the system. If there is only one large resettable segment for all nodes, then if *any* node encounters a danger bit a synchronisation of the *entire* cluster must occur before the reset can occur. This synchronisation is a problem – even if all chunks allocated to each node are the same size they will complete at very different rates, resulting in many nodes being idle. Two factors contribute/explain why nodes complete at different rates:

1) for each chunk the probability of finding a stuck bit, P_S, can vary, which has a large influence on chunk testing time (Equation 2).
2) depending on the region of the circuit there may be a lot of errors that cause the circuit to timeout, *e.g.* the error causes the circuit to no longer terminate.

If the resettable segments are very fine-grained, say one resettable segment per node, then each node can run asynchronously, greatly improving scalability.

A. Management Software

Experiment management for each campaign can be split into two main parts: the SoC-controller management software running on the desktop machine, and the node management software running on the ARM of each node. Figure 3 shows how the soc-controller and each node interact with each other and the tasks performed by each portion:

① **Chunking the problem:** This stage splits the overall problem into smaller chunks of addresses, each of which will be allocated to a node and tested. All chunks are stored in a (`./chunks`) directory where they await allocation to a node.

② **Allocating chunks to nodes:** A chunk is selected from the `./chunks` directory allocated to a node and moved to a `./pending` directory. Local experiment scripts are then rendered and, along with a copy of the chunk, are sent to the appropriate node over ssh. A command is sent to each allocated node indicating that it should start evaluating it's chunk. A list of currently running nodes is maintained.

③ **Evaluating the chunks:** On each node the experiment script assigned to that node starts by reconfiguring the FPGA logic to load the CUT. The script then iterates through each CRAM address (`addr`) in the allocated chunk file, sending the commands to flip the relevant bit. It then executes the software host code, defined by the user, with a user configurable timeout in case the hardware gets stuck (recorded as a `TIMEOUT` fault). Once the host code returns the result of the test, the experiment script attempts to repair the flipped bit by flipping it back to its original state and resetting the FPGA circuit. If the bit repair fails, we have encountered a stuck bit and a full device reconfiguration is attempted to fix the bit. If the reconfiguration fails, then the entire SoC device is restarted, resuming from where it left off. After all addresses in the chunk have been processed the results file is transferred back to the controller over ssh, which is also used to indicate it has completed processing that chunk and that they should be committed.

④ **Health monitoring:** While the nodes are processing chunks, the SoC-controller checks to see if any of the nodes have completed their experiments or if they have become unresponsive. Determining if any nodes have become unresponsive is achieved by issuing a ping request. If no response to the ping is received, then the node is assumed to have crashed and is added to a list of dead nodes.

⑤ **Power cycling:** If the health monitoring indicates to the SoC-controller that a node has become unresponsive then it will try to power cycle it. To do this the SoC-controller examines the resettable segment that the unresponsive node belongs to. Once all the other nodes in that segment have either become available for allocation or unresponsive it issues the command to reset that segment. It then keeps checking to ensure that all the nodes have rebooted successfully, if that is not the case, it keeps rebooting the cluster until all nodes are stable. If it appears that one node is repeatedly not able to boot (generally due to SDCard corruption) then it is taken out of the list of available nodes and logically removed from its resettable segment.

IV. Concrete Implementation

We have constructed a concrete implementation of our approach, called ParFlip, which is the example shown in Figure 2. In this implementation there are seven Digilent Zedboard nodes each containing a Xilinx Zynq device (XC7Z020-CLG484) that is running the Ubuntu (version 12.4) Linux-based operating system on its ARM core.

There is only one resettable segment that contains all nodes, future work aims to improve this by having a resettable segment per node. A single desktop ATX power supply unit powers all nodes (C); it was modified so that its `PS_ON` pin, which when pulse power cycled all nodes, is connected to the Arduino (D) controlled by the SoC-controller.

In this implementation, as we are using Xilinx devices, there are some Xilinx specific details that will be outlined. In particular: how the FPGA design needs to be modified (Section IV-A); and the use of the Xilinx Essential Bits report (Section IV-B).

A. Modifying the FPGA design

For Xilinx devices hardware fault injection is achieved by using an IP-core provided by Xilinx called the Soft Error Mitigation (**SEM**) core [1], which connects to the internal configuration port of the CRAM (**ICAP**) allowing addressable CRAM bits to be flipped from their original state. Injections can also be performed by software running on the ARM based processing system through the processor configuration port (**PCAP**), however, this is not explored in this work, but will be explored in future as we believe it can be used to increase the fault injection rate of the system [10].

To perform error injection experiments we built an AXI based interface for the SEM core, where memory mapped AXI slave registers are used to accept a CRAM address for the target CRAM bit to flip and to control the core. This enables software to pass input arguments, get status notifications, and start the execution of the CUT through AXI memory mapped registers.

CRAM bits related to the SEM are highly sensitive to bit flips, with a high proportion of them being danger or stuck bits, delaying testing time[6]. To avoid injecting errors into the SEM and other infrastructure logic we contained our design in a **pblock** which is a facility provided by Xilinx that we could use to focus our fault injection experiments only on the CUT.

B. Essential bits

In an effort to address the scalability challenge Xilinx provides a tool to extract the *essential bits* of a design. This tool returns a subset of the CRAM bits that *may* influence the device configuration or pblock, eliminating only bits which will have no impact on the output.

More formally, let B be the set of all bits in the CRAM. For a given design, the set of essential bits, E, reported by the tool must satisfy $E \subseteq B$. However, not all bits in E will induce an observable error when flipped. We define an observable error as one where either the output of the circuit is different than the expected output (assuming deterministic output) or the number of execution cycles required for the circuit was different to what was expected (assuming a deterministic execution time). The set of bits that when flipped induce an observable error are known as the set of critical bits, C, which must satisfy $C \subseteq E$ [11]. The set C is generally much smaller than E and varies for each application: from our experiments between 0.5% – 39.25% of essential bits are critical.

The set of non-essential bits, $N = B - E$, is the set of bits that when flipped should not produce any observable error.

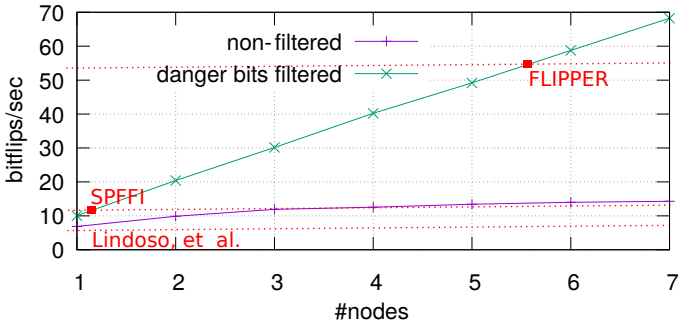

Fig. 4. Results to show how exhaustive CRAM injection for the `dfmul` benchmark scales as the number of SoCs increases and how filtering out the problem bits improves performance.

From this we can state the following constraint must *not* be true: $CN =$.

However, results from Section VIII suggest that this may not hold for the current essential bits provided by Xilinx, as we have found a number of bits in N that are also in C.

V. Scalability Results

Figure 4 shows the average bitflips/sec for the exhaustive static single bit testing of the `dfmul` CHStone benchmark. There are two lines, *"non-filtered"*, where the danger bits of the circuit were not filtered, and *"danger bits filtered"* where a static prediction of danger bits was used to attempt to filter. Without filtering the fault rate quickly saturates, with little improvement in performance as the number of nodes is increased. However, applying filtering results in a linear increase in the fault injection rate as the number of nodes is increased.

There are two reasons that explain this. Firstly, in this version there is only one resettable segment causing all nodes to synchronise and power-cycle whenever a single one becomes unstable due to a danger bit. Secondly, we've found that booting Linux on the Zedboard occasionally causes a kernel panic, which forces the SoC-controller to issue multiple reset requests until all nodes boot. This means that as we increase the number of nodes we also increase the chances that at least one node will encounter a danger bit and we also decrease the probability that all nodes will boot. We can model this effect on average danger bit recovery time, T_{RD}, using with the following:

$$T_{RD} = \frac{1}{(P_B)^N}(T_{BOOT}) + T_{SU} \tag{3}$$

Where: P_B, is the probability of a node booting successfully; N, is the number of nodes in the resettable segment; T_{BOOT}, is the time it takes for the node to boot; and T_{SU}, is the time it takes to setup the experiment scripts on each nodes. There are two main ways that we can improve the scalability of the system: reduce the number of danger bits encountered so that there is less need to power cycle the devices, or decrease the number of nodes in each resettable segment. Another potential area of improvement is to use a simpler more-lightweight operating system on each node, such as FreeRTOS, that can reduce T_{BOOT}.

bench	#esn	#crit		#doe	#ete
aes	1,764,569	203,009	*(11.50%)*	187,592	15,419
blowfish	1,720,603	587,266	*(34.13%)*	578,107	9,160
motion	2,024,568	10,074	*(0.50%)*	4,569	5,507
sha	1,603,211	403,217	*(25.1%)*	398,182	5,036
gsm	2,268,891	432,485	*(19.06%)*	111,728	320,759
mips	1,256,348	33,182	*(2.64%)*	9,794	23,389
dfsin	4,167,144	436,840	*(10.48%)*	119,552	317,289
dfdiv	1,653,102	31,374	*(1.90%)*	7,855	23,521
dfmul	399,528	10,926	*(2.73%)*	9,190	1,738
dfadd	993,256	40,299	*(4.06%)*	30,768	9,533
adpcm	2,002,204	62,928	*(3.14%)*	44,378	18,553

Fig. 5. Table showing the absolute error results for the CHStone benchmark where #esn is the number of essential bits, #crit is the number of critical bits, #doe is the number of critical bits that had observable errors on the final result, and #ete is the number of errors where the execution took the incorrect number of cycles.

The danger bit filter was constructed by analysing the results from exhaustively injecting all essential bits of each benchmark in the CHStone suite (Section VI). For each circuit the bits that consistently caused the system to lock up across all benchmarks were identified. On Zynq XC7020 devices the configuration memory is made up of: 7950 frames, each 101 words in length, where each word is 32 bits. In the CHStone results we found that the majority of danger bits occur in word-50 at a regular intervals of frames. Our filter works by simply stripping all word-50 CRAM addresses from the problem before testing.

Recent work by Wolf et al [12] uses fuzz testing to reverse engineer the bitstream of Artix-7 devices. Their results indicate that word-50 is special as it is used to manage the clock-tree of the device. Since the Artix-7 programmable logic is the same technology used for the logic portion of the XC7Z020, we believe that word-50 is likely disrupting the clock of the ARM management host, causing the system to lock-up.

By applying our filter to the essential-bit file of the circuit we were able to increase the fault injection rate as the number of nodes increased, seen in Figure 4. However, our filter is not perfect and will occasionally remove non-danger critical bits from the input testing file – from the CHStone results on average 0.3% of non-danger critical bits are filtered.

Figure 1 also shows a few comparison points for this work, all injection rate comparisons are for a single FPGA device, show their peak rate, and were collected from published results. FLIPPER [13] is a framework that injects faults via a JTAG interfaces that is able to achieve a high injection rate, however, it does this through accumulating bitflips until an erroneous effect is observed – meaning it cannot consider the effect of bitflips in isolation. Work by Lindoso et al., [6] is closest to our approach as they used the same FPGA technology and the Xilinx SEM core for fault injection. They achieve a comparable injection rate as us for a single device. FT-UNSHADES [14] is another fault injection framework that uses the Xilinx SelectMAP configuration interface, we believe that this approach could be combined with our framework to further increase injection rate.

VI. CHStone Results

Exhaustive fault injection was performed on the CHStone benchmark suite with circuits generated via the HLS tool LegUp [15]. Each of the CHStone benchmarks are self-stimulating and self checking, which means all the input vectors are contained within the bitstream. Figure 5 shows the results of an exhaustive static fault injection campaign on these benchmarks. From the results we can see that there was a large variance in the number of essential bits that were critical to the design, which we believe is due to how coverage of the circuit the input vectors had. There is also a good mix of data-flow heavy applications, where errors were mainly only effecting the output data but not the circuits execution time, such as `blowfish` or `sha`, and applications that are control-flow heavy where the majority of errors influence the execution time of the circuit, such as `mips` or `gsm`.

VII. Case Study: K-Means Image Clustering

In this section we will explore if it is possible to further extract information from the results by considering how the output is corrupted. We will examine an image processing case study and explore how errors in the circuit alter the returned images.

K-means clustering is often used as part of an image processing pipeline in domains where bitflips might be a concern, such as self-driving vehicles, or satellites [16]. In this section we will take a HLS version of the k-means clustering kernel, generate it with VivadoHLS, and perform an exhaustive static fault injection campaign using ParFlip. The generated hardware contains both an AXI slave, for command and control, and an AXI master interface, for bulk data transfer. Input and output data is stored in globally accessible shared DDR, this means that execution time is non-deterministic as there can be variations in access time. However, the numerical output is deterministic, enabling us to check for faults by comparing against a golden output.

For every critical bit encountered we will store the output for offline analysis. We want to analyse these faulty images for two reasons: to determine the qualitative severity of the error; and to try and identify sensitive regions of the bitstream. We anticipate that there might be errors that may for instance, shift the image by one pixel position. In such cases we would not want the analysis to return a poor score, so we decided to use structural-similarity index (SSIM) [17] as our metric for analysing the output images.

Figure 6 shows the frequency of errors within a given SSIM score bin. Along the x-axis the average image for each SSIM bin is also displayed. It can be seen that qualitatively, the threshold where the images start to become unrecognisable from the golden output is when the SSIM score is below 0.4. However, most errors have a SSIM score much higher than this, with 76.8% of all critical bits having a SSIM score $>$0.4 and 23.2% being below $\leq$ 0.4.

From these results we can make a case for fault protection strategies that only attempt to selectively protect the circuit against the most extreme faults [18]. For instance, Xilinx devices have built in ECC for checking if faults have occurred in any given frame of the CRAM. The latency at which a fault can be detected and for recovery to start is the time taken to scan through all the ECC portions of each frame. The data resulting from such an analysis could be used with the built in ECC checking of Xilinx devices to only inspect/repair frames that contain bits in the SSIM$\leq$0.4 range. For the k-means

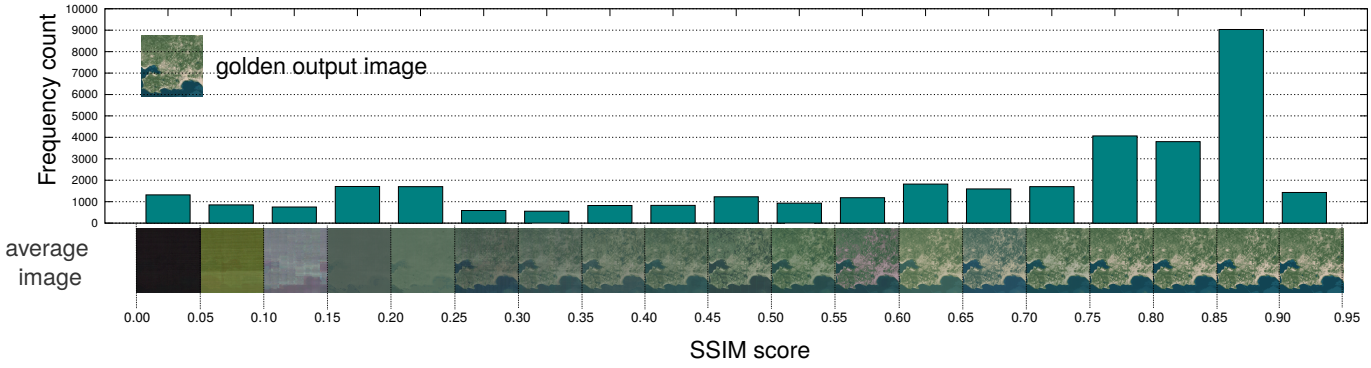

Fig. 6. Average of all images within a given ssim range

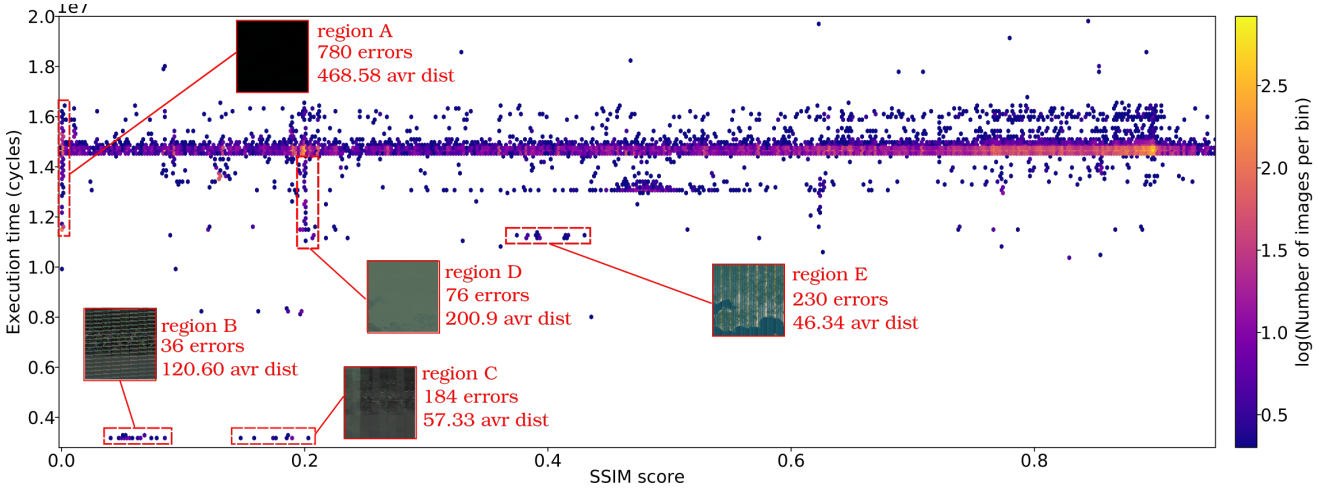

Fig. 7. 2D Histogram for the execution time and SSIM score for every "faulty" image returned

application an ECC based protection strategy would only need to check for errors in 1004 frames instead of 6590 frames, reducing the latency of such a fault detection approach.

As well as the SSIM score, deviations in the circuits execution time can also be used to further analyse outcome of critical bits. Figure 7 shows a 2D histogram with the execution time along the y-axis and SSIM score along the x-axis. The correct execution time can be seen as the most common occurrence at 1.47×10^7 cycles (100MHz clock). We have found that the clusters of faulty images in Figure 7 tend to have highly similar characteristics. For a selection of these clusters, labelled A - E, we have attempted to speculate their origin by taking into account the execution time, the average image, and euclidean CRAM space (frame $\times$ word $\times$ bit):

[Region A]: All output is black; there are plenty of causes for such an error, for example an error in the routing logic that transfers the end result to main memory. These errors are widely distributed across the entire CRAM space and have a large variation in execution time as they can occur in many different places.

[Region B]: The execution time in this region corresponds with the time expected if the kernel returned straight away (i.e. the done signal instantly went high). This is likely an error in the interface logic of the VivadoHLS generated hardware, which explains why the total number of errors is low and the average distance is low. The average image is just random noise from shared memory.

[Region C]: We believe that this region is similar to region B, however, in this case there was a partial returned image left over in the shared memory from a previous run giving it a slightly higher SSIM score. In future experiments we will ensure that all shared memory is zeroed before each test.

[Region D]: One or two large clusters in the output image – this error could occur in many different control-flow regions of the application where there is a loop over the total number of current clusters. Hence there is a high variation in execution time, and reasonably large CRAM distance.

[Region E]: Error in addressing logic – parts of the image are skipped and the amount skipped doubles with consecutive errors. This is likely hitting an index in the read or write to main memory, hence it has a low average distance.

benchmark	Frame	Word	Bit	Result	Cycles
dotproduct	5851	85	12	182400	1208
	5715	79	12	182401	604
	5578	79	13	91100	304
	5679	79	12	182600	604
dfmul	5850	75	13	40	217
	5851	75	12	21	217
	5715	18	12	12	217
	5815	22	12	12	217
	5714	18	13	12	217

Fig. 8. Non-essential bits that caused observable errors, in the `dotproduct` benchmark (expected result:**182400**, expected cycles: **604**), and `dfmul` (expected result:**20** , expected cycles:**217**) benchmark. Essential bits generated via Vivado 2017.4

VIII. Testing Essential Bits

The essential bits report generated by the Xilinx tools should be the super-set of all critical bits in the design. To test this we performed an injection campaign into all the bits that the Xilinx tool claim are non-essential for two benchmarks: a floating point dotproduct with deterministic cycles and output and the dfmul CHStone benchmark, both generated via LegUp.

For each benchmark the set of non-essential bits was calculated by calculating all bits not mentioned in the essential bits file. ParFlip was then used to exhaustively check to see if any of the non-essential bits produced an error. For each benchmark a small number of bits $\approx 0.02\%$ were found to cause an observable error in the output of the circuit, all of which seem to occur in either bit 12 or bit 13 of a CRAM word. We performed these tests on Vivado versions 2015.3 and 2017.4. Figure 8 shows the locations and results for each of the missed critical bits that we found for the `dotproduct` and `dfmul` benchmarks.

IX. Conclusion

Performing in-hardware fault injection can take a long time and is challenging to automate. This paper presents a framework which reduces testing time by using multiple identical reconfigurable SoCs in parallel. We demonstrated that with our approach it is possible to scale up the injection rate linearly with the number of nodes, in a flexible and reliable fashion. Exhaustive fault campaigns were performed on both the CHstone benchmark suite and a k-means image processing application. For the k-means application we learnt that: the majority of critical bits produce images that do not have a noticeably different output; and that if we consider both the SSIM score of the corrupted images and the execution time we can identify faults related to the circuit interface and memory addressing logic. By exhaustively exploring the CRAM of the Zynq device we were also able to learn two things: injecting errors into word-50 causes the complete system to lock-up, we speculate that this is because word-50 is related to the clock of the ARM processing system; and that the Xilinx Essential Bits report may have missed some CRAM bits that do induce errors in the output.

In future work we plan to explore how fine-grain resettable segments will allow for better coverage and scalability of the system. We would also like to explore ways to correlate the output errors of the k-means application to the lines of the input code to investigate instruction-level protection strategies of HLS generated circuits.

References

[1] *Soft Error Mitigation Controller*, v4.1 ed., Product Guide (PG036), Xilinx Corp, April 2018.

[2] A. J. Sánchez-Clemente, L. Entrena, and M. García-Valderas, "Partial TMR in FPGAs using approximate logic circuits," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2233–2240, 2016.

[3] Y. Hara-Azumi, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[4] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for fpgas," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, 2004.

[5] *Introduction to SEUs Simulation Tool SST*, 2nd ed., Tutorial, ARIES Research Center/ESA, April 2017.

[6] A. Lindoso, L. Entrena, M. García-Valderas, and L. Parra, "A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 374–381, 2017.

[7] A. Gruwell, P. Zabriskie, and M. Wirthlin, "High-speed programmable FPGA configuration through JTAG," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1–4.

[8] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop fpga-based fault injection technique," *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, 2014.

[9] L. Gong, T. Wu, N. T. Nguyen, D. Agiakatsikas, Z. Zhao, E. Cetin, and O. Diessel, "A programmable configuration controller for fault-tolerant applications," in *Field-Programmable Technology (FPT), 2016 International Conference on*. IEEE, 2016, pp. 117–124.

[10] A. Stoddard, A. Gruwell, P. Zabriskie, and M. Wirthlin, "High-speed PCAP configuration scrubbing on Zynq-7000 all programmable socs," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1–8.

[11] R. Le, "Soft error mitigation using prioritized essential bits," *Xilinx XAPP538 (v1. 0)*, 2012.

[12] "Project X-ray - Xilinx series 7 bitstream documentation," Accessed 2018-03-28. URL: https://github.com/SymbiFlow/prjxray, Tech. Rep., 2018.

[13] M. Alderighi, S. D'Angelo, F. Casini, G. Sorrenti, D. M. Codinachs, and S. Davin, "The FLIPPER fault injection platform: Experiences and knowledge from a ten-year project," in *ARCS 2017; 30th International Conference on Architecture of Computing Systems; Proceedings of*. VDE, 2017, pp. 1–8.

[14] J. Mogollon, H. Guzman-Miranda, J. Napoles, J. Barrientos, and M. Aguirre, "Ftunshades2: A novel platform for early evaluation of robustness against see," in *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*. IEEE, 2011, pp. 169–174.

[15] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[16] S. T. Fleming, D. B. Thomas, and F. Winterstein, "A power-aware adaptive FDIR framework using heterogeneous system-on-chip modules," in *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 2016, pp. 75–90.

[17] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[18] S. T. Fleming and D. B. Thomas, "StitchUp: Automatic control flow protection for high level synthesis circuits," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.