ABSTRACT

The ongoing miniaturization of digital circuits makes them more and more susceptible to faults which also complicates the design of fault tolerant systems. In this context fault injection plays an important role in the process of fault tolerance validation. As a result many fault injection tools have emerged during the last decade. However these tools only operate on specific domains and can therefore be referred to as hardware- or software-, simulation- or emulation based techniques.

In this paper we present FuSE, a single fault injection tool which covers multiple domains as well as different fault injection purposes. FuSE has been designed for usage with the SEmulator®- an FPGA-based hardware accelerator. The created tool set has been fully automated for the fault injection process and only requires a VHDL description and a testbench of the circuit under test. FuSE can then perform fault injection experiments with a diagnostic resolution that is known from simulation-based approaches, but at a speed that even handles long running experiments with ease.

1. INTRODUCTION

The increasing complexity of digital circuits makes their development more and more challenging and tricky. Especially the verification process becomes a crucial task. Simulations provide full access to internal signals but they can only cover short periods of real time (few milliseconds). A prototype implemented in an FPGA allows to observe the device for longer periods, but yields an extremely low visibility of internal procedures, which aggravates the debugging in the case of a malfunction. Furthermore, the pure observation does not suffice yet because the ongoing miniaturization of semiconductor structures makes digital circuits more and more error prone. As a consequence fault tolerance mechanisms are required to implement reliable systems. To verify these mechanisms, waiting for the occurrence of an actual physical fault would be impractical. Instead, a dedicated manipulation of the circuit has to be applied in order to artificially produce the desired fault. In addition to the trade-off between performance and visibility, a fault injection strategy has to find a balance between intrusiveness and reproducibility.

In this paper we present FuSE (Fault injection using SEmulation), a fault injection tool which combines the performance of a prototype implemented in hardware and the flexibility as well as the visibility of a standard HDL simulation. Due to the fact that FuSE is based on the SEmulator® engine, faults can be injected either in an HDL model running in the simulator or in a netlist downloaded to an FPGA. The switching between these two modes is transparent, allowing to observe the propagation of a fault inside the circuit in detail on the one hand and to perform millions of fault injection experiments within a reasonable amount of time on the other hand.

The remaining paper is organized as follows: In Section 2 we provide an overview concerning fault injection, highlighting the properties of different fault injection techniques. The FuSE concept is explained in Section 3 and results are presented in Section 4. The conclusion and an outlook is presented in Section 5.

2. FAULT INJECTION TECHNIQUES

Fault injection is a long researched discipline which has evolved over the years. Depending on the systems under test and on the associated requirements, various categories and objectives can be distinguished. Before the typical classification and the corresponding tools are presented, the basic intention of fault injection will be described throughout the following section.

2.1. Intention of Fault Injection

All fault injection techniques aim to solve two major problems: The injection of faults as well as the observation of their effects. Ideally one would be able to inject a fault at the transistor level which models an unintended physical effect, such as a signal transition caused by a heavy ion hit,
and resulting, e.g., in a communication error at the system level. While this approach is close to reality, a practical implementation is hardly possible.

A common problem in this context is the introduction of intrusiveness which refers to the alteration of the original system due to the fault injection experiment setup (e.g., changes in the program flow, additional components, temporal variation,...).

Depending on the actual intention of fault injection, respective tools have to cope with completely different requirements. In contrast to an ideal tool which always provides low intrusiveness, high visibility and high performance, available tools are only specialized on a subset of these requirements.

2.2. Classification of Fault Injection Techniques

Traditionally fault injection is classified as hardware implemented (HWIFI), software implemented (SWIFI) or simulation-based [1] techniques. In this paper we present a different metric that is used to classify fault injection tools based on two fundamental questions: (i) Which domain is used for injecting a fault? (ii) Which domain is used for executing the experiment? The answer to these questions is visualized in Figure 1. Faults can either be injected by means of hardware or software manipulation (y-axis). Experiments can either be processed on real hardware or within a simulation environment (x-axis). The resulting quadrants characterize the features of existing fault injection tools. A tool, e.g., that manipulates software, while the execution is performed on real hardware corresponds to the classic definition of software based fault injection.

The advantage of this classification scheme is that qualitative properties of a tool can be derived from the quadrant, in which the tool resides. SWIFI tools, e.g., provide high performance but usually suffer from reduced visibility and reachability.

The following section reviews actual fault injection tools based on the introduced metric.

2.3. Fault Injection Tools

Most of the HDL fault injection tools that can be found in literature use a simulation based approach which is located in quadrant I in Figure 1. Some tools like Mephisto-L [2] apply a source modification strategy for fault injection and require an HDL simulator as execution platform. Other tools like VFIT [3] also use specific simulator commands for the actual fault injection process. Nevertheless, they all suffer from the same main drawback: high temporal costs which grow with the complexity of the model, the complexity of the workload and the number of injections.

Other approaches which speed up the execution by emulating the circuit on an FPGA are often limited in their functionality with respect to the supported fault types and observation capabilities (refer to quadrant II in Figure 1). For example, the system presented in [4, 5] only supports permanent single stuck-at faults. In [6], each fault requires reprogramming the FPGA which significantly reduces the efficiency even if partial reconfiguration is used. Sometimes the specialization of the presented approaches even goes beyond a practical usage of FPGA devices, e.g., by altering the configuration bitstream file of commercial SRAM-based FPGAs [7].

Quadrant III in Figure 1 holds the SWIFI tools. A representative candidate for this class is Xception [8] which has become a de facto standard for later SWIFI tools. Although Xception is specialized for a few target processors and uses advanced debugging and monitoring features existing in modern processors for conducting fault injection experiments, it still suffers from the typical reachability problems native to this quadrant.

Due to performance reasons, Quadrant IV does not comprise any tools.

Based on the analysis of the potential aims of fault injection experiments and the corresponding tools we designed FuSE in order to bridge the gap between simulation-based and emulation-based approaches. In contrast to the majority of available fault injection tools, which only cover a single quadrant of our classification scheme, FuSE combines the features of multiple quadrants, allowing to apply software based as well as hardware based fault injection approaches, with the help of a single tool. The corresponding concepts and first experimental results concerning FuSE will be presented in the course of this paper.

3. FuSE APPROACH

FuSE supports emulation- and simulation-based fault injection but keeps the switching between these modes transparent to the user. The speed-up provided by the emula-
tion mode allows to perform a huge number of fault injection experiments within a reasonable amount of time on the one hand and is even sufficient to support software based fault injection techniques on the other hand. The simulation as well as the co-simulation mode supports the visualization of internal sequences and is therefore mainly intended for hardware development purposes. The technology for this transparent hardware acceleration is provided by the SEmulator® Engine.

3.1. SEmulator® Engine

The speed up of FuSE is achieved by the SEmulator® engine - a hardware accelerator for HDL simulations which bridges the gap between simulation and emulation. The basic building blocks of the SEmulator® software are shown in Figure 2. One of the core components is the Hpe_desk tool which provides a user friendly interface for managing the whole SMulation setup process. Additionally to the software components the SEmulator® relies on a standard FPGA development board, equipped with a PClexpress interface. The latter is used for establishing a communication channel between the FPGA and the HDL simulator.

The SEmulator® itself is intended to accelerate HDL simulations. This is achieved by moving selected HDL components to real hardware (FPGA) instead of processing them on the host computer. This process is transparent to the simulator and can therefore be easily integrated into the development tool chain of any HDL design.

The SEmulator engine supports three different modes of operation [9]:

- **Simulation Mode**: The simulation mode corresponds to a conventional HDL simulation - both the design under test (DUT) and the testbench are executed on a host computer. The performance of the simulation obviously depends on the computational power of the host machine and the complexity of the design/testbench. HDL simulations generally achieve execution speeds in the range of a few thousand real-time clock cycles per second.

- **Co-simulation Mode**: In the Co-simulation mode the DUT is partially (or entirely) moved into the FPGA. Thus the signal changes within the test bench are sent via the PClexpress interface to the “IO Manager”, which controls and observes all attached probe pins. The IO Manager forwards all signals to the corresponding pins and reports data changes at the probes back to the HDL simulator. In this mode the clocks are generated by the controlling HDL simulator. The maximum speed can be as high as 200,000 clock cycles per second, while the visibility of internal signals is still preserved.

- **Clock Acceleration Mode**: In clock acceleration mode, the design is executed for a specified number of clock cycles on the prototyping platform with the maximum speed possible. No communication between the HDL testbench and the DUT takes place – thus within the acceleration mode the internal signals are not visible. This mode allows to increase the simulation frequency to up to 100MHz for selected time slots of the simulation.

Furthermore, it is possible to switch between those modes dynamically. This allows to skip less interesting time intervals, e.g. a boot-loader sequence, in “real time” by using the accelerated mode and start the co-simulation afterwards, allowing to analyze the behavior of the circuit in greater detail again.

3.2. FuSE Concept

Although FuSE can be used as stand alone VHDL-based fault injection tool, all concepts and mechanisms have been tailored to the usage of the SEmulator® in order to achieve its full effectiveness. In this way the advantages of simulation based fault injection, namely flexibility and visibility, are combined with the high performance of execution based fault injection. While designing FuSE a strong focus was placed on flexibility. FuSE should not be optimized for a golden node/run strategy, but rather allow users to easily adapt the FuSE framework to their demands instead.

Figure 3 depicts a structural view of the presented concept which consists of three main parts: a control unit that is attached to the DUT, so-called saboteur devices, which apply faults inside the DUT, and small pieces of code that are added to the original testbench. The latter configure the control unit and record selected data during the fault injection experiments. Additionally the Hpe_desk environment has
been adapted to fully support the fault injection approach. For the actual fault injection process the DUT has to be enhanced by saboteur devices and the corresponding fault activation ports as well as by data recording ports. Each saboteur is equipped with a control port that is directly connected to FuSE via the corresponding fault activation lines. These allow to configure the saboteur’s behavior individually and even turn the devices off completely.

FuSE’s output is redirected to a text file, which holds all signals of the DUT that were selected during the configuration process. In order to reduce the communication overhead, and as a consequence to reduce the amount of data which has to be stored, FuSE supports so-called trace conditions. A typical example for a trace condition is the read enable signal of a memory block. While “read enable” is deactivated, data carried at the memory output will not be recorded.

Note that the FuSE control unit is situated at the same hierarchical level as the DUT). From the SEmulator’s point of view it is therefore interpreted as an arbitrary VHDL design. Thus, FuSE can take over all advantages introduced by the SEmulator.

### 3.3. Usage of FuSE

The setup of a fault injection experiment requires a sequence of configuration steps. First, location and type of faults to be injected must be defined. This can either be done by adding a stylized comment to the HDL code next to the signal/variable that should be manipulated, or by using Hpe_desk’s scripting interface. Once all faults are defined, the HPE_desk software automatically modifies the original HDL code according to the defined specification and adds the FuSE specific components to the testbench.

Subsequently the activation of each fault needs to be determined. This can be specified in terms of DUT clock cycles or by so-called injection conditions. Injection conditions allow to enable/disable faults according to internal events, e.g. “activate Saboteur2 when jmp.exe was active four times”.

For fault forecasting purposes, e.g., our framework also supports the injection of random faults. In this case a random generator selects the location, the type and the activation of each fault automatically. Note, although the experiment setup is fortuitous the execution will still be deterministic and therefore be repeatable.

Based on the setup information the HPE_desk generates a command file, which is downloaded to the FuSE control unit. The command file consists of a configuration part and a fault definition part. Besides the start-up configuration, FuSE also offers the possibility to perform a reconfiguration at run-time. This way the number of faults that can be injected in the course of a single experiment is not restricted by the allocated memory inside the FuSE control unit. During the fault injection experiment FuSE saves the state of the observation signals to a text file. Similarly to the saboteur control ports, these signals were automatically routed to the output of the DUT by the HPE_desk software.

In order to further improve the efficiency, an individual hardware configuration for each experiment should be avoided. The superset of fault locations and observation signals of all experiments of a fault injection campaign should be used instead. The SEmulator currently supports up to 12k probe pins and FuSE up to 4k saboteur devices. This way the design has to be synthesized only once in order to perform multiple experiments. The various fault injection scenarios can be selected dynamically by simply changing the configuration file. Note that a fault free reference can be generated by downloading a configuration file without fault definitions (= golden run approach).

### 3.4. Supported Fault Types

Although the emulation takes place in an FPGA, which probably will differ from the final target platform, it was an aim to keep the intrusiveness as low as possible. Therefore, the saboteur devices are kept extremely simple so that they could be optimized for the underlying hardware. Besides plain stuck-at faults our design decisions also had to consider fault types that depend on the state of a single or multiple signals as, e.g., bitflip, bridging or delay faults. We found an elegant solution that comprises three generic saboteur devices: a logical saboteur for stuck-at 1/0 and bitflip, a bridging saboteur for AND, OR, and dominant bridging faults of two signals and a delay saboteur which emulates delay faults. The first two devices yield an extremely efficient
4. EXPERIMENT SETUP AND RESULTS

The current section presents the results which were obtained for a simple fault injection experiment based on our first prototype implementation of FuSE.

The tool chain for conducting the experiments consisted of Altera QuartusII 8.0 [10], Mentor ModelSim 6.2g [11] and the latest version of Hpe_desk[9] from Gleichmann Electronics Research. The host computer running all the experiments comprised an Intel® Core(TM)2 Duo CPU running at 3 GHz and 3 GB RAM. Furthermore, the HMX2-AS2 FPGA development system which is equipped with two Altera Stratix II FPGAs (EP2S180F1508C3N) was used as co-simulation platform. The hardware overhead introduced by the FuSE prototype is negligible – in fact, requiring only 126 ALUTs\(^1\), 171 registers and 1024 memory bits, the target’s resource utilization is far below 1%. Additionally, with a maximum speed that is more than twice as high as the maximum clock frequency supported by the SEmulator Engine \(f_{\text{max}} = 100 \text{MHz}\), FuSE will not become a bottleneck on the current version of this platform.

As DUT a compact soft core processor called SPEAR2 [12] was used. While SPEAR2 basically is a 16-bit architecture the data path can be extended to 32 bit via the provided configuration framework. Due to higher performance and a larger address space the 32 bit version was chosen for our experiments. For the available hardware platform this results in an overall resource utilization of 2157 ALUTs, 694 registers and about 656kB of block memory and a maximum clock frequency of 75.17MHz.

During the experiment setup the original processor design was enhanced with ten logical saboteur devices which can be configured at runtime as stuck-at 0/1 or as bitflip. For the offline analysis, a 55 bit wide processor bus was attached to FuSE’s observation interface. FuSE itself was configured to hold a maximum of 21 fault injection commands that can be downloaded within a single configuration phase. These values can easily be changed through a configuration option in order to fit the user’s needs.

For the first fault injection experiment a period of 1 second of real time was selected. During the execution 10 stuck-at 0 faults were injected. Note that for the execution time of the experiment it makes no difference whether stuck-at 0/1, bitflip or an arbitrary combination of these faults is used. Each fault was deactivated again after a single DUT clock cycle. In order to derive performance measurements the experiment was executed as behavioral simulation only and as co-simulation using the SEmulator engine. The corresponding results are presented in Table 1.

While the simulation of 1 second takes 1.7 hours, in the co-simulation the same experiment can be performed within about 12 minutes. In contrast to the accelerated mode which allows to perform the experiments close to real time but only yields a “go/non-go” information, the co-simulation mode allows to monitor all internal signals.

Due to the fact that FuSE uses extremely simple saboteur devices which emulate complex fault types through additional fault injection commands, the overhead introduced by a repeated download of these commands should be investigated. By using the previous experiment setup this reconfiguration has to be performed multiple times during the experiment execution. For analysis purposes we decided to inject 10 transient faults (= 20 commands) per reconfiguration cycle. In order to put additional stress on the PCI-express interface we reduced the simulation time to 10 ms.

Table 2 presents the corresponding results which were derived for the co-simulation. The measurements show that additional reconfiguration cycles are negligible compared to the overall simulation time. Note that the determined temporal overhead is a static value for a given number of faults. Therefore, the injection of 10000 faults will always prolong the co-simulation of the same circuit by 0.5 seconds.

5. CONCLUSION AND OUTLOOK

Although the FuSE fault injection tool is still under development, the first prototype implementation already shows the benefits of this approach. As starting point for the fault injection process only an HDL description of the circuit and a testbench is required.

The subsequent fault injection setup process has been

\(^1\)Adaptive LUTs (4 Inputs, 1 Output) are the basic building blocks of Stratix II FPGAs.
fully automated so that the user only has to provide a minimum amount of information. This mainly consists of determining the fault locations, types and activation times. In the current implementation this information can either be provided by using stylized comments within the HDL description or the scripting interface provided by the Hpe desk tool suite.

The FuSE approach tries to find a tradeoff between resource usage and flexibility. The saboteur devices were developed to fit within a single logic element but still be configurable at runtime in order to conduct various fault injection experiments with the same set of saboteurs. Therefore, a multitude of fault injection experiments can be conducted based on a single synthesized circuit - only the saboteurs required for a specific experiment are enabled while all others are deactivated and do not disturb the execution. The overhead for even multiple reconfiguration cycles is negligible compared with the synthesis time of a complex circuit.

All these features were tailored to and integrated within the SEmulator® environment which allowed to speed up fault injection experiments at least by a factor of 8 in the co-simulation mode (4000 in the clock acceleration mode) compared to typical simulation based approaches. In contrast to conventional hardware acceleration systems the SEmulator® is based on standard tools (Modelsim and Quartus) and is set up on a low cost environment consisting of a prototyping FPGA board, a PCI-express Interface and an off-the-shelf PC.

Although our prototype allows to perform first fault injection experiments, some of the features and ideas, e.g., the insertion of delay faults, still need to be implemented.

A drawback we found during our experimental runs is that the observation files tend to become very large (1GB for the experiment presented in Table1). Nevertheless, compressing these text files with typical tools shows that the size can easily be reduced by a factor of 33. We are therefore confident to find a good solution for this problem.

Due to the increasing interest in multiple event upsets (MEU) our current research also addresses the injection of multiple concurrent faults. While this feature can be made available in co-simulation mode it still poses a problem for the clock acceleration.

### Table 2. Reconfiguration Performance

<table>
<thead>
<tr>
<th>Number of Faults</th>
<th>Co-Simulation Performance</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7,031 s</td>
<td>0 s</td>
</tr>
<tr>
<td>10</td>
<td>7,056 s</td>
<td>0,025 s</td>
</tr>
<tr>
<td>100</td>
<td>7,313 s</td>
<td>0,282 s</td>
</tr>
<tr>
<td>1000</td>
<td>7,325 s</td>
<td>0,294 s</td>
</tr>
<tr>
<td>10000</td>
<td>7,495 s</td>
<td>0,464 s</td>
</tr>
</tbody>
</table>

### 6. REFERENCES


