Fault-Effect Analysis on Multiple Abstraction Levels in Hardware Modeling

Bogdan-Andrei Tabacaru*†, Moomen Chaari*†, Wolfgang Ecker*†, Thomas Kruse*, and Cristiano Novello*
*Infineon Technologies AG - 85579 Neubiberg, Germany, †Technische Universität München
Am Campeon 1-12, 85579 Neubiberg
Firstname.Lastname@infineon.com

***Abstract*—The introduction of the ISO-26262 safety standard for automotive applications has increased the verification complexity of safety-critical systems-on-chip dramatically since failure-free behavior in case of faults must be verified in addition to correct functionality. Fault injection on gate level and RTL models has become the de-facto standard in simulation-based safety verification of SoCs. However, safety verification on such low-level models presents a series of challenges that are difficult to solve by classic verification methods. Therefore, we propose a safety-verification methodology using transaction-level models and show that single-bit fault injection in these models is a sufficient measure for early fault-effect analysis.**

Keywords*—*Fault injection; safety verification; SystemC; TLM; virtual prototypes

I. Introduction

Safety verification of large scale systems on chip (SoCs) is an ever increasing challenge for the automotive industry. Since its introduction, the safety standard for automotive applications, ISO-26262 [1], has driven the industry to define more stringent safety requirements for their products. As a consequence, the SoC developers are constantly implementing better processes to successfully verify the safety goals of their systems. In turn, each new process adds an extra element of complexity to the already existing safety-verification workload and, therefore, effectively increases the SoCs' development time and effort.

Nowadays, safety verification is mainly performed on gate level (GL) net-lists (i.e., permanent faults) and register-transfer level (RTL) models (i.e., transient faults). These abstraction levels are practical for safety verification because they contain a sufficient amount of detail with respect to the verified system. Therefore, fault-injection techniques can be successfully applied on such models. However, injecting faults into GL and RTL models of complex systems is very challenging due to these models' very slow simulation performance.

As a result, virtual prototyping (VPs) [2] using SystemC [3] and transaction-level modeling (TLM) [3] is evaluated to address the performance issue of GL and RTL models for safety verification. Therefore, the TLM abstraction level reduces implementation details of a system for much higher simulation speed. This trade-off introduces an important challenge for safety verification because, after abstracting implementation details, information about fault-injection locations within the model is also lost. Simply put, TLM does not model gate-level signals. As a consequence, fault injection into transaction-level (TL) models is limited to the model's boundaries (i.e., interface) since more implementation details are either not available or could dramatically slow down the simulation performance. In turn, the missing fault-injection locations must be manually traced down the abstraction path to the RTL and GL models (Fig. 1).

An advanced approach for safety verification is to go bottom-up and manually abstract parts of GL models to the RTL and, finally, to the TLM abstraction levels [4]. Afterwards, faults are injected into specific parts of TL models to test the corresponding software and hardware-based safety-mechanisms. The problem with this approach is that safety-concept errors (e.g., redundant/missing safety mechanisms) are only noticed late in the development stage, since it requires an existing, almost

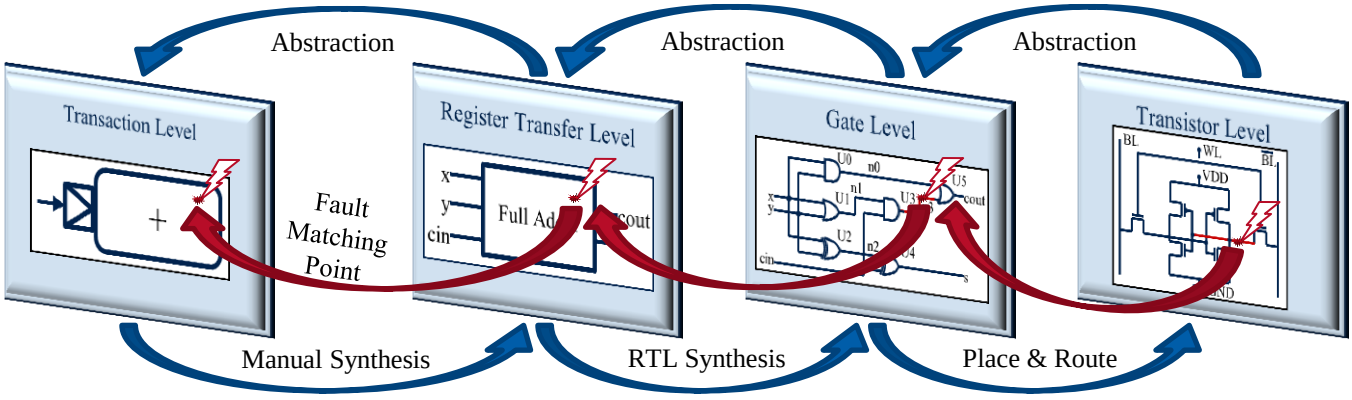

Figure 1. Identifying fault-injection matching-points on multiple abstraction levels

final, net-list. A further problem of this approach is that late-stage changes in the design are costly (i.e., RTL and GL models must be updated and the safety-verification flow must be repeated). For this reason, a methodology to inject a sufficient amount of *correct faults* at the TLM abstraction level without detailed knowledge of the GL net-list is highly required, but missing.

A *correct fault* is any single bit or multi-bit fault which, given a particular input pattern, leads to the same failure in the TL model and corresponding GL net-list. This differentiation is necessary when considering top-down development flows (i.e., the TL models are developed before the RTL or GL models). Here, the injection of single bit and multi-bit faults into TL models can lead to the observation of an impossible or statistically improbable combination of output bits [5]. Thus, the injection of these *incorrect* (i.e., *wrong*) faults can, ultimately, lead to the implementation of unnecessary safety mechanisms in the SoC.

Top-down SoC-development flows address the performance issues of bottom-up safety-verification methods. By abstracting complex SoCs as TL models, the simulation performance increases. Additionally, TL models can be more easily updated based on SoC-specification changes than RTL or GL models. As a result, concept errors can be discovered and handled more rapidly on TL models. However, the challenge of fault injection into these new models still remains. Without prior knowledge of a model's implementation, it is difficult to determine hardware-accurate fault-injection locations and the effects of faults injected therein.

In this paper, we aim to provide a pragmatic solution to the aforementioned problems. First, we present a generic and non-intrusive approach to inject faults into GL and TL models (i.e., faults are injected without changing the original source code of the model). Furthermore, our fault models can be interchanged during a simulation to suit any fault-injection scenario. Additionally, our approach supports single bit as well as multi-bit fault injection into the TL models. Second, we analyze several GL net-lists with respect to fault propagation, compare the results with TL models and code snippets, and derive a set of fault-injection rules (i.e., guidelines), which apply to the safety verification of TL models. This way, concept engineers can create TL models and use them to implement and verify safety mechanisms early in the development stages and with sufficient accuracy.

The remainder of this paper is structured as follows. Section II presents related work in the field of safety verification and fault injection. Sections III and IV are dedicated to the fault-injection frameworks used for GL net-lists and TL models, respectively. Next, the verification framework used to simulate the GL and TL models is described in section V. In section VI, the analysis of several combinational models is summarized and the results are presented. Finally, section VII contains the paper's conclusions.

II. Related Work

The first simulation-based fault-injection techniques have been developed for GL net-lists [6], [7]. These approaches vary from explicitly changing (i.e., mutating) parts of the original net-list to emulate the behavior of one or more faults to adding new components (i.e., saboteurs) to the net-list that inject faults during GL simulations. However, simulating GL net-lists for large systems (i.e., automotive ECUs) suffers from slow simulation performances because of the large gate-counts. This amount of gates increases the computational cost of a GL simulations.

To improve the verification speed of GL models, emulation-based techniques have been created which upload synthesized net-lists either to graphics processing units (GPUs) [8], [9] or to field-programmable gate arrays (FPGAs) [10]–[13]. Therefore, fault-injection campaigns are carried out by fast-running dedicated hardware components. However, these methods are very expensive and still rely on the presence of the SoC's synthesized net-list, which drives the safety verification into the late stages of the SoC-development process.

For this reason, fault-injection methods have been developed for RTL models [6], [7], [14], [15]. These techniques are focused around the following principles: 1) saboteurs, 2) mutants, and 3) simulator commands.

Saboteurs are generic RTL modules integrated in key parts of a system. Each saboteur can be activated during a simulation to imitate the behavior of a hardware fault. *Mutants* are software which explicitly changes the original RTL model. The changes emulate the effect of different hardware faults. *Simulator-command-based* fault injection is a technique that uses the RTL simulator to manipulate values of registers or signals during a simulation.

However, these fault-injection methods also contain a series of drawbacks. First, RTL fault-injection techniques add a considerable amount of simulation overhead to the already low-speed RTL systems. Second, the fault-effect analysis mainly ends at the RTL abstraction level without comparing the results to the GL net-list [16], [17]. Last, since RTL models take a long time to develop and verify, the safety-verification process of large systems is delayed until the RTL system is stable (i.e., when the cost of fixing errors in the safety concept are high). Finally, these methods, especially mutants, produce a set of wrong failures (i.e., effects of injecting wrong faults).

Moreover, mixed abstraction level fault-injection methodologies have shown that it is difficult to find fault-injection locations (i.e., matching points) which are common to both the RTL and GL abstractions [18]–[21].

Nowadays, fault-injection techniques for VPs are becoming more prominent. SystemC-based methods have been developed for RTL-like models [22]–[24]. Additionally, approaches have been created for the TLM library [25]–[29]. However, even though VPs offer faster simulation speeds to RTL and GL models, the number of matching points is significantly reduced. Therefore, safety verification performed on the TLM abstraction level is incomplete (i.e., cannot be used for SoC sign-off) and is regarded

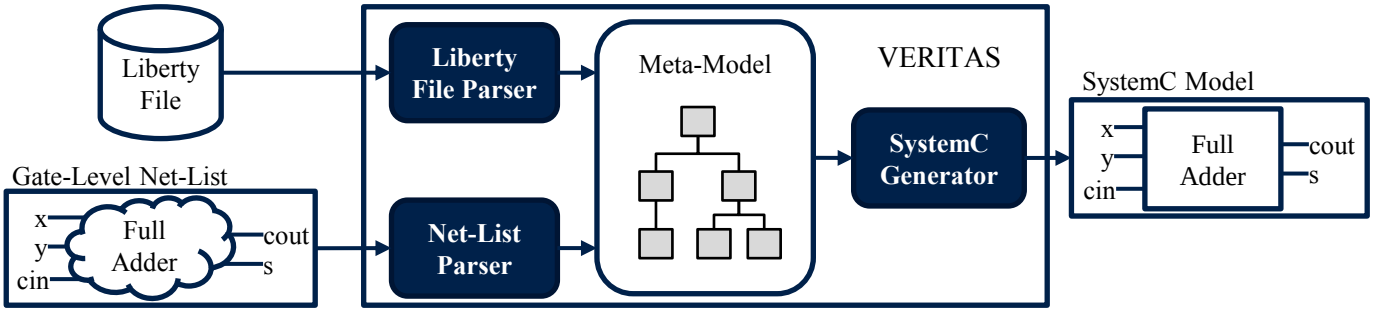

Figure 2. VERITAS framework

by experts as less reliable than on other abstraction levels [5]. For this reason, guidelines are required to improve the safety-verification reliability of TL models.

In this paper, we focus on improving the reliability of TLM-based safety verification by analyzing TL-GL correspondence and deriving a generic fault-injection plan for TL models. Thus, the final structure of the model under verification is not needed in early safety-verification runs and the TL model can be verified using a black-box approach. Moreover, the models can be used to develop architecture-level safety mechanisms for large scale SoCs more effectively before the GL net-lists are available.

III. Gate-Level Fault Injection

Gate-level (GL) models contain all fault-injection locations of an SoC but require long simulation runs. To speed-up the simulation time of GL models, we have developed and applied a compiled-code simulation approach called VERITAS (Verilog net-list to SystemC transformer) to the synthesized net-lists (Fig. 2). Thus, Verilog net-lists and the logic-gate descriptions present in Liberty files are transformed into SystemC code which has the same structure as the GL net-list, including the same fault-injection locations (i.e., matching points). These models are then integrated in existing SystemC/TLM-based virtual prototypes.

Faults are injected into the GL models using SCFIT (SystemC Fault-Injection Tool) [22], [24]. SCFIT uses the GNU Debugger (GDB) to set breakpoints and watchpoints to input ports and internal signals of SystemC models during execution (Fig. 3). Thus, the port's and signal's contents can be accessed and altered to emulate the effect of a hardware fault. This is achieved automatically with the help of Python scripts, which actively configure and control GDB with every execution step. This way, not only permanent (e.g., stuck-ats), but also transient faults (e.g., single-event upsets) can be injected into the generated GL models.

SCFIT follows a particular fault-injection execution flow due the use of GDB (Fig. 4). First, the SystemC models are compiled on a target machine (e.g., Intel x86_64 CPU). Next, SCFIT is configured by setting up its fault-injection parameters:

- **fault-injection location**: the name of the GL model (e.g., `full_adder`) and the internal signal (e.g., `n0`) in which to inject the desired fault.
- **fault type**: permanent (i.e., stuck-at-0, stuck-at-1) or transient (i.e., bit-flip).
- **fault (de-)activation**: simulation time or event (e.g., variable assertion) when the fault effect is triggered or stopped.

Afterwards, the compiled SystemC models are loaded into GDB, which is initialized using SCFIT's configuration scripts. Next, GDB automatically sets up the corresponding breakpoints and watchpoints to the configured fault-injection parameters. Finally, the simulation is started and results are collected.

The execution of the compiled-code approach has higher performance (i.e., faster simulation speed) compared to the RTL simulation [30] and easier fault-injection management when utilizing object-oriented programming features from C++ (e.g., polymorphism). However, the standard Intel x86_64 CPU architectures only provide four hardware breakpoints and watchpoints [31], [32]. If this number is exceeded, ample software breakpoints are still available. However, the simulation performance is

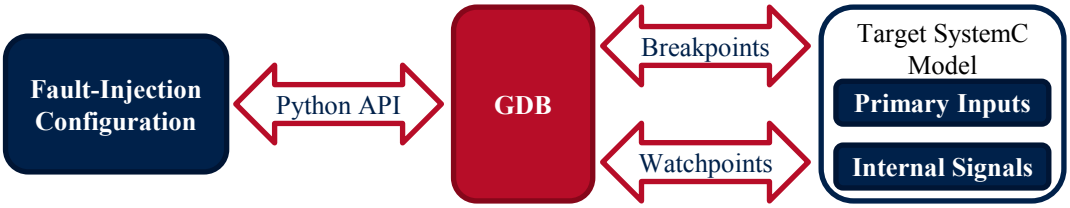

Figure 3. SCFIT's fault-injection mechanism

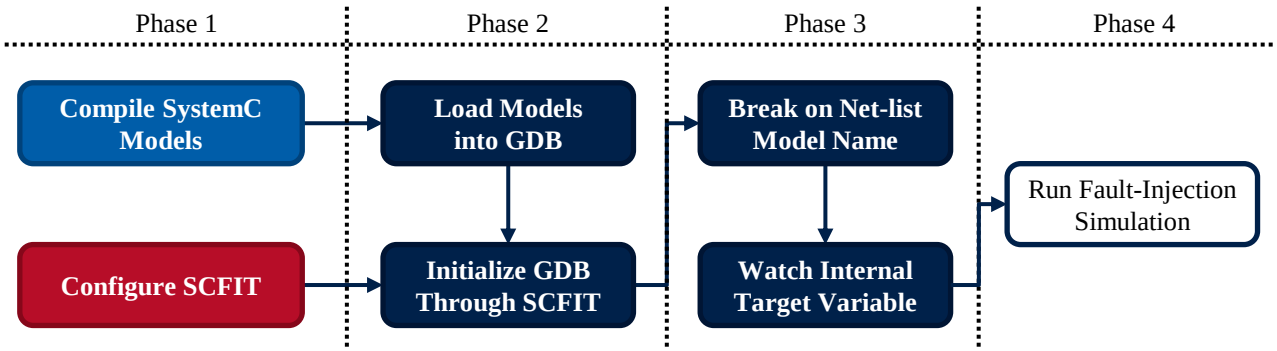

Figure 4. SCFIT's execution flow

greatly affected. Fortunately, this limitation does not affect the fault-injection approach's performance when injecting single faults per simulation.

IV. Transaction-Level Fault Injection

Transaction-level (TL) models are highly abstracted behavioral models with a fast simulation speed but limited structural information (e.g., reduced number of fault-injection locations compared to functionally equivalent GL models). For this reason, TL models are highly efficient prototyping tools (e.g., virtual prototypes) because they allow much faster changes than RTL or fate-level (GL) models. As a result, TL models are often created before RTL and GL models and serve as functional proofs-of-concept for SoCs under development.

While the verification environment of a TL model is conceptually identical to that of a GL model, the structural composition is different (e.g., TL models use socket-based communication instead of ports and signals which are used by GL models). Moreover, virtual prototypes (VPs) based on the TLM-library [3] do not have built-in fault-injection features, whereas faults can be injected into GL net-lists using simulator-commands which dynamically change a gate's value during simulation. For this reason, we have provided new fault-injection mechanisms for the TL modeling patterns.

To add fault-injection features to TL models, we have extended the TLM library with additional fault-injection initiator and target sockets. These sockets contain configurable call-backs (Fig. 5), with which faults can be injected during a TL simulation [26], [29].

Faults are injected by a fault injector (i.e., a C++ function) connected to one of the specific call-backs either on the initiator or target socket (Fig. 6). The call-backs represent extensions of TLM transport mechanism:

- **pre/post-transport**: a fault-injector connected to this call-back injects faults before/after the execution of the transport method implemented in the *target* block. Therefore, faults injected with the *pre*-transport call-back are useful for fault-effect analysis on the target block's forward path, whereas faults injected in the *post*-transport call-back are used to emulate error-propagation from the *target* block on the TLM response path.
- **transport override**: a fault-injector connected to this call-back effectively replaces the transport method implemented in the *target* block. Thus, faults injected here are particularly useful when the block's whole correct functionality must be temporarily exchanged with a faulty one, without replacing the original *target* block's code.

Fault injectors gain access to the transport method's payload, phase, and delay arguments. Thus, the payload's attributes and the transport method's arguments can be modified by a fault model to emulate the effect of any injected fault (i.e., single or multi bit).

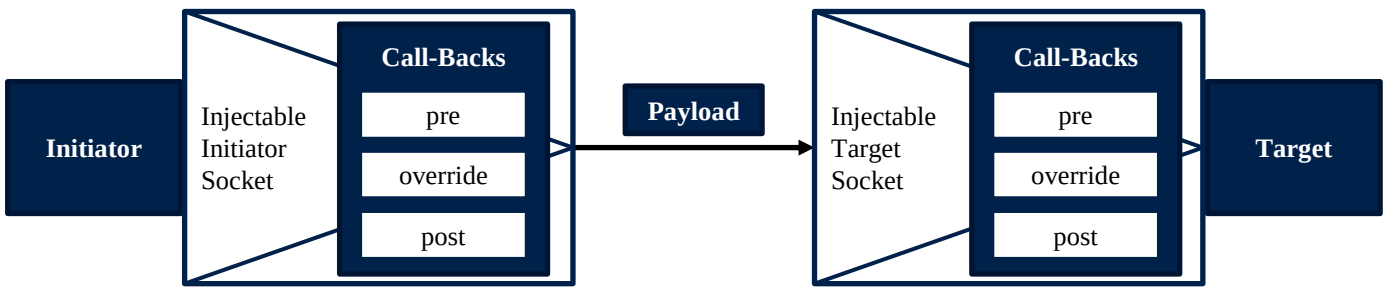

Figure 5. TLM injectable initiator and target sockets

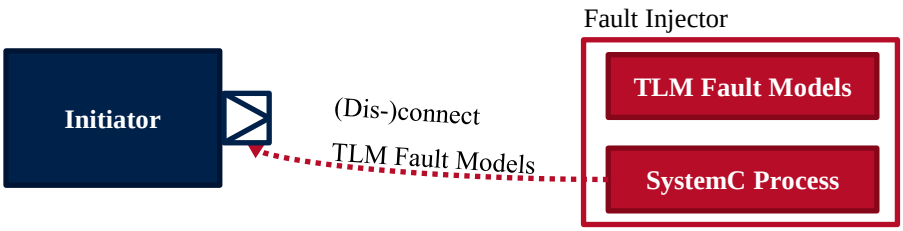

Figure 6. TLM fault-model decoupling

Fault models are a part of the fault injector and can be (dis-)connected during simulation to emulate the behavior of permanent or transient faults; a dedicated SystemC process is used to trigger the activation and/or deactivation of a TLM fault model during a simulation. Thus, non-intrusive fault injection and fault modeling become possible and provide higher fault-model re-usability (e.g., register address fault, bus access fault). Furthermore, the injected faults as well as their effects can be mapped to the faults injected into GL models. Thus, improved matching points can be established between different TL, RTL, and GL models and fault-effect analysis on these abstraction levels becomes feasible.

The TLM injectable initiator and target sockets only allow fault injection into the interface of a TL model. To also inject faults inside TL models, this approach can be combined with the one presented in section III. Thus, the gate-level-accurate SystemC models can be easily connected to a TLM test-bench by means of a TLM transactor (Fig. 7). In this case, during a (non-)blocking transport-method call, the SystemC models are (de-)activated from the target block. Furthermore, faults can be injected inside the target block and the SystemC models using GDB. Thus, the TL models gain control of the SystemC models and become enriched with the complete structure of the GL models (i.e., all GL matching points). Moreover, because of the compiled-code simulation approach [30], the execution speed of the SystemC models is still much higher than that of a pure GL model.

V. SaVer Framework

The generated GL models as well as the TL models are simulated using the SaVer (Safety-Verification) platform. This simulation environment follows the classic pattern of a hardware-verification framework (Fig. 8).

The *stimulus generator* is a block which drives the input values of the model under verification. The stimuli are organized into a *stimulus library*.

The *monitor* is a block that reads in the changes at the outputs of a model under verification with each simulation step.

The *fault injector* represents an additional stimulus generator with access to the internal wires of a model. It is used to emulate the effects of a physical fault (i.e., short circuits are modeled as stuck-at-1, open circuits are modeled as stuck-at-0). Similar to the stimulus generator, the faulty stimuli are grouped into a *fault library*.

The *data analyzer* uses the data from the monitor to determine whether a failure occurred at the output of a model or not. Furthermore, the type of fault (i.e., single or multi bit) is also determined.

The *controller* is a block which manages the stimuli and injected faults. Additionally, it stops the simulation upon the detection of a failure.

As shown in Fig. 9, the fault-injection simulation flow is divided into four main phases. First, input patterns (i.e., input stimuli) are generated and driven to two instances of the same GL model. Additionally, faults are injected into one of the models, while the second model is executed fault-free and serves as a reference to the first one. The SaVer framework supports the injection of any number of faults; for this analysis, we have injected single-bit faults for each input pattern. Next, the two models are executed sequentially. The outputs of the faulty and reference models are compared after each simulation step. Finally, any differences detected in outputs are failures and are sent to the failure-coverage processing phase. In this final phase, the number of observed failures are counted and compared to the number of injected faults which did not lead to a failure (i.e., faults which are masked by the circuit topology during simulation). The number of failed output bits are also counted and used to calculate the damage of the injected fault (i.e., how many output bits failed after injecting a single-bit fault for a given input pattern).

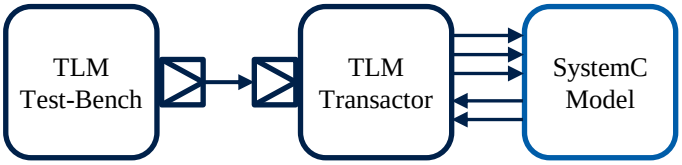

Figure 7. TLM transactor which controls the SystemC representation of a gate-level model

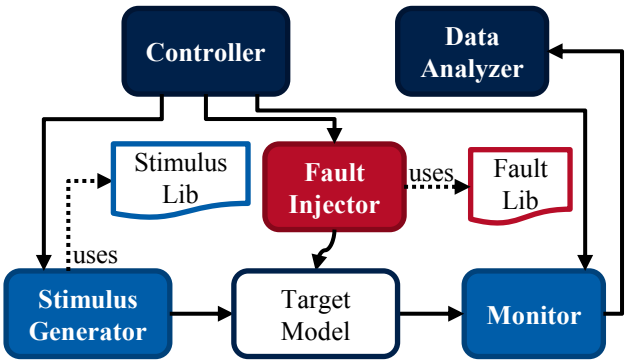

Figure 8. SaVer's verification environment with fault-injection features

VI. Case Studies and Results

We have applied the aforementioned fault-injection approaches on GL and TL models of several adder architectures and sub-blocks of a MIPS CPU. Our focus has been to quantify the differences of fault effects and observed failures between the two abstraction levels. For this reason, we have analyzed whether the effects of single-bit faults injected into GL models can be reproduced using single-bit fault injection into the TL models which have less fault-injection locations than the GL models. Moreover, we have studied which faults injected into the TL models are realistic (i.e., lead to an observable failure after injecting a single-bit fault in the GL model). The results of our simulations as well as the performance of our methods are discussed below.

The stimuli driven onto the inputs of GL and TL models have been generated randomly using the Monte Carlo method [33], [34]. The same approach has also been applied to the generation of the injected faults. Additionally to the generated stimuli, faults have been injected into the internal signals of a GL model as well. Thus, we have run and documented the results of fault-injection campaigns with permanent faults (i.e., stuck-at-1, stuck-at-0) using multiple sample ranges (e.g., 10 000, 100 000, 1 000 000 samples).

All simulations in our case studies have been conducted on a 64-bit PC with an Intel® Xeon® E5 CPU @3.00 GHz, L3 cache of 25600 kB, and 264 GB of RAM.

A. Adder Architectures

An adder has the same intrinsic functionality regardless of its implementation (e.g., carry adder, carry-lookahead adder). In other words, an adder calculates the sum and overflow of two or more inputs. For this reason, the behavioral (i.e., TL) model of an adder is always the same; the only real difference is in the adder's size (e.g., full-adder, nibble-adder, 8-bit adder). Conversely, depending on its implementation, GL models vary in complexity and area; for example, carry-save adders faster execution speeds compared to carry-adders and also have considerably higher gate-counts. As a consequence, carry-save adders also have substantially more fault-injection locations than carry-adders.

The GL model of an n-bit adder is synthesized from an RTL representation of a specific implementation (e.g., carry-adder, carry-lookahead adder). The input interface contains two n-bit inputs (i.e., x, y) and a carry-in bit. The output interface contains one n-bit output (i.e., sum) and an overflow bit. The TL model uses the same input-output interface. In comparison to the GL model, the TL model only implements the "+" (i.e., add) operation and, therefore, has no internal variables.

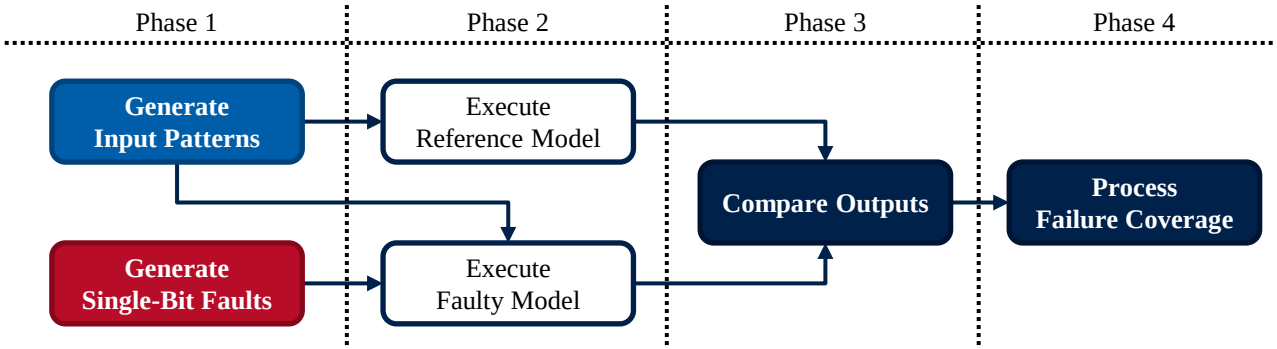

Figure 9. SaVer's fault-injection simulation flow

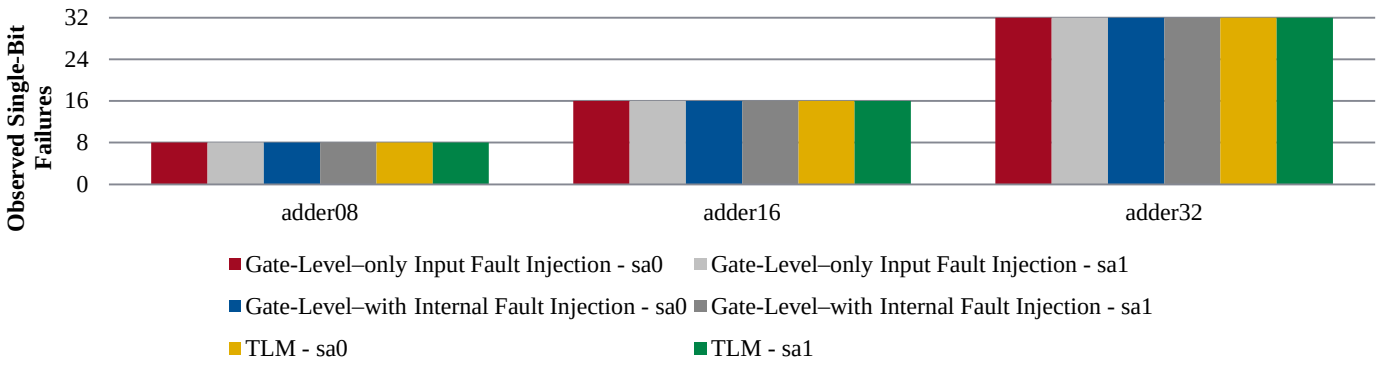

Figure 10. Observed number of faulted bit positions in the sum output of an 8, 16, and 32-bit adder after 10 000 samples

Using the Monte Carlo method, we have randomly generated samples to cover a large spectrum of input-pattern combinations into which faults are injected. As a result, 10 000 samples have been sufficient to observe at least one failure on each bit of an adder's sum output regardless of its size (i.e., 8, 16, 32 bits). As shown in Fig. 10, both the stuck-at-0 and stuck-at-1 faults cumulatively lead to the maximum number of observable single-bit failures (SBFs) in the sum output for each adder size. Moreover, these results are achieved both for TL and GL models and are independent of fault-injection locations (i.e., at the input-interface level or internal).

Conversely, random generation of stimuli and faults are not as efficient when considering corner cases (i.e., less probable outcomes). As a consequence, Fig. 11 shows that the Monte Carlo approach is not always a reliable sampling method. Here, random stuck-at-1 injection into the adder models successfully leads to failures in the carry-out output (i.e., the bars below the 50% margin). However, stuck-at-0 faults are not as successful when randomly injected into 32-bit adder models (i.e. missing bars above the 50% margin for the 32-bit adder). In this case, not even one carry-out failure is observed even after 10 000 000 simulations. Therefore, a greater sample size is required to cause a carry-out failure. However, more samples also require a longer verification processes. As a result, directed tests are necessary to reduce the total verification time of the models.

Furthermore, we have quantified the differences of injecting faults into GL and TL models by comparing the size of the observed multi-bit failures (MBFs) after injecting stuck-at-1 (Fig. 12) and stuck-at-0 (Fig. 13) faults. Clearly, fault injection into the inputs of a GL model offers the same results as when injecting faults into the inputs of the corresponding TL model (i.e., the same input patterns have been used for both abstraction levels). Additionally, faults injected into inputs and internal signals of a GL model provide similar results to those injected just into the inputs of a TL model.

The aforementioned results are mirrored when considering the size of the observed MBFs. Therefore, the average number of random samples required to observe MBFs is mainly dependent on the adder's size and becomes unmanageable as the size increases (e.g., 10 000 000 samples caused only a 21-bit failure in a 32-bit adder).

To understand how many random samples are averagely required by an n-bit adder to cause a maximum MBF (i.e., n+1 failure) in an n-bit adder, we have calculated the the failure's probability as a function of the adder's size. For this calculation, we have considered fault injection only into the model's inputs. Thus, these samples can be applied to GL as well as TL models.

The total number of all outcomes (AO) is given by the total number of input stimuli into which either stuck-at-1 or stuck-at-0 faults can be injected. This number of stimuli is required to exhaustively cover all input bit configurations of an adder. Given

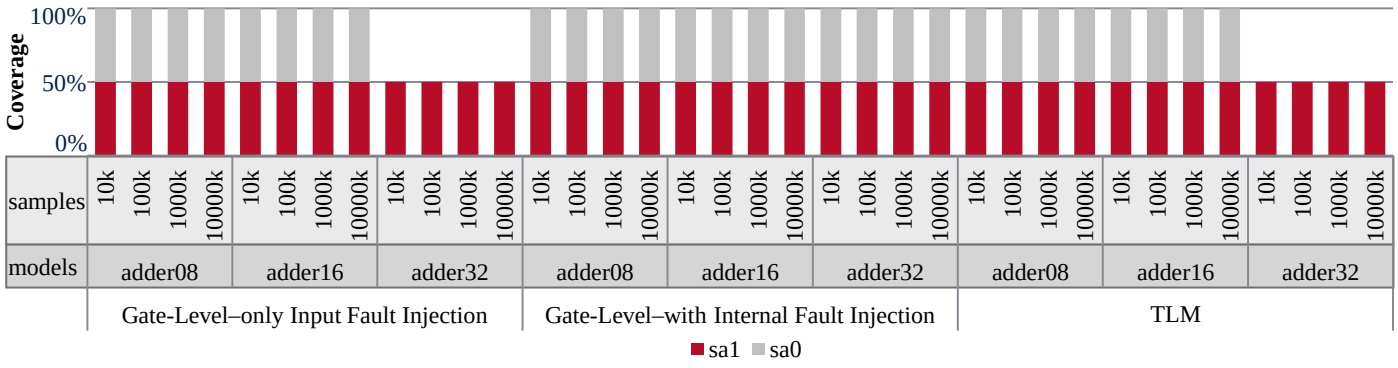

Figure 11. Coverage of carry-out failures per fault type observed for 8, 16, and 32-bit adder

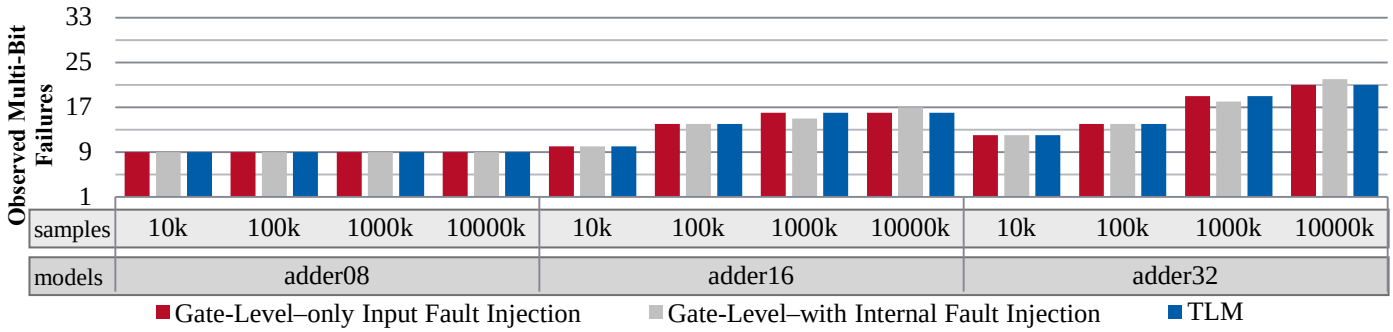

Figure 12. Maximum multi-bit failure observed after injecting stuck-at-1 faults in adders of variable sizes

the adder's size (i.e., n-bits) and its inputs (i.e., two n-bit inputs and one carry-in bit), the number of input stimuli is:

$$N_{stimuli} = 2^{2n+1} \tag{1}$$

where

$n =$ adder size

Additionally, stuck-at faults can be injected into any of the adder's 2n+1 input bits, regardless of the given input pattern. Thus, AO becomes:

$$AO = (2n+1) \cdot 2^{2n+1} \tag{2}$$

Next, we have derived the number of samples necessary to cause an n+1-bit failure in an n-bit adder. Maximal MBFs are observed in one of two ways:

- A stuck-at-1 is injected in the least significant bit (LSB) of an input with initial value 0, when the adder's sum result is maximal and no overflow is registered. For a full adder, the input pattern a=0, b=0, cin=1 leads to outputs cout=0 and s=1. A stuck-at-1 fault injected into either a or b causes a 2-bit failure by flipping the output bits (i.e., cout=1 and s=0).
- Inversely, a stuck-at-0 is injected in the LSB of an input with initial value 1, when the adder's sum result is minimal and an overflow is registered. In this case, the input pattern a=0, b=1, cin=1 leads to outputs cout=1 and s=0. A stuck-at-0 fault injected into either b or cin causes a 2-bit failure by flipping the output bits (i.e., cout=0 and s=1).

The number of desired outcomes (DO) for stuck-at-0 faults has been determined experimentally and generalized as:

$$DO = 2(2^n + 2^{ceil(\frac{n-1}{n})}) \tag{3}$$

Thus, the probability of causing an n+1-bit fault by randomly generating input patterns and injecting a particular stuck-at fault is calculated as:

$$P = \frac{DO}{AO} \tag{4}$$

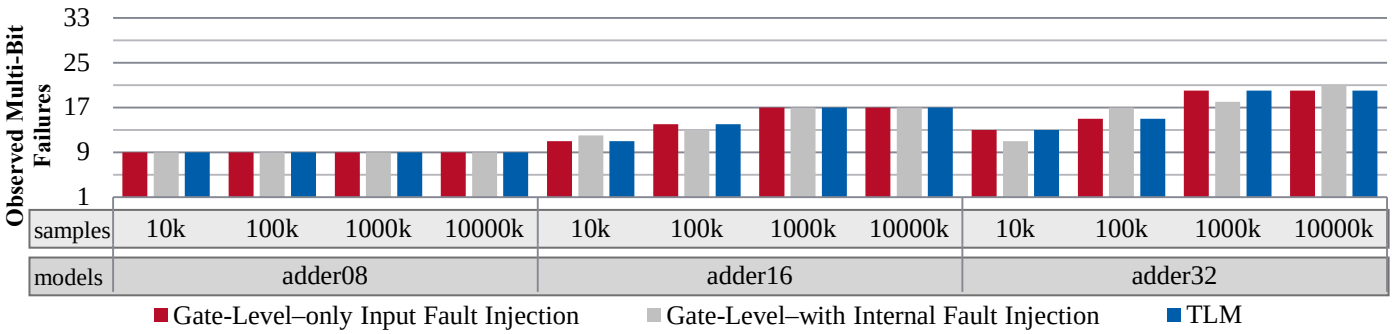

Figure 13. Maximum multi-bit failure observed after injecting stuck-at-0 faults in adders of variable sizes

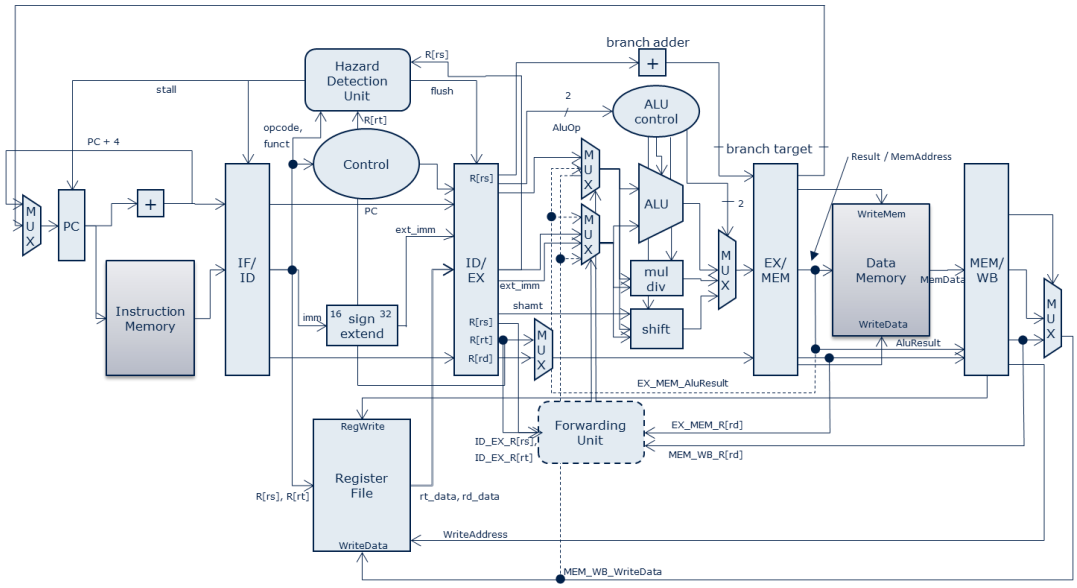

Figure 14. Block diagram of the register-transfer model of a MIPS CPU

Finally, the average number of simulations with random stimuli required to cause a maximum MBF is:

$$N_{avg} = \frac{1}{P} \tag{5}$$

Table I shows the results of applying Eq. 5 on various adder sizes. It is obvious that the probability of obtaining a maximum MBF for a 32-bit adder is intrinsically small (i.e., about $3.58 \cdot 10^{-10}\%$). Therefore, directed testing becomes a powerful method in overcoming this limitation.

B. MIPS CPU Architecture

The MIPS CPU contains a 32-bit integer-based instruction set, hazard detection and forwarding units, and a five-stage pipeline. These features are mirrored within the RTL and TL models. While the `write back` pipeline stage is modeled independently for the RTL implementation (Fig. 14), in the TL model, this stage is contained within the `memory access` (Fig. 15).

After synthesizing the RTL models and transforming the net-lists into SystemC models, we have injected faults into specific sub-modules of the MIPS CPU implementation (e.g., look-up tables, shifters, branching units) to test the performance of our fault-injection method. We have also simulated the whole MIPS CPU (i.e., GL and TL models) to study the effects and propagation of the injected faults.

Similar to the previous case study, 10 000 samples are sufficient to cause at least one single-bit failure (SBF) in all outputs of individual sub-blocks of a MIPS CPU (Fig. 16). Moreover, there is no difference in the results collected after injecting both types of stuck-at faults. The same results have also been observed for the corresponding TL models of the MIPS CPU.

When considering observed MBFs, the results coincide with those from the previous case study (Fig. 17). The more output bits a sub-block has, the more samples are needed to cause a greater MBF (e.g., the write_back block contains 32 output bits, but after 1 000 000 samples, only a 29-bit failure is observed). However, the large amount of samples required to obtain all MBFs for each MIPS sub-block renders the Monte Carlo approach inefficient. For this reason, directed tests are required to cover the missing MBFs more efficiently. Additionally, no significant difference has been observed between the injection of either stuck-at-0 and stuck-1 faults.

Table I. AVERAGE NUMBER OF SIMULATIONS REQUIRED TO CAUSE A MAXIMUM MULTI-BIT FAILURE IN SEVERAL ADDERS

Adder Size	Average Number of Simulations
1	4
8	4318
16	2 162 622
32	279 172 874 110

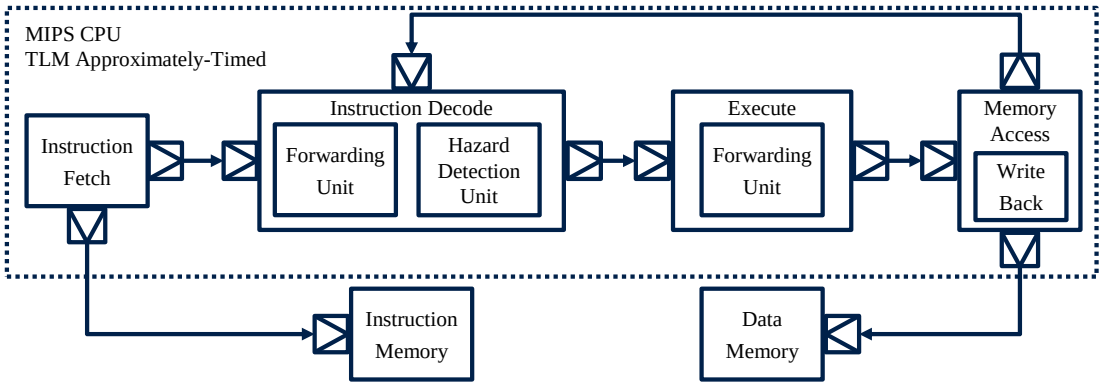

Figure 15. Block diagram of the transaction-level model of a MIPS CPU

C. Fault-Injection Performance

To test the performance of the two fault-injection methods presented in this paper, we have measured the simulation overhead introduced by SCFIT on the GL models of the MIPS CPU (Table II) and by the injectable initiator and target sockets on the CPU's TL models (Table III).

On the one hand, SCFIT presents a slightly lower static simulation overhead than the TLM fault-injection method (i.e., about 5% less). On the other hand, considering the simulation slowdown introduced by the actual fault injection process, SCFIT is greatly outperformed when injecting one or multiple faults (i.e., SCFIT is on average twice slower).

Moreover, the injectable sockets support the injection of any number of faults during a simulation, whereas SCFIT is limited by the target machine's number of debug registers.

Based on the simulation-time measurements of the MIPS CPU, the GL models are about 15 times slower than the true TL models. However, they have much faster simulation speeds than pure GL models (i.e., about three orders of magnitude) and provide the TL model with all GL fault-injection locations (i.e., all matching points). This makes the GL models, although slower, much more robust with respect to fault-effect analysis.

VII. Summary

Our proposed approach enables safety verification of transaction-level (TL) models by verifying the safety-related functionality of system-level models. Moreover, we have shown how matching points are created between TL and gate-level (GL) models by integrating GL-accurate SystemC models into TL models.

The fault-injection methods presented in this paper are useful for injecting faults into models developed on different abstraction levels (i.e., GL and TL models). Thus, by using either SCFIT's GDB-based approach or the TLM injectable initiator and target sockets, fault-effect analysis can be performed quickly and non-intrusively on GL models as well as on TL models. Furthermore, fault models can be reused throughout the verification environment without affecting the models. The simulation slowdown introduced by the fault-injection methods is almost negligible when performing single-bit fault injection.

The results of our case studies show that, given the correct amount of samples, the effect of any single-bit fault injected into the GL model of an adder is reproducible through single-bit fault injection into the inputs of the adder's corresponding TL

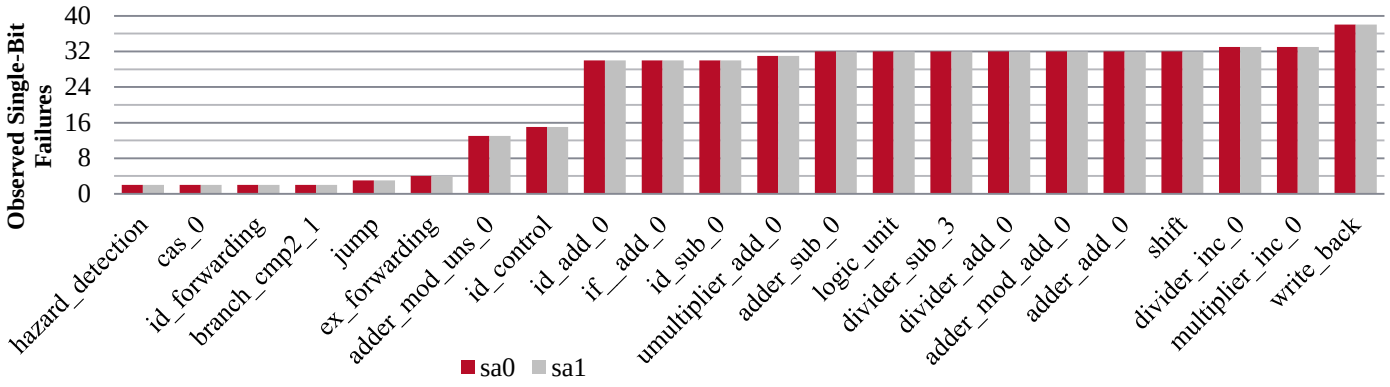

Figure 16. Single-bit failures observed for various MIPS CPU gate-level sub-blocks after injecting permanent faults into 10 000 random samples

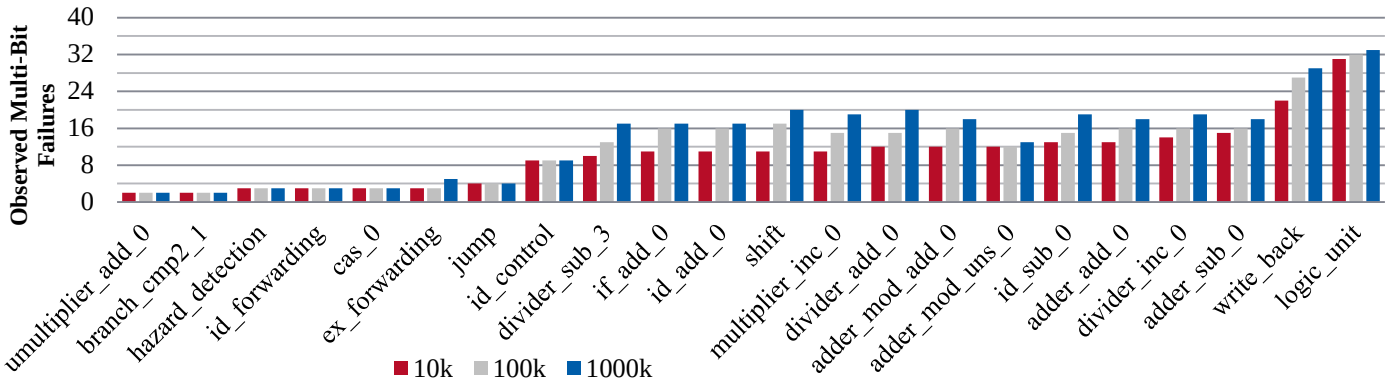

Figure 17. Multi-bit failures observed for several MIPS CPU gate-level sub-blocks after injecting stuck-at-1 faults

model. Therefore, all such faults are correct and realistic at the transaction level.

In the case of adders, stuck-at-0 and stuck-at-1 faults ultimately lead to the same failures, but from different input stimuli. In this case, the injection of either stuck-at-1 or stuck-at-0 faults can be skipped without losing any failure coverage.

Fault injection into virtual prototypes cannot be used to sign-off the safety-verification process of SoCs because of the limited number of fault-injection locations in comparison to the GL net-list. Additionally, the faults injected into virtual prototypes are similar to those injected into GL models, but not identical. However, as shown in this paper, a sufficient amount of failures can be detected early in the SoC's concept phase through TLM-based fault injection.

We have not performed multi-bit fault injection into TL models for two main reasons. First, single-bit fault injection has yielded sufficiently good results (i.e., all single-bit and most multi-bit failures have been observed at the outputs of our TL and GL models). Second, multi-bit fault injection into TL models also causes wrong failures (i.e., failures which cannot be reproduced by single-bit fault injection into GL models).

ACKNOWLEDGMENT

This work is partially supported by the German Federal Ministry of Education and Research (BMBF) in the project EffektiV (contract no. 01IS13022).

REFERENCES

[1] ISO, CD, "26262, Road Vehicles–Functional Safety," *International Standard ISO/FDIS*, 2011.

[2] J.-H. Oetjens, O. Bringmann, M. Chaari, W. Ecker, B.-A. Tabacaru *et al.*, "Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[3] O. S. Initiative *et al.*, "IEEE Standard SystemC Language Reference Manual," *IEEE Computer Society*, 2006.

[4] R. Mariani, "Panel: Functional Safety in the Automotive Value Chain," https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/2015_Conference_Program.pdf, Accessed: 2016-02-03.

[5] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–10.

[6] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[7] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[8] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-Level Simulation with GPU Computing," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 3, p. 30, 2011.

Table II. MEASURED SIMULATION OVERHEAD INTRODUCED BY SCFIT

A	B	C	D	E	F
Model	Faults Injected	Reference Simulation Time (s)	Fault-Injection Simulation Time (s)	Slowdown Factor (D/C)	Slowdown Percentage (100*(1-E))
MIPS CPU	0	3.71	3.77	1.016x	1.6 %
MIPS CPU	1	–	4.10	1.105x	10.5 %
MIPS CPU	2	–	4.50	1.213x	21.3 %
MIPS CPU	3	–	4.61	1.243x	24.3 %
			Average	1.144x	14.4 %

Table III. MEASURED SIMULATION OVERHEAD ADDED BY THE INJECTABLE INITIATOR AND TARGET SOCKETS

A	B	C	D	E	F
Model	Faults Injected	Reference Simulation Time (ms)	Fault-Injection Simulation Time (ms)	Slowdown Factor (D/C)	Slowdown Percentage (100*(1-E))
MIPS CPU	0	248.538	253.706	1.021x	2.1 %
MIPS CPU	1	–	258.934	1.042x	4.2 %
MIPS CPU	2	–	272.149	1.095x	9.5 %
MIPS CPU	3	–	274.137	1.123x	12.3 %
			Average	1.070x	7.0 %

[9] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 221–230.

[10] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A Fault Injection Analysis of Virtex FPGA TMR Design Methodology," in *Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on*. IEEE, 2001, pp. 275–282.

[11] M. Alderighi, F. Casini, S. d'Angelo, M. Mancini, S. Pastore, and G. R. Sechi, "Evaluation of Single Event Upset Mitigation Schemes for SRAM Based FPGAs Using the FLIPPER Fault Injection Platform," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE, 2007, pp. 105–113.

[12] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection," *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 313–322, 2012.

[13] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, "SCFIT: A FPGA-Based Fault Injection Technique for SEU Fault Model," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012, pp. 586–589.

[14] A. K. Ghosh, T. A. DeLong, B. W. Johnson, and J. A. Profeta III, "Fault Injection in the Design Process Using VHDL," in *VHDL International Users' Forum Fall Conference*, 1995, pp. 15–19.

[15] V. Sieh, O. Tschäche, and F. Balbach, "VHDL-Based Fault Injection with VERIFY," *Interner Bericht*, vol. 5, p. 96, 1996.

[16] M. Schwarz, M. Chaari, B.-A. Tabacaru, and W. Ecker, "A Meta-Model-Based Approach for Semantic Fault Modeling on Multiple Abstraction Levels," *DVCon Europe*, pp. 1–6, 2015.

[17] I.-D. Vidrascu, "Implementation of a Safety Verification Environment (SVE) based on Fault Injection," Master's thesis, Fachhochschule Kärnten, Klagenfurt am Wörthersee, Austria, 2015.

[18] R. Leveugle, D. Cimonnet, and A. Ammari, "System-Level Dependability Analysis with RT-Level Fault Injection Accuracy," in *Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004. Proceedings. 19th IEEE International Symposium on*. IEEE, 2004, pp. 451–458.

[19] H. R. Zarandi, S. G. Miremadi, and A. Ejlali, "Dependability Analysis Using a Fault Injection Tool Based on Synthesizability of HDL Models," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*. IEEE, 2003, pp. 485–492.

[20] J. Espinosa, D. de Andrs, J. C. Ruiz, C. Hernandez, and J. Abella, "Towards Certification-Aware Fault Injection Methodologies Using Virtual Prototypes," *Forum on Specification and Design Languages (FDL) – Work in Progress (WiP)*, pp. 1–4, 2015.

[21] J. Espinosa, C. Hernandez, and J. Abella, "Characterizing Fault Propagation in Safety-Critical Processor Designs," in *On-Line Testing Symposium (IOLTS), 2015 IEEE 21st International*. IEEE, 2015, pp. 144–149.

[22] B.-A. Tabacaru, M. Chaari, W. Ecker, and T. Kruse, "Runtime Fault-Injection Tool for Executable SystemC Models," *DVCon India*, 2014.

[23] W. Lu and M. Radetzki, "Efficient fault simulation of SystemC designs," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 487–494.

[24] B.-A. Tabacaru, M. Chaari, W. Ecker, and T. Kruse, "A Meta-Modeling-Based Approach for Automatic Generation of Fault-Injection Processes," *DVCon Europe*, pp. 1–7, 2014.

[25] A. d. Silva, P. Parra, Ó. R. Polo, and S. Sánchez, "Runtime instrumentation of SystemC/TLM2 interfaces for fault tolerance requirements verification in software cosimulation," *Modelling and Simulation in Engineering*, vol. 2014, p. 42, 2014.

[26] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, K. Liu, N. Hatami, C. Novello, H. Post, and A. von Schwerin, "Fault-Injection Techniques for TLM-Based Virtual Prototypes," *Forum on Specification and Design Languages (FDL) – Work in Progress (WiP)*, pp. 1–4, 2015.

[27] V. Guarnieri, N. Bombieri, G. Pravadelli, F. Fummi, and H. e. a. Hantson, "Mutation analysis for SystemC designs at TLM," in *Test Workshop (LATW), 2011 12th Latin American*, March 2011, pp. 1–6.

[28] M. Sousa and A. Sen, "Generation of TLM testbenches using mutation testing," in *Proceedings of the eighth IEEE/ ACM/IFIP international conference on hardware/software codesign and system synthesis*. ACM, 2012, pp. 323–332.

[29] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Comparison of Different Fault-Injection Methods into TLM Models," *Resiliency in Embedded Electronic Systems (REES), 1st International ESWEEK Workshop on*, pp. 1–6, 2015.

[30] T. S. Tan and B. A. Rosdi, "Verilog HDL Simulator Technology: A Survey," *Journal of Electronic Testing*, vol. 30, no. 3, pp. 255–269, 2014.

[31] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," *Volume 3A: System Programming Guide, Part 1*, 2010.

[32] ——, "Intel® 64 and IA-32 Architectures Software Developer's Manual," *Volume 3B: System Programming Guide, Part 2*, 2013.

[33] W. R. Gilks, *Markov Chain Monte Carlo*. Wiley Online Library, 2005.

[34] W. K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains And Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.