2017 IEEE Computer Society Annual Symposium on VLSI

Towards Making Fault Injection on Abstract Models a More Accurate Tool for Predicting RT-Level Effects

Tino Flenker*, Jan Malburg†, Görschwin Fey*†, Serhiy Avramenko‡, Massimo Violante‡, Matteo Sonza Reorda‡
*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Institute of Space Systems, German Aerospace Center, 28359 Bremen, Germany
‡Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy
Email: flenker@cs.uni-bremen.de, {jan.malburg, goerschwin.fey}@dlr.de,
{serhiy.avramenko, massimo.violante, matteo.sonzareorda}@polito.it

***Abstract*—Fault injection and fault simulation are a typical approach to analyze the effect of a fault on a hardware/software system. Often fault injection is done on abstract models of the system either to retrieve early results when no implementation is available, yet, or to speed-up the runtime intensive fault simulation on detailed models. The simulation results from the abstract model are typically inaccurate because details of the concrete hardware are missing.**

Here, we propose an approach to relate faults from an abstract untimed algorithmic model to their counterparts in the concrete register transfer models. This allows to understand which faults are covered on the concrete model and to speed up the fault simulation process.

We use a mapping between both models' variables and mapped timing states for fault injection to corresponding variables on both models. After fault simulations the results are compared to check, whether a given fault produces the same behavior on both models. The results show that an injected fault to corresponding variables leads to the same behavior of both models for a large share of faults.

Keywords* — *fault simulation, fault injection, mapping models

I. Introduction

Hardware systems are omnipresent in our daily life. At the same time they gain in complexity. Depending on the application scenario, a system must continue to work properly even under internal faults. At the same time, progress in circuit technology makes the components more susceptible to faults. E.g., for space applications soft-errors in hardware due to cosmic radiation must be handled. Therefore integrated circuits in space applications must be soft-error resilient. One way to prove the system's resilience to soft-errors is the use of proton- or neutron-beams. This approach, however, has several drawbacks. First of all, the validation of a design against soft-errors should be done as early as possible, but the evaluation of soft-errors based on proton- or neutron-beams requires silicon chips. Second, such tests are expensive and may not yield very much information about the part of the design, that requires additional hardening. A widely used alternative is *register transfer level* (RTL) fault simulation, where artificial bit-flips are introduced into flip-flops and latches of the design. Sanda et. al. showed that fault simulation models work very well for radiation induced soft-errors [1].

But modern hardware designs get more complex and typical fault simulation on RTL becomes computationally expensive. For example, the authors of [2] used several FPGA emulation systems and a petascale computing system to evaluate different soft-error protection schemes. Fault simulation on a higher level of abstraction can largely reduce the computational cost. However, existing higher level fault simulation does badly correspond to RTL fault simulation [3].

The technique presented in this paper ultimately aims to predict the effect of faults on RTL by fault injection on abstract level. As a first step we study how to determine correspondences between fault injection in abstract models and fault injection in RTL models. This is a challenging problem as typically the results for the two levels may differ arbitrarily [3]. However, in certain situations results can be correlated [4]. Thus, drawing conclusions about the behavior of the RTL model by fault simulation of the corresponding abstract model is a challenging task.

We propose an approach based on a mapping between variables of the abstract and the RTL model. Based on this mapping we correlate the results of abstract and RTL fault simulation. The approach works even though timing information on the abstract level is absent. To achieve this, for both hardware models *synchronized* fault lists are generated. The synchronization is done by state information of a given set of variables present in both models. After this, fault simulation is performed and the faults are categorized which allows to find equal behavior for corresponding parts in the models. Our experimental results show that our mapping approach allows to predict the results of the RTL fault simulation quite accurately.

This paper has the following structure. In Section II the presented work is compared with related work. Section III depicts the overall flow of this work. Sections IV and V explain how fault list synchronization is done and how faults are categorized, respectively. The evaluation metric is presented in Section VI. Next, Section VII discusses the advantages and limitations of our approach. Finally, the experimental results are described in Section VIII and at the end a conclusion is given in Section IX.

This work was supported in part by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative and the German Research Foundation (DFG, grant no. FE 797/6-2), by the European Union (IMMORTAL project, grant no. 644905) and has received funding from the European Union's Horizon 2020 research and innovation programme (grant agreement no. 637616).

2159-3477/17 $31.00 © 2017 IEEE
DOI 10.1109/ISVLSI.2017.99

II. Related work

For evaluating a design with respect to robustness against soft-errors, for example introduced by radiation, the system is simulated while artificial faults are introduced. These faults can be introduced at different levels of abstraction. One level typically used is the RTL. At this level, storage elements are randomly flipped to simulate the effect of radiation on the real chip [5]. Sanda et. al. showed that RTL fault injection very closely relates to real radiation caused faults [1]. However, this type of simulation is computationally expensive.

An approach to reduce the computational cost is to commence fault injection at a higher level of abstraction. There are several techniques that evaluate the effect of soft errors on higher levels of abstraction, e.g., [4], [6], [7], or [8].

In case of software level fault injection, the value of program variables or contents of the heap memory can be changed [8]. The main advantages of software level fault injection is the speed and that no knowledge of the underlying hardware is needed.

Instruction set architecture (ISA) based fault injection [7] introduces errors by changing the value of ISA registers of the design. Compared to the source code level fault injection, this means that stack-pointers and return addresses for functions may be affected, resulting in irregular program flow. ISA based fault injection is typically done by using a debugger or instrumenting the test program at assembly level.

Micro-architecture level soft-error injection uses a model of the design under test [6]. This model includes not only the ISA visible part of the design, but also internal storage elements, like caches, load/storage queues, issue queues or branch prediction logic. The faults can be introduced in all of those memory elements leading to reading from or writing to incorrect cache lines or performance penalty due to incorrect branch prediction.

All the above mentioned abstraction approaches have in common that they do not take into account the relation between soft-errors happening in real systems and the abstracted system. This can lead to large deviation between the behavior of the real system and the abstracted system with respect to soft errors [3]. The approach presented in this paper mitigates this difference by relating soft-errors to corresponding abstracted faults.

The authors in [4] evaluate the effect of soft-errors on lossless compression algorithms. In their study they evaluate the robustness by source code level fault injection and ISA level fault injection. The authors do not deny that the source code level fault injection is highly inaccurate, indeed they focus on relative comparison to rank compression algorithms in term of robustness. Furthermore the figures obtained by source code level fault injection are processed using some hardware dependent information in order to achieve more realistic estimations. In contrast to our approach, the authors of [4] want to identify the best option among a set of candidates of the same class of algorithms, considering compression algorithms as a case of study. Further they consider the early phases of the design flow in which all the details about final hardware are not available yet, thus RTL fault injection is not an option in their case.

In [9] a combination of gate level and timed behavioral level fault simulation is presented. In that approach each module of the design is provided at gate level as well as behavioral level. The fault injection is applied to the gate level model and the effect on the corresponding module's outputs is computed. Those outputs then are used to drive the behavioral models of the modules not affected by faults. Compared to our approach, this approach allows only a limited amount of abstraction as the faulty modules must still be simulated at gate level, further their behavioral models must still be timed models, where our approach allows any level of abstraction.

The RAVEN tool [10] computes soft error vulnerability estimation by first partitioning the design into small blocks. Then they do fault simulation of these blocks with respect to faults at their primary inputs to compute the probability that a block propagates a fault at one of its inputs towards its outputs. Additionally, RAVEN uses fault simulation to compute the probability of the block propagating SETs and SEUs appearing inside the block to its outputs. In the last step those measured probabilities are used to compute an overall estimation of the circuit vulnerability against soft errors. The RAVEN tool is orthogonal to the approach presented in this paper.

III. Overall flow

This section explains the overall flow of our approach, schematized in Fig. 1.

First, a partial mapping between the variables of the abstract and the RTL model is required. This partial mapping can be given as an input or derived by a heuristic [11]. One requirement to the partial mapping is that the variables defining the architectural state of both models have a good matching. The architectural state is defined by a subset of the models' variables which are mappable from the abstract to the RTL model. This is required because we want to find corresponding effects of faults in both models which requires the two models to be synchronized with respect to their behavior. The output produced by the models is considered as behavior.

Second, simulation on both levels of abstraction is performed to validate the method. For fault list generation ① a *golden* run simulating the same use case on both models is mandatory, because the same use case on both models produces the same results on both levels. For example, calculating the

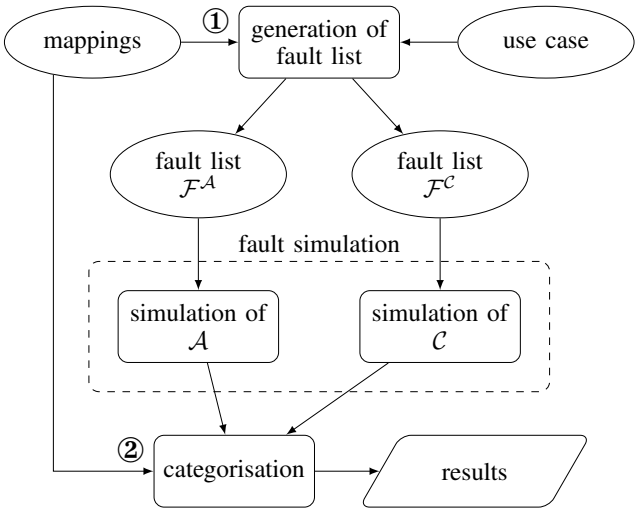

Figure 1. Overall flow for fault injection and categorization process

Fibonacci numbers on both models gives the same numbers on both levels. The golden run produces a trace from which we retrieve values of the variables and timing information. In addition, a memory dump at the end of the golden run is stored. The golden traces and the mapping enable generating *synchronized* fault lists. Two synchronized fault lists map a fault from the abstract model to a corresponding fault in the RTL model. Section IV provides details about the synchronization.

Next, the generated fault lists are used for fault simulation. For each fault on the corresponding list, a simulation on each model is executed. At the end of the simulation a memory dump is stored in a separate file. The next step, described in Section V, is the categorization of the faults ②. The memory dump resulting from the golden run is used to determine the differences in the resulting data.

The results yield an insight whether the mapping between variables is useful to predict actual fault effects on RTL by the fault simulation of the abstract model.

IV. Fault list synchronization

This section describes the process for the fault list synchronization. At first, all needed definitions for the fault list are presented. Finally, the fault list generation is shown which also ensures the synchronization of the fault list for the abstract and the RTL model. The synchronization is for considering the architectural states in two models. Thus, two fault lists for models of different abstraction levels are called *synchronized*, if

- both lists contain the same faults and
- the faults are injected, when the models have the same architectural state.

To simplify the presentation we consider all variables in the RTL model and in the abstract model to be bit vectors. The extension to other types of variables is straightforward and handled by our implementation. An exception are pointer which are not considered in this approach. The sets $\mathcal{V}^{\mathcal{A}}$ and $\mathcal{V}^{\mathcal{C}}$ contain the variables of the abstract model $\mathcal{A}$ and the RTL model $\mathcal{C}$, respectively. In the following, registers and signals out of $\mathcal{C}$ are also called variables. For a given variable $v \in \mathcal{V}^{\mathcal{A}} \cup \mathcal{V}^{\mathcal{C}}$, `bitsize(`$v$`)` returns the number of bits of that variable.

For fault simulation a fault location $l \in \mathcal{L}$ is defined as a tuple (v, b) of a variable v out of the set of the models variables $\mathcal{V}$ and one variable index b which specifies the intended bit v_b to be flipped in the fault simulation.

$$l := (v, b) \text{ with } v \in \mathcal{V} \text{ and } 0 \leq b < \texttt{bitsize}\,(v)$$

The set $\mathcal{V}$ represents $\mathcal{V}^{\mathcal{A}}$ and $\mathcal{V}^{\mathcal{C}}$ as a generic placeholder. The domains $\mathcal{L}^{\mathcal{A}}$ and $\mathcal{L}^{\mathcal{C}}$ contain all fault locations of the abstract and the RTL model, respectively, where the fault locations $l \in \mathcal{L}^{\mathcal{A}}$ contain the variables v out of $\mathcal{V}^{\mathcal{A}}$ and analogously the variables in the fault locations $l \in \mathcal{L}^{\mathcal{C}}$ are from $\mathcal{V}^{\mathcal{C}}$.

Next, to get the correspondence between the variables of the models on the different abstraction levels, a mapping function is needed. Thus, the function map is introduced which represents a partial mapping for the correspondences between the variables from the abstract model $\mathcal{V}^{\mathcal{A}}$ to the variables of the RTL model $\mathcal{V}^{\mathcal{C}}$ if exist.

$$map : \mathcal{V}^{\mathcal{A}} \longmapsto \mathcal{V}^{\mathcal{C}}$$

This function is used for the calculation of the points of time for the fault injection on abstract level.

Subsequently, the identification of a point of time is mandatory which indicates when a given fault is injected. The definitions for a fault's point of time are necessarily different for the two models, because on the abstract level no accurate timing information is present but on RTL this information exists.

To get the points of time $\mathcal{T}^{\mathcal{C}}$ for the fault injection on RTL, the timing information of the RTL trace, e.g., in simulation cycles, is used. So all values in $\mathcal{T}^{\mathcal{C}}$ are bounded between 0 and `Cyc` where `Cyc` is the number of clock cycles for the golden simulation of $\mathcal{C}$.

$$\mathcal{T}^{\mathcal{C}} := \{t \mid 0 \leq t \leq \texttt{Cyc}\}$$

The abstract model does not have a time reference. Instead the architectural states are used to construct a *time reference* for the fault injection. Later these two time references are used to synchronize faults. The function tr_c for a trace of the RTL model indicates the value assigned to a variable at a specific point of time. The RTL trace is a mapping of a variable out of $\mathcal{V}^{\mathcal{C}}$ and a point of time out of $\mathcal{T}^{\mathcal{C}}$ to an assigned value out of $\mathbb{N}$:

$$tr_c : \mathcal{V}^{\mathcal{C}} \times \mathcal{T}^{\mathcal{C}} \longmapsto \mathbb{N}$$

The trace is required by the determination of the architectural states of the abstract model which is used for the fault list. For the determination the trace of the RTL model is used, because this trace is more detailed in comparison to the trace of the abstract model.

The unique points of time for the abstract model are stored in a sequence defined as follows:

$$\mathcal{T}^{\mathcal{A}} := (\hat{v}_{11}, \cdots \hat{v}_{n1}, c_1), \ldots (\hat{v}_{1t}, \ldots \hat{v}_{nt}, c_t)$$

The $\mathcal{T}^{\mathcal{A}}$ is a sequence of the values of a flexible number of variables n. The sequence only adds a new tuple, if the considered tuple has a changed value $\hat{v}_{ij}$ of any variable in comparison to the predecessor tuple. Thus, $\hat{v}_{ij} \neq \hat{v}_{ij-1}$ holds for at least one j. To each tuple the number of occurrences c_j of the current combination of values $\hat{v}_{ij}$ of the variables is appended. The count is used for an exact determination of the point of time in the simulation of $\mathcal{A}$.

Example 1. Considering a data bus model. The model has the primary input $DataIn$ and the primary output $DataOut$. The values in the tuple $(DataIn, DataOut, c_j)$ occur in the following sequence:

$$(0,0,1)(2,0,1)(0,0,2)(0,2,1)(2,2,1)$$

The first two values of the tuples above show the values of the given variables. During the golden simulation first the values of $DataIn$ and $DataOut$ are 0. Because this is the first occurrence of this combination in the trace, the counting value 1 is appended to the tuple. In the next tuple the value of $DataIn$ is changed and the combination is a new one. Therefore, a 1 is appended again. The third tuple consists of the values $(0,0)$ again which is a combination that already occurred one time before, so the count value of 2 is added.

The function `calc_timing()` in Algorithm 1 shows how unique points of time for the abstract model's fault list are computed.

Algorithm 1 Computation of timing values for fault list of $\mathcal{A}$

```
1:  function calc_timing(variables, tr_c, map)
2:      Frame last ← [X, ..., X]
3:      t ← 0
4:      for t ≤ Cyc do
5:          Frame frame
6:          for all absVar ∈ variables do
7:              RTLvar ← map[absVar]
8:              frame.append(tr_c(RTLvar, t))
9:          if frame ≠ last then
10:             count ← counts[frame]
11:             T^A.append(frame, count)
12:             counts[frame] += 1
13:             last ← frame
14:         t ← t + 1
15:     return T^A
```

The input of `calc_timing()` is a list of variables of the abstract model ($variables \subseteq \mathcal{V}^{\mathcal{A}}$). The variables of the abstract model are used to determine their points of time. In addition to this, the trace of the golden RTL simulation tr_c is given. Also the mapping (*map*) between the variables of the abstract model and the RTL model is given. The generation of the points of time is done with the trace tr_c instead of using the trace of the abstract model because the RTL trace is cycle accurate and it allows due to concurrent execution multiple changes of the values in one time step. On the other side the abstract trace only changes the values step by step. In some cases the abstract trace has some value combinations which do not exist in the RTL trace and vice versa.

The results of `calc_timing()` are the points of time $\mathcal{T}^{\mathcal{A}}$ for the fault injection on the abstract model. In the next step, the values of $\mathcal{T}^{\mathcal{A}}$ are compared with the trace of the golden simulation of the abstract model and all combinations not included in the trace are removed. This reduces the number of fault injections but guarantees a mapping between the fault locations of both models for the simulation.

Given the time references, the fault lists are synchronized as explained in the following. Initially, the definitions for the different fault lists are shown.

$$\mathcal{F}^{\mathcal{A}} \subseteq \mathcal{L}^{\mathcal{A}} \times \mathcal{T}^{\mathcal{A}}$$
$$\mathcal{F}^{\mathcal{C}} \subseteq \mathcal{L}^{\mathcal{C}} \times \mathcal{T}^{\mathcal{C}}$$

Each fault list is a subset of the cross product of all fault locations ($\mathcal{L}^{\mathcal{A}}$ or $\mathcal{L}^{\mathcal{C}}$) and a point of time of the corresponding abstraction level ($\mathcal{T}^{\mathcal{A}}$ or $\mathcal{T}^{\mathcal{C}}$).

Next, the function *lmap* maps a given fault location $l^{\mathcal{A}} \in \mathcal{L}^{\mathcal{A}}$ with $l^{\mathcal{A}} = (v_a, b_a)$ to the corresponding fault location $l^{\mathcal{C}} \in \mathcal{L}^{\mathcal{C}}$ with $l^{\mathcal{C}} = (v_c, b_c)$.

$$lmap(l^{\mathcal{A}}) := l^{\mathcal{C}} \quad \text{with} \quad v_c = map(v_a) \wedge b_a = b_c$$

The function *tm* represents a mapping of a point of time in $\mathcal{T}^{\mathcal{C}}$ to the corresponding tuple in $\mathcal{T}^{\mathcal{A}}$. This helps to keep the fault lists synchronized.

$$tm : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$$

Fig. 2 helps to explain how the synchronization of $\mathcal{F}^{\mathcal{A}}$ and $\mathcal{F}^{\mathcal{C}}$ is done. The fault list for the abstract model $\mathcal{F}^{\mathcal{A}}$ is shown

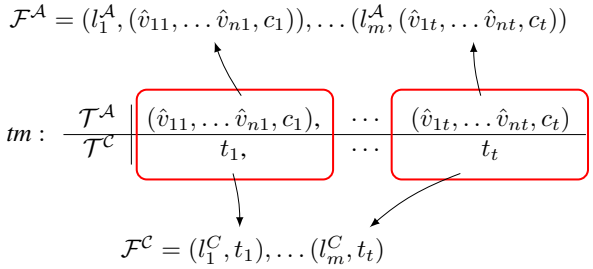

Figure 2. Synchronization of fault lists

on the top. In the center the time mapping *tm* between the points of time is shown. Below the mapping is the resulting fault list for the RTL model $\mathcal{F}^{\mathcal{C}}$. The first element in $\mathcal{F}^{\mathcal{A}}$ and $\mathcal{F}^{\mathcal{C}}$ respectively consists of the first selected fault location $l_1^{\mathcal{A}} \in \mathcal{L}^{\mathcal{A}}$ and a randomly selected corresponding point of time. In $\mathcal{F}^{\mathcal{C}}$ it is t_1 and in $\mathcal{F}^{\mathcal{A}}$ it is $tm(t_1)$. In Fig. 2

$$tm(t_1) := (\hat{v}_{11}, \ldots \hat{v}_{n1}, c_1) \text{ where } \hat{v}_{11}, \ldots \hat{v}_{n1}$$

are the values of the considered n variables out of $\mathcal{V}^{\mathcal{A}}$ at the corresponding point of time t_1 in $\mathcal{C}$. The value c_1 is the count value for the exact determination of the point of time in $\mathcal{A}$ because it is possible that the values $\hat{v}_{11}, \ldots \hat{v}_{n1}$ occur multiple times. For the fault locations $lmap(l_i^{\mathcal{A}}) = l_i^{\mathcal{C}}$ holds where $0 < i \leq m$ and m is the number of faults in the list. Both fault lists contain the same corresponding fault locations with the same corresponding points of time. Thus, $\mathcal{F}^{\mathcal{A}} = (l_1^{\mathcal{A}}, tm(t_1)), \ldots, (l_m^{\mathcal{A}}, tm(t_t))$ and $\mathcal{F}^{\mathcal{C}} = (lmap(l_1^{\mathcal{A}}), t_1), \ldots, (lmap(l_1^{\mathcal{A}}), t_t)$. Fig. 2 shows that to each fault in the corresponding fault list and depending on the given model the same associated point of time is assigned.

As a result, both fault lists contain the same faults with the same points of time and are synchronized. The next step shown in Fig. 1 is the fault simulation. The results of the simulation are used for categorization described in the next section.

V. Fault Categorization

This section describes how fault categorization is done. First, we explain how the faults are divided in general and afterwards how two corresponding faults are categorized.

After fault simulation is done, the faults need to be checked if the faults on both abstraction levels yield the same effect. In case of a good mapping between the synchronized fault lists simulation on both levels should yield the same results, otherwise the results may be arbitrarily different. In case of similar results it is possible to argue about the RTL model given the results for fault simulation on the abstract level.

Fault injection and simulation obtain a partition of all faults out of $\mathcal{F}^{\mathcal{A}} \cup \mathcal{F}^{\mathcal{C}} = \mathcal{F}$ into the two sets *unknown* $\mathcal{F}_{unkn}$ and *critical* $\mathcal{F}_{crit}$. For the sets $\mathcal{F}_{crit} \cap \mathcal{F}_{unkn} = \emptyset$ holds.

The faults assigned to $\mathcal{F}_{unkn}$ are called *unknown*. A fault is called unknown if the fault causes no observable effect by the given test stimuli but it is also possible that the fault is critical with another set of test stimuli. The second set of faults is called *critical* ($\mathcal{F}_{crit}$). A fault is called critical if the effect of the fault is observable.

Two given faults $(l_a, t_a) \in \mathcal{F}^{\mathcal{A}}$ and $(l_c, t_c) \in \mathcal{F}^{\mathcal{C}}$ with $lmap(l_a) = l_c$ and $t_a = tm(t_c)$ are assigned to one of the following categories:

different: One fault is assigned to $\mathcal{F}_{unkn}$ and the other to $\mathcal{F}_{crit}$.
similar: Both faults are assigned to the same set of faults.
equal: Both faults are assigned to $\mathcal{F}_{unkn}$ or both faults are assigned to $\mathcal{F}_{crit}$ and also produce equal observable results in fault simulation.

All other fault pairs, which do not fulfill the above mentioned conditions are not considered, because these pairs do not correspond to each other.

VI. Evaluation metric

This section presents the evaluation metric used for our approach.

First, the faults of the abstract model are divided to $\mathcal{F}^{\mathcal{A}}_{unkn}$ and $\mathcal{F}^{\mathcal{A}}_{crit}$ by fault simulation. The unknown faults of the abstract model are represented by $\mathcal{F}^{\mathcal{A}}_{unkn}$ and the critical faults are in $\mathcal{F}^{\mathcal{A}}_{crit}$. The sets for the RTL model are divided to $\mathcal{F}^{\mathcal{C}}_{unkn}$ and $\mathcal{F}^{\mathcal{C}}_{crit}$ and have the corresponding meaning to the sets of the abstract model.

The function *fmap* maps a given fault $f_a \in \mathcal{F}^{\mathcal{A}}$ with $f_a = (l_a, t_a)$ to the corresponding fault $f_c \in \mathcal{F}^{\mathcal{C}}$ with $f_c = (l_c, t_c)$.

$$fmap(f_a) := f_c \quad \text{with} \quad l_c = lmap(l_a) \wedge t_a = tm(t_c)$$

The set $\mathcal{F}^{\mathcal{A}}_{\mathcal{M}}$ is a subset of $\mathcal{F}^{\mathcal{A}}$ and includes all faults of the abstract model which can be mapped to the RTL model. Thus, $\mathcal{F}^{\mathcal{C}}_{\mathcal{M}}$ is the set of the corresponding mapped faults out of the RTL model which are considered on abstract level.

$$\mathcal{F}^{\mathcal{C}}_{\mathcal{M}} := \{f_c \in \mathcal{F}^{\mathcal{C}} \mid f_a \in \mathcal{F}^{\mathcal{A}}_{\mathcal{M}} \wedge fmap(f_a) = f_c\}$$

The faults about which no conclusions can be drawn are in the set $\mathcal{F}^{\mathcal{C}} \setminus \mathcal{F}^{\mathcal{C}}_{\mathcal{M}}$. That are for example faults which do not exist for the abstract model because these faults affect variables which have no corresponding variables in the abstract model.

To denote the share of all considered faults of the abstract and the RTL model the metrics and sh_a and sh_c are used.

$$sh_a = \frac{\mid \mathcal{F}^{\mathcal{A}}_{crit} \cup \mathcal{F}^{\mathcal{A}}_{unkn} \mid}{\mid \mathcal{F}^{\mathcal{A}} \mid}$$

The metric sh_c is defined analogously to sh_a.

In the following the sets $\mathcal{F}^{\mathcal{C}}_{crit}, \mathcal{F}^{\mathcal{C}}_{unkn}$ are subsets of $\mathcal{F}^{\mathcal{C}}_{\mathcal{M}}$ and $\mathcal{F}^{\mathcal{A}}_{crit}, \mathcal{F}^{\mathcal{A}}_{unkn}$ are subsets of $\mathcal{F}^{\mathcal{A}}_{\mathcal{M}}$, respectively. Thus, only faults injected to mappable variables are considered.

VII. Discussion

This section summaries the advantages and limitations of our approach.

First, some limitations we need to consider are discussed. An important role for appropriate results has the mapping of variables between the models which is given as an input. No useful results will be obtained, if the mapping does not give good correspondences between both models. In addition, in our case the abstract model does not have any timing information and does not have structural similarities with the RTL model. The mapping is not able to give correspondences for missing structures in the abstract model. Nevertheless, structural information which exists in both models enables a useful mapping. The absent timing information in the abstract model makes a synchronization method essential. An insight is already given in Section IV.

Next, the advantages are discussed. In [3] the authors do fault injection on different levels of abstraction. However, their results show that the setting does not yield conclusions regarding the RTL model by simulation on the abstract level. The authors in [9] also combine RTL model simulation with a more abstract behavioral model, however, their approach requires that the abstract model is clock synchronous with the RTL level which allows only a limited abstraction. Our approach is capable of finding corresponding points of time in different abstraction levels despite the timing information is not available on abstract level. As result the next advantage is that synchronized fault lists enable the possibility to get conclusions about the RTL model by results of the fault injection simulation on abstract level.

VIII. Experimental Results

This section presents the experimental results of the presented approach. The experiments are applied on the y86 processor[1]. This is a small implementation of a processor, which supports a subset of the x86 instruction set and also exists on RTL and as a C++ implementation on abstract level.

The fault injection on abstract level is performed by a GDB script using the python API which observes the variables of the architectural state and flips the bit if the correct point of time is reached. On RTL the fault injection is done by QuestaSim.

For calculating the points of time for the abstract model four variables are used (opcode, instruction pointer, two registers (`eax`, `ebx`)). As use case, an assembler program with 15 instructions for calculating the Fibonacci numbers is used. As result 29 points of time are obtained, which exist in both models.

Next, for fault generation five variables (`opcode`, instruction pointer (`IP`), `eax`, `ebx` and the zero flag (`ZF`)) are chosen. For these variables the corresponding variables in both models are known. The total number of faults is the sum of all bits for all variables. By this, all possible faulty cases are considered. But to reduce the sum of faults, for example for a register variable, only one fault for one bit is generated. This is because, it does not matter which bit is flipped, no effect to the control flow is caused. For `opcode` a bit flip for all bits is generated because the effect can be different for each fault. In sum 609 corresponding faults are generated for the models.

The share of the considered faults for the abstract model is $sh_a = 54\%$ and for the RTL model is $sh_c = 9\%$. The share of 9% is due to the small number of variables in the abstract model in comparison to the number of variables in the RTL model. Thus, a small number of variables can be mapped between the models which also reduces the total number of possible considerable faults between the models.

To analyze the effect of the faults on both models golden simulation without fault injection is performed. At the end of the simulation the memory is dumped. Next, for each simulation with fault injection the memory is also dumped after execution. Subsequently, the memory dumps are compared.

[1] http://www.digitaltechnik.org/examples/Y86_seq.zip

Table I. CLASSIFICATION FOR THE FAULTS OF THE Y86 MODELS

	$\mathcal{F}^{\mathcal{A}}_{crit}$	$\mathcal{F}^{\mathcal{C}}_{crit}$	$\mathcal{F}^{\mathcal{A}}_{unkn}$	$\mathcal{F}^{\mathcal{C}}_{unkn}$
#	311	355	298	254
%	51	58	49	42

Table II. CATEGORIES FOR THE FAULTS OF THE Y86 MODEL

	different	*similar*	*equal*	$\mathcal{F}_{unkn}$	$\mathcal{F}_{crit}$
#	146	463	388	185	203
%	24	76	64	30	33

Table I presents how the faults are classified on both levels of abstraction. The line with # in the first column shows the total number of critical or unknown faults. The next line with % in the first column shows the percentage of the classified faults. The abstract model has 298 (49%) faults where the result for the given use case is unknown ($\mathcal{F}^{\mathcal{A}}_{unkn}$). The numbers show that the effect of the faults is observable more frequently on RTL.

How two corresponding faults are categorized is shown in Table II. The headline shows the category presented in the given column. The second line shows the number of pairs of faults classified to the given category. The percentage is presented in the last line of Table II. Categorized as *different* are 146 pairs of faults which mean on abstract level the fault is in $\mathcal{F}_{unkn}$ and on RTL the fault is in $\mathcal{F}_{crit}$ or vice versa. The remaining pairs of faults are classified as *similar* which means for both models both faults are in $\mathcal{F}_{unkn}$ or in $\mathcal{F}_{crit}$, respectively. To be precise, the similar pairs are classified with a percentage of 84% as *equal*. For these 388 memory dumps equal dumps are obtained. From that 53% (33% of all fault pairs) of the faults are critical ($\mathcal{F}_{crit}$) with an identical observed behavior in the memory dumps. Of the equal pairs 48% (30% of all fault pairs) are classified as not observable ($\mathcal{F}_{unkn}$). These faults produce no observable behavior on either level of abstraction in the memory dumps.

The category of the fault pairs per variable is presented in Table III. The first column shows the considered faulty variable. The second column contains the number of faults considered for the given variable. The next two columns show the percentage of how many faults are categorized as *different* and *similar*, respectively. The next column shows the percentage of *equal* categorized fault pairs. The last two columns show how the splitting of *equal* fault pairs. The columns show the percentage of observable ($\mathcal{F}_{crit}$) and not observable ($\mathcal{F}_{unkn}$) fault pairs.

The results for the `opcode` variable show that 52% of the fault pairs have *equal* behavior. For all of these fault pairs no faulty behavior can be observed in both models. All faults on the abstract level are not observable by this given use case. For 48% of the faults on RTL a different behavior can be observed which is the cause for the 48% of the *different* classified fault pairs. The register variables (`eax`, `ebx`) show a different behavior in fault simulation for 28% and 24%, respectively. Thus, 55% and 66%, respectively, of the fault pairs have an *equal* behavior. For the zero flag register (`ZF`) 100% of the fault pairs are *equal* and all faults are not observable in both models. The fault pairs for the instruction pointer (`IP`) have *different* behavior for 7% but a large proportion of 93% with a *similar* behavior which is a good result for an early work in this topic. More than the half of the fault pairs (70%) cause an *equal* behavior in both models

Table III. CATEGORIES DIVIDED BY FAULTY VARIABLE

	$\lvert\mathcal{F}\rvert$	*different*	*similar*	*equal*	$\mathcal{F}_{crit}$	$\mathcal{F}_{unkn}$
`opcode`	232	48	52	52	0	52
`IP`	290	7	93	70	57	13
`eax`	29	28	72	55	38	17
`ebx`	29	24	76	66	28	38
`ZF`	29	0	100	100	0	100

where 81% (57% of all fault pairs) of this are observable faults ($\mathcal{F}_{crit}$). Thus, 81% of all observable fault pairs with an equal behavior produce an equal memory dump on both models.

For an early approach to observe the same behavior on an abstract level without timing information and on RTL the presented results are pretty good. We can show that for a relatively large proportion (76%) of the considered faults at least a similar behavior is observable. A percentage of 64% of identical observed behavior is a promising result for an early approach in this direction of research.

IX. CONCLUSION

The results in Section VIII show that with a given mapping of variables from the abstract model to registers and signals in the RTL model and synchronized faults for both models a high rate of similar behavior of 76% is achieved.

As future work a bigger benchmark model can be used. Besides, the influence of a fault can be analyzed to reduce the number of faults to be checked. This is supposed to yield more performance improvements.

REFERENCES

[1] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones, "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development*, vol. 52, pp. 275–284, 2008.

[2] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores," in *Design Automation Conference*, 2016, pp. 1–6.

[3] 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*, 2013, pp. 1–10.

[4] S. Avramenko, M. Sonza Reorda, M. Violante, and G. Fey, "A high-level approach to analyze the effects of soft errors on lossless compression algorithms," *Journal of Electronic Testing*, vol. 33, pp. 53–64, 2017.

[5] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," in *International Conference on Dependable Systems and Networks With FTCS and DCC*, 2008, pp. 122–127.

[6] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *International Symposium on Workload Characterization*, 2015, pp. 172–182.

[7] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *International Conference on Dependable Systems and Networks*, 2010, pp. 557–562.

[8] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *International Conference on Dependable Systems and Networks*, 2010, pp. 431–436.

[9] S. Mirkhani, M. Lavasani, and Z. Navabi, "Hierarchical fault simulation using behavioral and gate level hardware models," in *Asian Test Symposium*, 2002, pp. 374–379.

[10] S. Mirkhani, S. Mitra, C. Y. Cher, and J. Abraham, "Efficient soft error vulnerability estimation of complex designs," in *Design, Automation and Test in Europe*, 2015, pp. 103–108.

[11] T. Flenker and G. Fey, "Mapping abstract and concrete hardware models for design understanding," in *Design and Diagnostics of Electronic Circuits and Systems*, 2017, pp. 20–25.