An Automated Setup for Large-Scale Simulation-Based Fault-Injection Experiments on Asynchronous Digital Circuits

Patrick Behal, Florian Huemer, Robert Najvirt, Andreas Steininger
Institute for Computer Engineering, TU Wien, Vienna, Austria
{pbehal,fhuemer,rnajvirt,steininger}@ecs.tuwien.ac.at

***Abstract*—Experimental fault injection is an essential tool in the assessment and verification of fault-tolerance properties. Often, in these experiments it is impossible to reasonably cover the huge parameter space spanned by target state and fault parameters, and compromises or restrictions must be made. This is even more pronounced for asynchronous circuits where a convenient discretization of time through a synchronous clock is not possible. In this paper we present a fault-injection toolset that allows for a very efficient injection and data processing, thus bringing studies with many billions of meaningful injections into asynchronous targets within reach. The key ingredients of our solution are an auto-setup feature capable of optimizing parameter values, seamless distribution of the simulation load to many host computers, and efficient arrangement of the important settings and readings in a database. We will use the example of a comparative study of different asynchronous pipeline styles to motivate the need for such an approach and illustrate its benefits.**

***Index Terms*—asynchronous circuits, fault injection, fault-tolerance assessment, tool chain**

I. Introduction

Fault injection experiments can be considered the test of fault-tolerance capabilities. By injecting faults into the running system, the proper function of provisions intended to cope with these faults can be verified and deficiencies be tracked (“fault removal”). In addition, when injecting faults that are foreseen to occur in the actual operating environment (like frequent single-event upsets through particle hits in space), even if not all of these can be handled, statistical predictions on the failure rate of the target can be derived (“fault forecasting”). In both cases the process is the same: Faults are injected into the running system, and the reaction is observed and categorized.

Typically, fault injection studies suffer from two types of problems: Firstly, for precisely targeting the injected faults it is necessary to have **access to nodes or transistors** inside the target. Similarly, a detailed observation of the target’s reaction to a fault requires monitoring of internal signals. Secondly, the **parameter space** that needs to be considered for a complete, or at least representative picture is huge, since, ideally, all conceivable types of faults shall be injected into all locations inside the target during all states of the target.

The former problem can be largely solved by simulation-based fault injection. This, however, aggravates the second issue, as experiments run significantly slower in simulation than in real time. Hence a good coverage of the solution space is even harder to attain within reasonable time limits. On top of those issues that are common to most fault-injection studies, our setting is different through a number of points that make the experiments even more challenging:

- The fact that our targets are asynchronous (quasi delay-insensitive (QDI)) circuits, requires us to have a continuous (i.e., non-discretized or cycle-based) view of time. When checking for deviations from the expected behavior, event traces must be compared irrespective of their absolute timings (i.e., only with respect to the QDI protocol). This stands in sharp contrast to synchronous systems, where it is relatively easy to capture and to compare states. For the same reason, the injection of faults (injection time and fault duration) cannot simply be aligned to clock cycles, but needs to be varied in continuous time, again widening the fault space.
- While in synchronous circuits the clock alone determines the speed of operation, the behavior of QDI circuits is more intricate: A data source supplies data items (tokens) with a certain speed, and a sink consumes the results, again at its own pace. Due to the handshaking these speeds are not uncorrelated and so there is a variety of operating conditions for the circuit. It is already known that these are relevant for its susceptibility to faults [1].
- In contrast to synchronous fault injection studies we do not consider a single, fixed target circuit only. Asynchronous design offers a variety of templates and we want to compare these while also varying the function they implement, and also the protection scheme they employ. Again, this significantly blows up the parameter space.

In this paper we present the specific solutions we elaborated to make experiments with sound statistical results feasible under these challenging circumstances. Its structure is as follows: Section II briefly introduces the concepts of asynchronous circuits, to allow some basic understanding of the specific challenges this poses for the toolset design. A more general discussion of the envisioned experiment flow and the problems that need to be addressed is given in Section III. Next, Section IV presents the solution we propose for our toolset, with a particular emphasis on the specific techniques we

This research was partially supported by the project ENROL (grant I 3485-N31) of the Austrian Science Fund (FWF).

used to solve the mentioned challenges. Finally, some sample results from our comparative study on asynchronous circuit resilience are given in Section V to illustrate the capabilities of our toolset, before the paper is concluded in Section VI.

II. Background

Rather than using a rigid clock for synchronously coordinating the data transfer for all registers in a circuit, asynchronous circuits employ individual handshakes between the respective communication partners, which provides a very appealing flexibility to adapt the timing to the specific conditions [2].

These **handshakes** are constituted by transitions of the respective transmitting partner (source) and receiving partner (sink) that obey some specific protocol. The protocols used differ by the semantic interpretation of those transitions on the ack and req signals. The 4-phase protocol only uses the rising transitions to convey information, while the falling transitions are (generally) just viewed as a "return to zero" without semantic meaning.

While the ack is always a dedicated signal line, with relatively uncritical timing, the req signal is different. It is used to indicate that new data is available for the sink to consume and therefore we need timing assumptions. An alternative approach is to apply a suitable encoding to the data that allows a so called completion detector (CD) at the sink to recognize when a new valid token has arrived – by just looking at the encoded data. These types of protocols are referred to as delay-insensitive (DI). However for actual circuits using those protocols the slightly stricter QDI timing model is used [3].

There are numerous DI encodings, but the most popular one is a (4-phase) dual-rail encoding on the bit level: Each bit D is represented by two rails $(D.T, D.F)$, and a logic HI on D is expressed as (1,0), and a LO as (0,1). The (0,0) code is used to express that the bit does not carry a value, the code (1,1) is not used (and must not be reached during normal operation). Words with all bits at either (1,0) or (0,1) are called DATA tokens, while words with all bits at (0,0) are called NULL tokens (return to zero). Figure 1 illustrates how a pipeline stage based on 4-phase dual-rail protocol could look like.

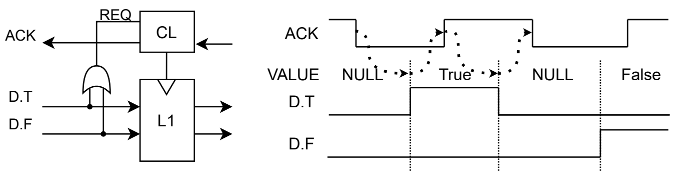

Fig. 1. QDI pipeline register with dual rail encoding and completion detector

The fault injection toolset presented in the following is intended for experiments on a variety of 4-phase QDI circuits and templates. Specifically we will focus on pipelined structures as they are very natural to use in the asynchronous context, and generally very popular.

III. Experiment Design and Requirements

A. Aims of the experiments

As outlined in Section II there are various QDI design styles, all of which come with their pros and cons regarding different aspects. Our focus is on resilience against transient faults. In this context numerous studies have been published, each focusing on a specific style, or a robustness improvement thereof, and each coming with its own assessment [1], [4]–[7]. One key aim of our research is to elaborate a like-for-like comparison of the different design styles plus their proposed improvements, with respect to robustness. Here the overall strategy will be to implement a given function in those design styles, with and without the proposed enhancements, subject them to the same set of faults while operating under the same conditions, and then compare the rate and types of erroneous behavior. Initially this will allow a direct and fair comparison of the robustness of the design styles, plus an assessment of the benefits obtained through the enhancements. In a next step the problematic cases can be identified and mitigation techniques systematically developed, or existing ones refined.

The challenge is to create a setup that is at the same time realistic enough to justify generalization of the results, simple enough to allow for a detailed analysis of problematic cases, fair in the sense of not introducing a bias in favor of any of the considered styles, and feasible with respect to the number of experiments and computational efforts required.

In addition we want to assess how certain parameters influence the robustness. This understanding is useful for a potential optimization of the operating conditions, but also for projections of results obtained for one setting to another. Examples of related research questions are:

- How does the speed of operation influence robustness?
- How does the balancing of the speed of input and output influence robustness?
- Is the control path more sensitive to faults than the data path?
- How does data width impact robustness?
- How does a timing skew among different data rails influence robustness?
- Where and how do faults get masked?

Finally, for the purpose of our further analyses, we do not only want to know, *whether* the circuit failed as a consequence of an injected fault, but also *how* it failed. To this end we define different failure categories, like deadlock, value fault (silent data corruption), coding fault (appearance of the forbidden (1,1) code word), timing fault (result arrives too late or too early), token faults (the number of tokens received at the output does not match the number of tokens put into the circuit). This distinction is important for obtaining a first impression of the root cause of the failure and hence devising appropriate countermeasures. Also, it allows considering that different types of failure may have consequences of different severity in a given application, like, e.g., a deadlock may be perfectly fine in a fail-stop application, while it is disastrous when a fail-operational capability is required.

B. Experiment Layout

Our experiments are based on fault injection into a simulation model of the target circuit during its operation. More specifically we use a pre-layout digital circuit model with annotated timing for gates in a Modelsim simulation. While this may be considered less realistic, and also lead to longer run times than injecting into a physical prototype, it allows for a comprehensive access to all internal signals for manipulation and observation. The flow of the simulation experiment involves the following steps:

Definition of the parameter space: In a first phase it needs to be decided which part of the parameter space shall be covered by the planned experiment. This not only includes the target and its configuration, but also the operating conditions (e.g., workload). Here often a confined space is selected in order to keep the focus on certain parameters or properties, as a partitioning of the experiment is anyway desirable to have better control in case of excessive run time, hang-up or data loss. We will present more details on the parameters of interest in the next subsection.

Golden run: Failures resulting from injected faults are defined as a deviation from the desired behavior. In order to have a reference for the latter, the target circuit is simulated performing the same operation run as later during the fault injection; this time, however, without any fault being injected. In this way, a simulation trace can be obtained and saved that constitutes a ground truth for the ideal behavior of the target – the so called *Golden run*.

Fault injection run: The operation performed during the Golden run is repeated, but now a randomly selected fault is injected and the reaction is observed. While this sounds simple, a lot of decisions need to be made that are crucial for the quality of the results: The target should be in a "steady" state; notably this is not the case right after reset, so some pre-injection time must be waited before allowing the first fault to be injected. Similarly, after injection of a fault, some post-injection time, must still be left to allow for an propagation of the fault. So the actual injection of faults must be confined to a restricted interval, the fault injection window. With respect to the fault location we consider every node of a given target circuit (on the abstraction level of primitive logic gates, i.e., not within gate-internal nodes) a potential victim. Therefore we need to make a lot of fault injection runs to cover all nodes over the defined time window. Estimating the needed number of injection runs is not a trivial task either.

Classification of behavior: By comparing the simulation trace from each single injection run with that from the Golden run we can, in principle, make a decision like "identical" or "deviation". However, not every deviation must be considered a failure; in a delay-insensitive circuit a slightly delayed but otherwise correct trace is perfectly fine. Furthermore, for the sake of later analysis, it is extremely helpful to have more information about the type of misbehavior. Therefore, after studying the results of some preliminary experiment runs, and also considering the related literature, we decided to distinguish several types of reaction to the fault injections (deadlock, value fault,...), as mentioned above already. Obviously, considering the huge amount of faults to be injected, this classification must be automated. Note that the failure types as we defined them are not mutually exclusive; one can, e.g., have an incorrect value followed by a deadlock. Avoiding a reduction to a single symptom, like prioritizing the first or the most severe one, allows for a more fine-grained diagnosis and an unbiased statistics about each individual symptom.

Presentation of statistics: In a final step, statistics about the occurrence rate of the different failure types are compiled, and correlation with different parameter settings is analyzed, in order to answer the research questions formulated above. Finding the right way of correlating data and presenting the relevant numbers in a way that supports the intuition and allows drawing the desired conclusions is a challenge of its own – but this is not within the focus of this paper.

Analysis of single cases: There are several reasons why a single fault injection along with its consequences may need to be analyzed in detail, i.e., the signal traces need to be studied. One such case may be finding an explanation for some unexpected type of behavior, to clarify whether it results from a flaw in the setup or interpretation, or points to an actually interesting insight into the circuit's properties.

C. Requirements on the tools

The aims and the consequent setup of the experiments imply a lot of challenges for the experiment toolset. While some of these are known from other fault injection studies, our envisioned study of various asynchronous design styles makes the problem even harder, as already outlined in Section I, especially through further blowing up the parameter space. Figure 2 gives an overview of the parameter space we consider.

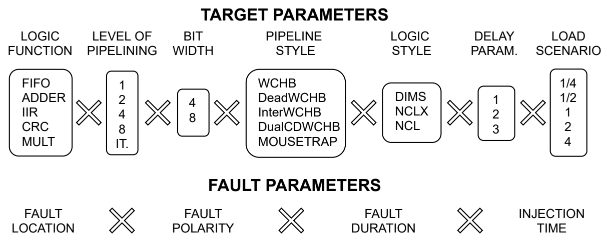

Fig. 2. Illustration of the considered parameter space

In more detail we have the following set of parameters:

- *Logic function:* In order to make our results as general as possible and useful for practical applications we have selected several types of function that are often found in a larger context: empty pipeline (FIFO), adder, infinite impulse response filter (IIR), cyclic redundancy check (CRC) generator, multiplier.
- *Level of pipelining:* A logic function can be realized in a single pipeline stage, or be distributed over several stages. Additionally it is often possible to implement a function through iteration (non-linear pipeline with a loop).

- *Bit width:* The above functions can be implemented with different data bit width.
- *Pipeline style:* There are techniques for robustness enhancement that we consider as own pipeline styles to simplify the presentation. Without being able to go into details about them here, we list the following styles as our target: WCHB (plain [8] / with dual completion detection [7] / with deadlocking buffers [1] / with interlocking buffers [1]), duplication and double checking [5], and Mousetrap [9]. For details, please see the original papers.
- *Logic implementation style:* There are also several options for implementing combinational logic in a QDI manner. Here we include DIMS [10], NCLX [11] and NCL [12].
- *Circuit delay parameters:* In asynchronous circuits propagation and interconnect delays directly impact the overall timing behavior – unlike synchronous circuits where those details are hidden behind the timing margins. In particular, it has already been observed in literature, that skew on the data buses has quite some influence on the sensitivity to transient faults. So, in order to obtain results with general validity these delay parameters need to be varied – which spans a huge space.
- *Load scenario:* The operation – and fault susceptibility – of a QDI circuit is not only determined by the supplied input values, but notably by the relative and absolute speed of its source and sink [1]. Consequently, variation of these parameters is crucial.
- *Fault location:* In our simulation approach, transient faults, in the shape of pulses that force the signal to a certain logic level, can be applied to any signal of the target circuit. Considering that signals are expected to have largely different sensitivity to faults, an appropriate choice of victim signals is crucial.
- *Other fault parameters:* The fault polarity (forcing to HI or to LO), its duration, as well as the time of injection, relative to the ongoing operation, are relevant for the effect. As already mentioned, asynchronous circuits must consider continuous time rather than discrete cycles, so for all time-related parameters a dense grid is required.

Note that for some of the above parameters the choice is discrete, which suggests a scan of all possible options, while others are continuous, where random choices (within a given range, possibly considering a given distribution) or a scan with a relatively dense grid seem natural. For the target synthesis we need flexible tooling to make it possible to automatically generate target variations just by changing the relevant design parameters on a high abstraction level. In summary, the parameter space is huge; in fact, covering all permutations of discrete parameters plus obtaining a decent density for the continuous parameters easily yields a requirement of billions of fault injection runs. This, in turn causes three major challenges:

Firstly, performing billions of fault injections requires high computational efforts. Therefore, availability of computing power, as well as overall run time of the experiment series need to be addressed in the design of our toolset.

Secondly, the data recorded throughout such an enormous number of experiment runs tends to be unmanageable. Therefore our toolset makes clever use of storage space and provides efficient data organization to allow for reasonably fast data post-processing and compilation of statistics.

And, finally, our first experiences with the experiments showed, that often in retrospective, during the collection of statistical parameters and their interpretation, one realizes that other aspects of the behavior, most notably signal traces, should have been recorded and stored as well. Therefore, reducing the recordings to the bare minimum is risky and a good balance must be found.

IV. Proposed Toolset

In this section we take a closer look at the toolset we already used to generate a total of over 2200 variations of our target circuits and coordinate a total of over one billion simulations. We will first give an overview and then go into details on how we solved the challenges formulated above.

A. Overview

As shown in Figure 3, everything is built around a central database.

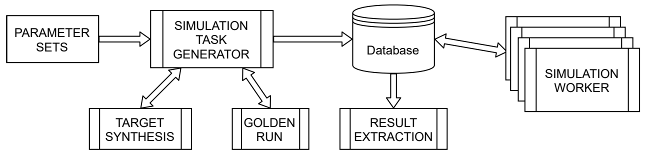

Fig. 3. Overview of the toolset

In general we can divide the toolset into four main parts:

- Target synthesis: Dependent on given parameters, a target circuit is synthesized and an appropriate test bench is generated.
- Task generation: Controlled by parameter sets, target variations are generated and added to the database as open tasks.
- Simulation execution: All open tasks are executed and the results are saved back into the database.
- Data extraction: A collection of routines is available to visualize data from the results.

To conveniently describe asynchronous circuits, we developed a simple language specification and an accompanying Python library and toolset. We based our circuit representation on production rule sets (PRS), a common gate-level specification approach for asynchronous circuits. The developed language and library allow us to efficiently design test circuits for our analyses, easily change parameters (level of pipelining, bit width), and even quickly switch between different QDI pipeline and logic implementation styles. The library is also able to generate VHDL code with timing annotation for simulation. The toolset also provides a parameterizable testbench generator. In summary, all files necessary for simulation are generated by our custom tools, and changing certain (circuit) parameters is straight forward. Due to space restrictions, we do not go into more detail about the target synthesis flow.

We inject only one fault per simulation run, and then reset the target before injecting a next one. This is more time consuming than allowing for an injection of multiple faults per run, but mandatory for maintaining a correlation between cause and effect. Without resetting the target in between two fault injections $F1$ and $F2$, one could not determine whether a failure observed after injecting $F2$ was caused by $F1$ or $F2$.

B. Speeding up the simulation

The basic strategies we apply for reducing the run time of the simulations are the following:

Full automation: In our planning of the toolset we took great care to avoid the need for manual intervention. Once the parameter space has been specified, the whole experiment is running to completion fully automated. This not only avoids the delays associated with user interaction, it also helps to reduce the probability of human error.

Target generation: The simulation task generator and the simulation workers are all controlled by a central target configuration file. We use a YAML file here, as it is human readable and can be easily generated and processed in Python. Figure 4 shows a simplified but representative example:

```
1   parameters:
2     CIRCUIT: PIPELINE
3     DATAWIDTH: 8
4     BUFFERSTYLE: WCHB
5     PULSDURATION: 10000
6   build:
7     tb:
8       .... test bench configuration .....
9     tb_prs: "pl.prs"
10    cmds:
11      — synthesize.py —w %DATAWIDTH% —s %BUFFERSTYLE%
12      — prscom.py ——siglist —o pl.siglist ——vhdl pl.prs
13      — prscom.py —e —o pl.vhd pl.prs —m
14  vcom_args: "—2008 —work work —suppress 1236 —novopt"
15  vsim_args: "—msgmode both"
16  simulation_time: —all
17  time_resolution: 1ps
18  compile_files:
19    — pl.vhd
20    — tb.vhd
21  fault_injection:
22    victim: {mode: random, file: pl.siglist}
23    injection_value: {mode: random}
24    injection_time: {mode: random, range: [178473, 330910]}
25    injection_duration: {mode: fixed, value: "%PULSDURATION%"}
26    iterations: 200000
```

Fig. 4. Example configuration file

On line 1 the key *parameters* contains a dictionary which includes all important parameters of a circuit. The idea behind this is to have a parameter set which uniquely identifies a circuit variation. This parameter set is also saved in a separate table in the database to be accessible over SQL queries. Additionally, to make it easier to modify the configuration quickly, all the parameters are provided as variables in the configuration file. To access a parameter value it is just necessary to surround its name with %.

The synthesis instructions are defined in lines 6...13. On line 8 the configuration for generating the testbench is added. From line 14...20 we define all necessary options for Modelsim to compile and simulate our target circuit. All information about our transient fault injections is specified on line 21...26.

To make the system more flexible we can use lists for our parameters in brackets, like *DATAWIDTH: [4,8]*. In general the different possible options for each parameter will be permuted. Therefore is it possible to describe a lot of different target variations with a single target configuration file.

Parallelization: The simulation runs for different circuit parameters are completely independent of each other, so it is easy to dispatch them to different computers (more generally, we call them "simulation workers") and just collect the individual results in the database. Our approach is highly flexible: A simulation worker just needs to have Python and Modelsim installed, with Linux or Windows as OS, and be connected to the database. Once started, it is looking periodically for open tasks in the database and starts processing them if there are any (for details on how these tasks are prepared, see below). For each available processor core a Modelsim instance is started.

Avoiding overheads: In preliminary experiments it turned out that restarting Modelsim takes up a lot of resources therefore running multiple simulations in one instance can speed up the simulation process. The actual number is mainly dependent on the complexity of the simulation itself; we empirically determined that for our targets around 100 fault injection simulation runs per instance are most suitable. If something went wrong the simulation worker stops, which makes it possible to retrace the problem and fix it. This provides some safety against potential corruption of the database, at the cost of missed computation time.

C. Managing the parameter space

We pursue the following strategies to confine the parameter space without losing relevant information:

Parameter pre-selection: We performed preliminary experiments to explore the influence of fault duration on the results. As shown in Figure 5 it turned out that a fault duration

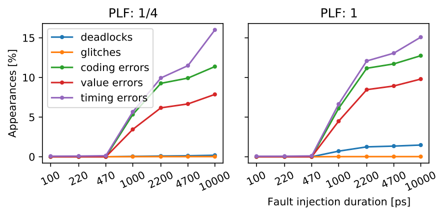

Fig. 5. Fault injection duration experiment results for the multiplier

below a certain threshold (dependent on the circuit delays) is masked by the inertial delay model we chose to use for our implementations. This bounds the useful fault duration from below. Similarly, fault duration beyond a certain threshold does not yield any changes compared to shorter durations. This is because QDI circuits simply stop operating if transitions that are due to occur are blocked. This forms an upper limit for a useful fault duration[1]. Based on these findings we decided to use a single fault duration of 1.2 ns only – which eliminated one dimension in the parameter space.

[1]It should be noted here that for other target circuits the upper limit was not always as clearly visible.

Auto-tuning of parameters: According to the requirements formulated in Section III we do not want to start the fault injection before the target is running in steady-state operation after reset. Neither do we want to perform injections when no further tokens are arriving anymore, as we want to allow the fault effects to propagate. These conditions define lower and upper boundary of the *fault injection window*. Additionally the length of the injection window needs to cover relevant activities in all interesting stages of the handshaking of all components in the circuit.

Obviously, we cannot afford to choose these bounds too far on the safe side, as increased simulation time will consume costly computation resources. In order to approach the effective borders as close as possible, we performed exploration experiments, based on whose outcome we empirically decided to start the fault injection not before the first two tokens have reached the output. For the injection duration we again wait for two tokens at the output. This proved to give a good coverage of all phases of pipeline operation. Experimentally determining the end time of the simulation is more difficult, as the waiting time depends on the number of tokens a circuit can contain. To explore that, we run a simulation with extremely long sink delay, and count the number of input tokens consumed by the circuit until the first one reaches the output. On top of this number we add one more token, and this defines the minimum number of tokens that must be processed at the output to let possible faults propagate to the output. In this manner the injection window and simulation time are automatically tuned to the respective circuit and load scenario. Figure 6 illustrates this method.

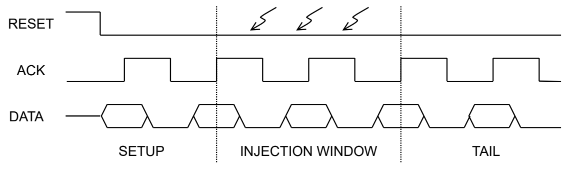

Fig. 6. Illustration of the fault injection window

Similarly, we have implemented an auto-set capability for the load scenario. Its principle is as follows: In Figure 7 the different phases of the handshake protocol are shown. In case the sink has a long delay for generating the acknowledgment signal to a data token it received, the next upstream buffer cannot hand over the next token it may already hold, and this backlog continues up to the source. Ultimately, the pipeline will be filled with tokens and its speed will be determined by the slow sink. This is called the *bubble limited* mode of operation. In contrast, if the source is slow, then the previous token it provided will have propagated through the whole pipeline and been consumed by the sink before the source issues the next one. In this *token limited* mode of operation the pipeline is empty most of the time, and its speed dictated by the source. Obviously, there is a whole range of possible pipeline fill levels in between these two extremes. To quantify that, we introduce the pipeline load factor (PLF):

$$PLF = \frac{\text{waiting time for acknowledgment}}{\text{waiting time for token}}$$

If the PLF is greater than 1 we are mainly waiting for the acknowledgment signal to arrive and therefore the pipeline is in a bubble limited mode. And if the PLF is smaller than 1 we are in a token limited mode.

To measure the PLF we monitor the different handshaking phases of every buffer's input channel which is not directly connected to the primary input. To make the measurement more accurate we average over a run with 100 tokens. For a linear pipeline we can fully control the PLF by setting the sink and source timing of our testbench. Small and big values for the PLF will increase the simulation time significantly, and therefore also the injection window and space, which is costly. Hence for our experiments we use the following PLFs: $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4. The PLF will be set by repeatedly running Golden simulations and searching for the needed sink and source delay through bisection.

Note that the above discussion applies to a linear pipeline only. For the target variant with iterative computation, the PLF is largely determined by the loop roundtrip time and several other factors. Knowing that we have little control over the PLF through source and sink delays here, we do not attempt to attain a target PLF, but rather measure the PLF as is.

Automated (exhaustive) scanning of discrete parameters: For many parameters we have a discrete list of possible choices. This list can be specified by parameter sets (recall Fig. 4). At the start of the experiment, the respective simulation tasks are generated and saved in the database, where they are then picked by the simulation workers. The parameter pre-selection and the auto-tuning approaches explained above ultimately also generate such lists and hence those parameters are finally included in the same way.

Another somewhat discrete parameter is formed by the actual gate delays. First, circuits are always generated using fixed delays for each of the used gate types. To check the circuit under PVT variation we then vary those delays randomly by up to 10% for each individual gate. For our experiments we consider 3 circuit timing scenarios (see Figure 2).

Randomization plus coverage verification: Randomization is used for the following parameters: (i) victim signal, (ii) fault polarity, and (iii) start time of the injection (within the injection window). Recall that time needs to be considered continuous in context with QDI circuits. To get statistically valid and meaningful results, we need to collect a sufficient number of random samples in the space we want to cover. However, the size of this space depends on other parameter settings (most notably the size of the fault injection window),

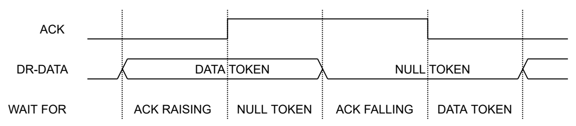

Fig. 7. The four phases of a dual rail 4-phase handshake protocol

so we need to adapt our requirements dynamically. This, again, calls for auto-tuning of the sample size. In view of future extension that we may want to add (e.g., a random injection duration), it seems appropriate to avoid relying on any details of the random parameter generation mechanism.

Therefore, we simply run the parameter generation process (as a black box) multiple times and count the number of generated identical parameter sets ("collisions"). While an exact derivation is a tough combinatorial problem, we have empirically determined that the coverage has a roughly linear relation to the proportion of collisions (relative to the generated set) observed. Specifically we have found that for a proportion of 0.1% of collisions we obtain a good coverage of our randomly selected space. This was verified by plots that graphically illustrate the coverage over time and space (signal).

D. Managing the data

Our experiments produce a huge amount of data that must then be scanned and evaluated to derive the trends and conclusions, as desired. The problem of balancing the selection of data to be stored in a way to keep the database size reasonable while still enabling the detailed reproduction of every single simulation, has already been mentioned. Our solution encompasses the following methods:

- For the observed target behavior we store only simulations that deviate from the golden run, which turned out to allow considerable savings of space.
- For the generation of the random injection parameters we use pseudo-random generation that is initialized with a specific seed. Storing this seed in the database enables us to reproduce every "random" experiment in detail.
- We do not store the complete timing waveforms but only observed symptoms. If needed, the information stored can be used to rerun any simulation.

E. Result Extraction

For analyzing the resulting database entries and extracting insightful figures and statistics representation we mainly use Jupyter notebooks, which provides an interactive execution environment for Python and other languages. This makes it an ideal solution for interacting with the database and visualizing results. Additionally the internal memory management makes it possible to temporarily cache results and therefore save time waiting for the database to respond.

V. CASE STUDY

To illustrate the capabilities of our toolset and give evidence for its suitability we present some sample results here that we obtained by it. The results presented in this section are taken from our first big scale experiment with over one billion executed fault injection simulations. Figure 8 shows the progress of the executed simulation on around 40 computers over a time span of roughly one moth. There where some initial problems, as can be observed in the graph, but in the end everything was running reliable.

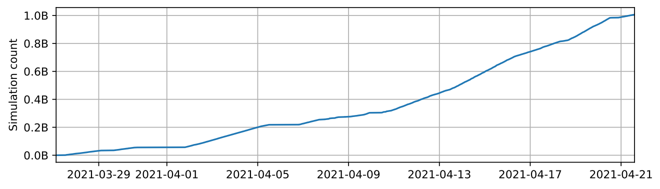

Fig. 8. Simulation progress for over 1 billion simulations

Figure 9 shows the results we obtained for a pipelined 8-bit multiplier circuit, implemented with six different buffer types, showing one bar per observed effect class. The bars going upward refer to the data path, while the downwards growing bars refer the control signals. The scale on the y axis shows the portion of simulation runs that lead to the particular outcome. The right-most implementation style (DD WCHB) is completely tolerant to single faults and thus shows now effects (except for some timing deviations, which are however fine in the case of QDI circuits). A detailed discussion would go beyond the scope of the paper and the space limitations, but it is clearly visible how the buffer types differ.

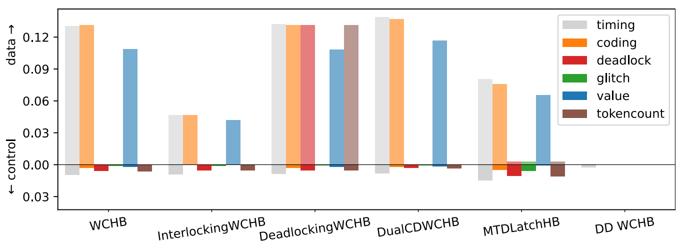

Fig. 9. Example result showing the sensitivity of the different buffer types

The impact of the pipeline load factor can be observed in Figure 10 where the occurrence of the different behavior classes is drawn over the PLF. It is clearly visible that the pipeline load factor has a strong influence on the resilience, but there is no clear trend observable. Therefore further experiments need to be conducted to get a better understanding.

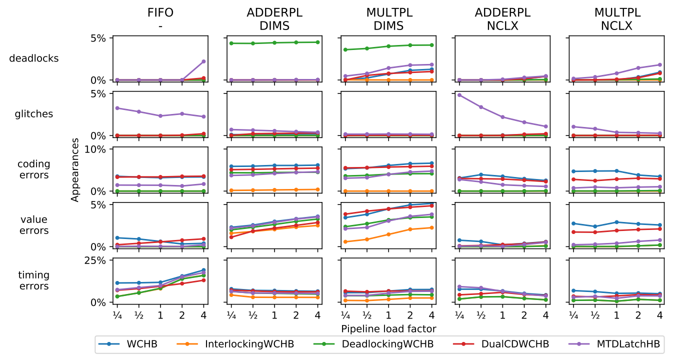

Fig. 10. Example result showing the impact of the load factor

It is also possible to look at the main source causing a failure type. In Figure 11 we show a heatmap of the different circuits and mitigation methods. If the color is mainly yellow the main source of the failure is caused by injections into data signals. For blue regions it is mainly caused by control signals. In general one row includes all information presented in Figure 9, but without the rate of occurrence. This can help to better understand the weak points of a circuit and aid in optimizing.

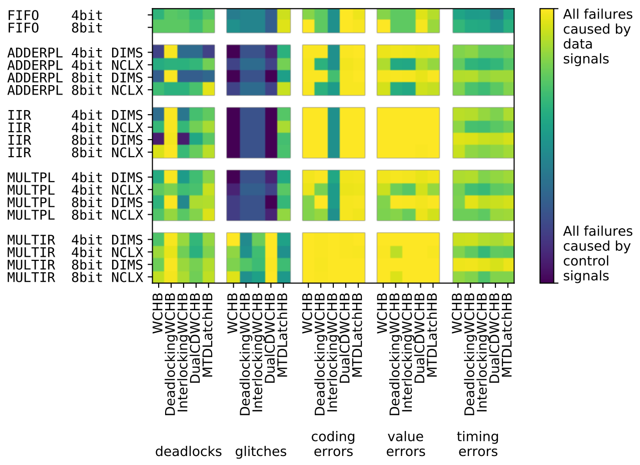

Fig. 11. Example result showing the main source of the different failure types

VI. Conclusion

Attaining reasonable coverage of the parameter space is a notorious problem encountered in fault injection studies. The facts the asynchronous nature of our targets prohibits us from restricting ourselves to a discrete (clocked) time representation, and that there are many different styles for implementing buffers and combinational logic substantially aggravate this problem. We have presented a toolset that addresses this issue through several techniques.

Among these are well-controlled randomization of parameters, parallelization of the simulation tasks, and careful avoidance of overheads. We specifically address adaptive performance of asynchronous circuits through a novel, automated method for tuning the pipeline load factor, and through targeted pre-selection of the fault duration via preliminary analyses we effectively eliminate one continuous-time parameter. Our solution to the big variety of implementation templates of asynchronous designs is full automation of the experiments including generation of target circuits and testbenches.

Another key ingredient of our solution is the use of a central database to configure and dispatch the individual simulation tasks, collect the results and perform evaluations, which is key to keeping an overview of the huge amount of simulation data.

In this context we have also presented approaches for reducing the sheer amount of data, while still allowing a reproduction of every single experiment in all detail – in spite of parameter randomization.

As an illustrating example we have used our envisioned experiments on a variety of asynchronous target circuits. This is a hard problem, as, on top of the challenges commonly encountered in fault injection experiments, our planned survey and comparison entails that we have many different target circuits plus enhancement techniques, which blows up the parameter space further.

We have shown some sample results that give evidence for the suitability of our toolset and illustrate the type of insights one can obtain by using it.

Our future work will be dedicated to adding support for more detailed analysis (like recording key indicators for the status of the internal pipeline stages), decomposing the time dimension into protocol phases (within which the exact time of fault injection may be less relevant for the outcome), and extending our synthesis tool to also handle bundled data circuits and synchronous ones.

References

[1] F. Huemer, R. Najvirt, and A. Steininger, “Identification and confinement of fault sensitivity windows in qdi logic,” in *2020 Austrochip Workshop on Microelectronics (Austrochip)*, Oct 2020, pp. 29–36.

[2] J. Sparsø, *Introduction to Asynchronous Circuit Design.* DTU Compute, Technical University of Denmark, 2020, paperback edition available here: https://www.amazon.com/dp/B08BF2PFLN.

[3] R. Manohar and Y. Moses, “Analyzing isochronic forks with potential causality,” in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, 2015, pp. 69–76.

[4] Y. Monnet, M. Renaudin, and R. Leveugle, “Asynchronous circuits sensitivity to fault injection,” in *10th IEEE International On-Line Testing Symposium*, July 2004, pp. 121–126.

[5] W. Jang and A. J. Martin, “SEU-tolerant QDI circuits [quasi delay-insensitive asynchronous circuits],” in *11th IEEE International Symposium on Asynchronous Circuits and Systems*, March 2005, pp. 156–165.

[6] S. Peng and R. Manohar, “Efficient failure detection in pipelined asynchronous circuits,” in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT’05)*, 2005, pp. 484–493.

[7] W. J. Bainbridge and S. J. Salisbury, “Glitch Sensitivity and Defense of Quasi Delay-Insensitive Network-on-Chip Links,” in *15th IEEE Symp. on Asynchronous Circuits and Systems*, May 2009, pp. 35–44.

[8] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer’s Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[9] M. Singh and S. M. Nowick, “MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[10] J. Sparsø and J. Staunstrup, “Delay-insensitive multi-ring structures,” *Integration, the VLSI Journal*, vol. 15, no. 3, pp. 313 – 340, 1993, special Issue on asynchronous systems.

[11] A. Kondratyev and K. Lwin, “Design of Asynchronous Circuits by Synchronous CAD Tools,” in *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, June 2002, pp. 411–414.

[12] K. M. Fant and S. A. Brandt, “NULL Convention LogicTM: a complete and consistent logic for asynchronous digital circuit synthesis,” in *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, Aug 1996, pp. 261–273.