Project Number: FP7-611404

D5.4.2 - Early Validation Report

Authors[1]

M. Pipponzi (YOGITECH), F. Sforza (YOGITECH), S. Di Carlo (POLITO), A. Savino (POLITO), M. Iacaruso (YOGITECH)

Version 1.0 – 27/04/2015

Lead contractor: YOGITECH
Contact person: Mauro Pipponzi YOGITECH Spa Via Lenin 132/P 56017 S.Martino Ulmiano (PI) - Italy E-mail: mauro.pipponzi@YOGITECH.com
Involved Partners[2]**:** YOGITECH, POLITO, UoA, CNRS, UPC
Work package: WP5
Affected tasks: T5.5

Nature of deliverable[3]	R	P	D	O
Dissemination level[4]	PU	PP	RE	CO

[1] *Authors listed here only identify persons that contributed to the writing of the document.*

[2] *List of partners that contributed to the activities described in this deliverable.*

[3] *R: Report, P: Prototype, D: Demonstrator, O: Other*

COPYRIGHT

[4] ***PU**: public, **PP:** Restricted to other programme participants (including the commission services), **RE** Restricted to a group specified by the consortium (including the Commission services), **CO** Confidential, only for members of the consortium (Including the Commission services)*

INDEX

Scope of the document

This document is the main outcome of task T5.5 **"Preliminary Validation of Developed Models"**, elaborated in the Description of Work (DoW) of the CLERECO project under Work Package 5 (WP5).

Figure 1 depicts graphically the goal of this deliverable, its main results, the inputs it uses and the outputs it provides (including which WPs will use its outputs).

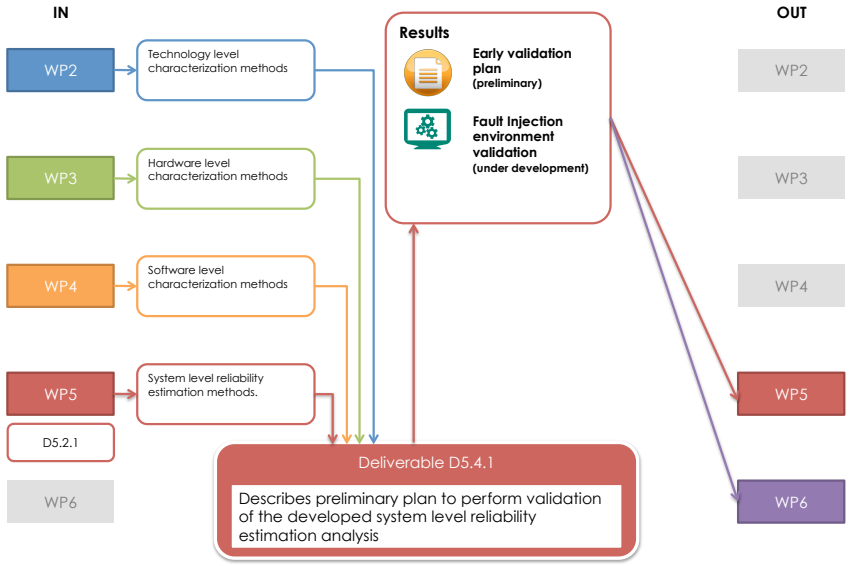


Figure 1: inputs and outputs of this deliverable

This document describes the preliminary validation environment envisioned for the validation of the models and algorithms developed in CLERECO for system level reliability evaluation. It takes its inputs from the results of several work packages and it is deeply connected with the activities and results of all WPs.

The document is organized in the following sections:

- **Description of the problem**. This section introduces the main challenges to face in the implementation of the validation plan
- **Set Up of the Validation Platform**. This section describes the fault injection environment used for validation.
- **Injection campaign**. This section overviews problems and solutions in the setup of the injection campaign
- **Validation workflow**. This section describes the technical steps implemented to perform the validation.
- **Preliminary experimental results**. This section provides preliminary validation results on a set of benchmarks.
- **Next steps**. This section concludes the document and highlights the next steps to implement until the end of the project.

This document is an updated version of Deliverable D5.4.1. The main update w.r.t. the previous document are:

- Complete implementation of the validation workflow that is described in section 7.
- Preliminary experimental results from the validation.

This is a draft deliverable. A final version submitted during M36 will replace this document.

1. Introduction

Validating system level reliability estimations obtained resorting to the high-level models developed in CLERECO requires to compare obtained results with well established industrial practices to perform reliability evaluation.

This document details the features of the validation environment set up for the CLERECO project. It describes the adopted solutions, the fault injection environment and the operations carried out to validate the developed models, algorithms and metrics.

2. Description of the problem

The different models and algorithms composing the CLERECO system level reliability evaluation method need a low level validation step, that enables to guarantee that the metrics computed with an high-level analysis are confirmed when the actual hardware and software structure are at play.

Figure 2 shortly recalls the Bayesian Network model exploited in CLERECO to perform system level reliability analysis described in deliverable **D5.2.2 – System Reliability Estimation Models.** The CLERECO system level reliability model describes the system in terms of a Bayesian network whose nodes represent system technologies, hardware components and software blocks. Interactions among nodes represent capability of the system to propagate errors from one component to another component. The reliability evaluation method, estimates the rate of failure at the application level by propagating failure rates across nodes describing the system (Figure 2) using a set of conditional probabilities P_{fail} that model the probability that a failure at a certain level of the architecture hierarchy is propagated to the higher level. The final figure is the failure rate at the application level. The conditional probabilities P_{fail} characterizing the models are the results of a set of high-level fault injection campaigns carried out – with different tools - at the micro architectural and at the software levels.

It is the task of the validation to confirm the estimated failure rate at the application level by applying state-of-the-art injection of faults at a lower level (i.e., RTL level).

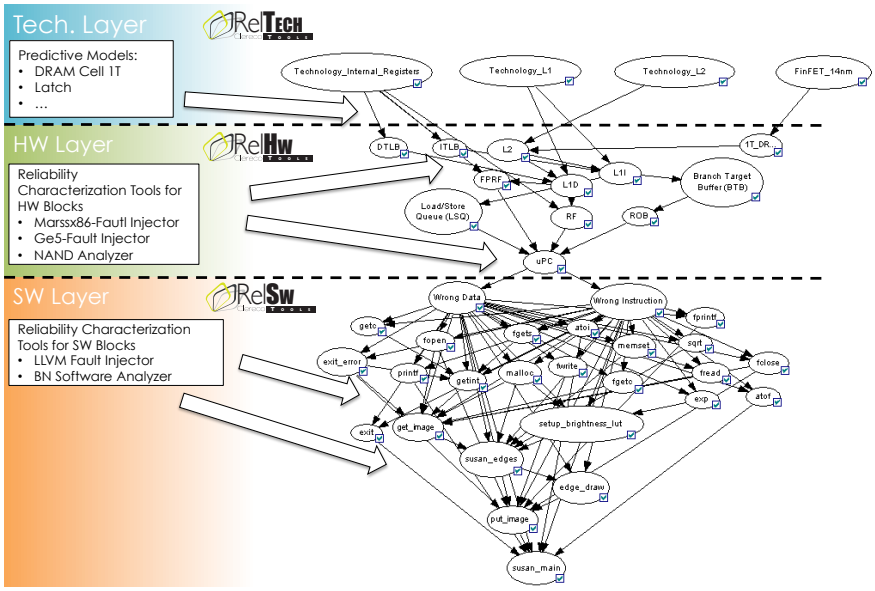


Figure 2: Reliability evaluation model.

2.1. Complexity of the injection campaign

The problem has a nontrivial level of complexity because of the performances of a low level simulation (RTL and below). A complex system can require injecting a number of faults in the order of the hundreds of millions, and the performances of the injection are highly dependent on the performances of the simulation, that can vary quite dramatically from one design to another. In our experience, an injection campaign can vary from tens of thousands of faults injected per hour to a few faults injected per hour (see also Section 6.2).

The parameters determining these numbers are many:

- The workload used for the fault simulation (which in turn reflects the application).
- The size of the design.
- The performances of the simulator.
- The number of simulators that can be run in parallel.
- The structure of the simulation environment.
- The possibility to adopt techniques that make it possible to speed-up the injection process (e.g., avoiding to run the full simulation for each fault).

In general, it is a safe assumption to claim that an injection campaign can require months of simulation even on specifically designed system.

It is therefore necessary, in order to be able to bring the problem within an acceptable number of simulation hours, to adopt an approach that, without sacrificing the accuracy of the analysis permits to reduce the simulation time as much as possible.

3. The Adopted Solution

During the project two main courses of actions have been analyzed and discussed with the partners to establish a validation protocol for the project results:

- The first option involves a divide-and-conquers approach: partitioning the systems into blocks, performing fault-injection based analysis of the different blocks and then combining them developing an ad-hoc methodology for the composition of the metrics. This approach is already in the roadmap of YOGITECH's mainstream development.
- The second approach is based on RTL level fault-injection on the full system using different techniques to reduce the fault injection time by dropping simulations for which the effect of the fault can be somehow predicted.

The discussion within the consortium has considered both approaches as interesting. However, the first approach requires developing methods to compose partial low-level fault injection results, mimicking at a lower level what the CLERECO reliability estimation model performs at system level. While this is still an interesting approach, it is not well consolidated at industrial level and may bias the robustness of the validation phase. Differently, the second approach represents a consolidated method. It must also be stressed the fact that the validation would be more significant with an approach that, even if at a different level of abstraction, does not repeat steps already used by the solution to be validated.

As a conclusion, we decided to adopt, at least for demonstrators in the embedded domain that will be based on ARM architecture and for the preliminary use cases, an RTL injection validation campaign.

4. Phases of the Validation Campaign

The validation campaign is carried out through a fault injection on the complete hardware in order to validate the failure rate at the application level as estimated by the high-level reliability estimation method.

Figure 3 shows the validation workflow.

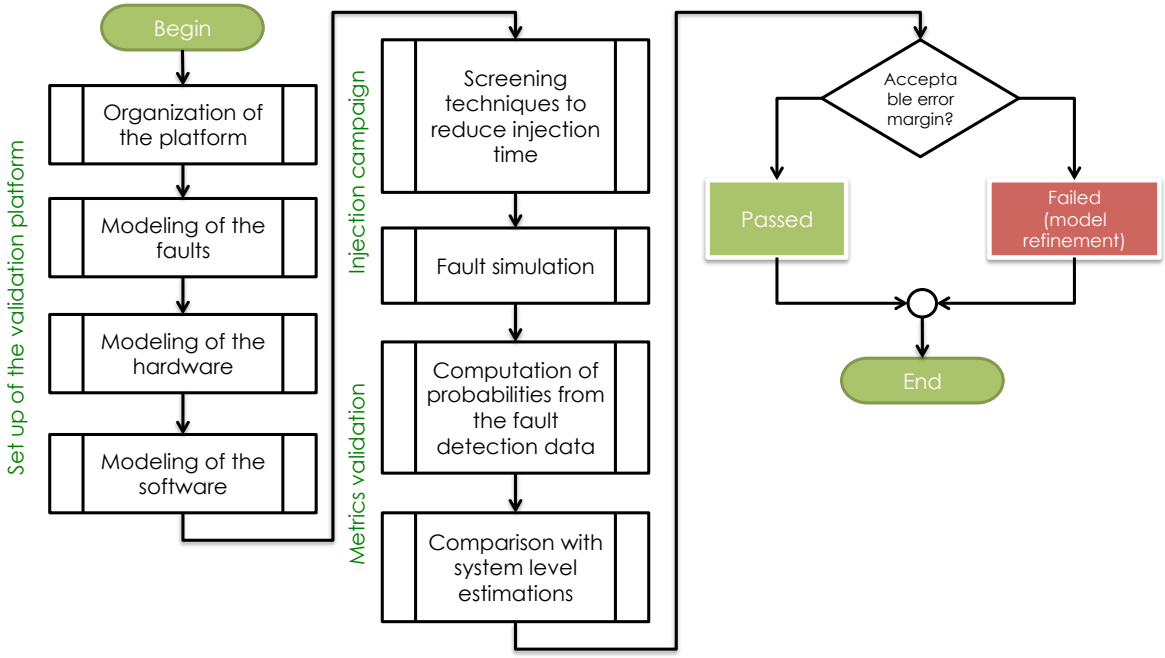


Figure 3: Validation workflow

The validation campaign can be partitioned into three main phases:

1. **Set up of the validation platform:** this phase requires studying the target system to validate in order to identify the models of the different components, and to define the target fault models.
2. **Injection Campaign:** in this phase the actual validation is performed by means of a low-level fault injection campaign. Due to the complexity of this task screening techniques to reduce the injection time must be implemented.
3. **Validation of the metrics:** in this final phase, application level failure rate are computed starting from the fault detection data obtained with the fault injection campaign. These numbers are compared with equivalent numbers obtained through CLERECO early reliability estimation models in order to assess the accuracy of the high-level models. In case accuracy is not enough, further refinements of the models will be implemented and a new validation phase will be required.

5. Set Up of the Validation Platform

The validation approach used is summarized in Figure 4. From it exploits the YOGITECH commercial tool chain that is nowadays a state-of-the-art suite for the safety analysis of com-

plex systems and in particular it exploits two main tools: the Safety Designer and the Safety Verifier.

The Safety Designer is used for two purposes: to focus the fault injection activity by selecting elementary parts and applying sampling factors according to the device under test and to provide a conservative estimation in case fault injection is not feasible (common situation for complex SoC and/or very complex applications).

The Safety Verifier is used for the fault injection campaigns. As showed also in Figure 4, the Safety Designer provides some inputs to the Safety Verifier, in order to reduce the fault injection complexity.

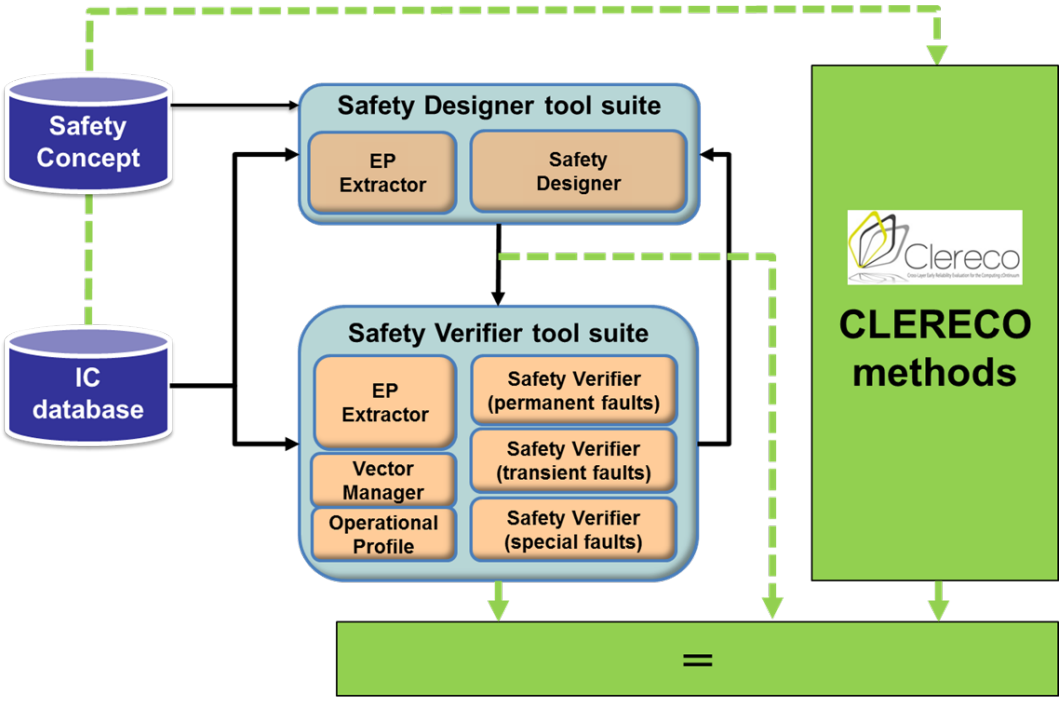


Figure 4: Overall view of validation

The validation goal is, first of all, to show that CLERECO accuracy is close to fault injection results in order to drastically reduce or remove the need of fault injection and to show that CLERECO accuracy is less conservative than high-level safeness analysis. This concept is summarized in the Figure 5 where we show, as the CLERECO system reliability analysis can be closer to the estimation results or to the fault injection results.

Figure 5: Validation goal

5.1. Organization of the Platform

It has been decided by the consortium that the preliminary validation plan is based on RTL injection on ARM based microprocessor, at least for the embedded demonstrators and for the preliminary use cases. In particular, an ARM Cortex A9 based platform has been chosen.

The choice of the platform and of the strategy, imply a number of requirements, such as:

- RTL database of the full HW layer.
- Transient fault model (bit flip of the output of memory elements / memory cells).
- Workloads (tests) at different level of abstraction modeling the low level functions of the software up to the target application.
- A test environment built in such a way to allow using the target simulator's features for saving and restoring simulation snapshots.

5.2. Modeling of the faults

The CLERECO system level reliability estimation methodologies enables to consider several types of fault model. In this preliminary validation plan we focus on analysis of the effect of transient fault that represent a significant class of faults for reliable systems. In particular we model faults as Single Event Upset (SEU), where the value of a bit is flipped (0->1 or 1->0) at a certain point of the simulation (see also Figure 7).

The fault is characterized by:

- **Location**: where the fault takes place in the mode of the HW.
- **Time**: when the fault takes place during the simulation.

Typically, transient faults are injected at the output of memory elements, therefore, in a RTL model they are injected on:

- Clocked signals.
- Output of memories.
- If the model of the memory makes the bit accessible, individual bits within a memory block.

5.3. Modeling of the HW

The HW is being described as a hierarchical RTL model of the platform as sketched in Figure 6. The hierarchical organization of the RTL model – even if not mandatory – should match as much as possible the high level decomposition of the hardware functions. This would allow additional benefits, such as speed up the fault injection campaigns (for further information, see Section 2).

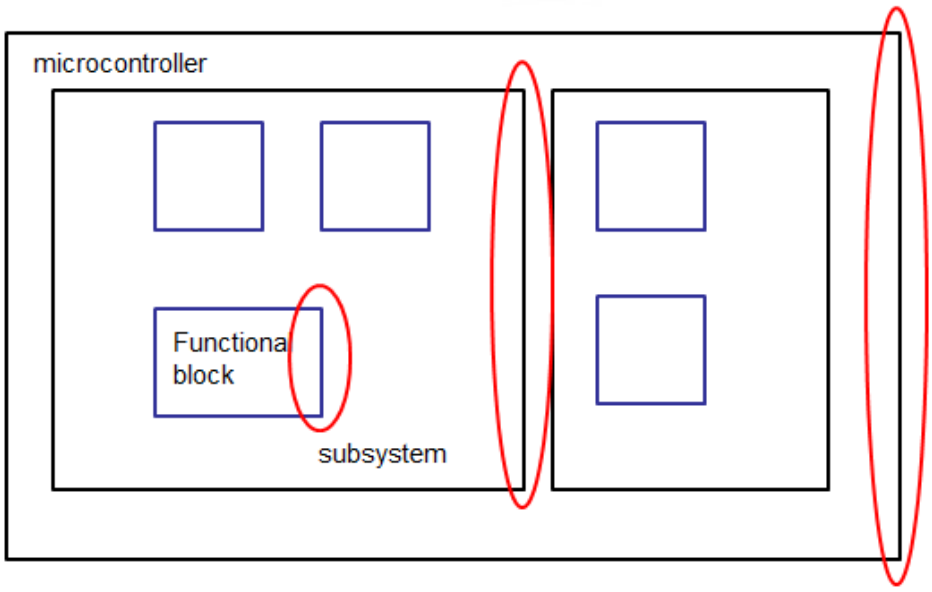


Figure 6: Hardware layers

The list of the subset of outputs of the functional blocks significant with respect to the detection of the fault (so called *observation points*) is part of the model of the functional blocks.

5.4. Modeling of the SW

The software functions, from the most basic to the full application, are modeled as workloads (test benches) for the hardware. These workloads can be either modeled with a HDL or can be modeled as vectors to be applied at the input of the HW.

The vectors to be applied to the hardware are captured by *strobing* the application running on a physical platform, saving them and then formatting them as suitable inputs for the simulation software.

As for HW, also in the case of the SW, part of the model of each function is the set of signals (registers, contents of memories, etc.) that are significant with respect to the detection of the injected faults.

6. Injection Campaign

6.1. Fault Simulation

The injection campaign injects transient faults (identified by the where/when pairs) into the HW, in order to determine whether these faults change or not the behavior of the device.

The faults are modeled as Single event upset (SEU), i.e., bit-flips on the output of memory elements (flip-flop, latches, memories). Figure 7 shows how bit-flips can be emulated at the simulation level (the example relies on HDL language formalisms).

```
flip (s)
input s;

begin
  if (s==1'b0)
    flip = 1'b1;
  elsif (s==1'b1)
    flip = 1'b0;
  else
    flip ? 1'bX;
  endif
end
endfunction
```

Figure 7: bit-flip model

The detection takes place at the output of the hierarchical blocks and functions, i.e., a fault is detected if it changes the behavior of the device at the selected sets of observation points (see Figure 8). In our case, the observation points are those registers, memory locations and outputs that are significant with respect to the data produced by the application.

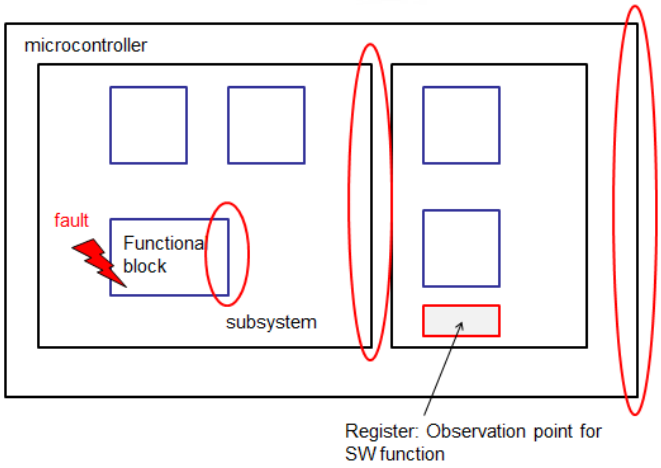


Figure 8: Observation Points

The actual injection platform (see Figure 9) is organized as a sequence of functional simulations, one for each fault injected, where a faulty machine is instantiated together as a golden machine and the two simulation results are compared with respect to the observation points.

Each fault is simulated and classified after the result of the simulation, whether detected or undetected.

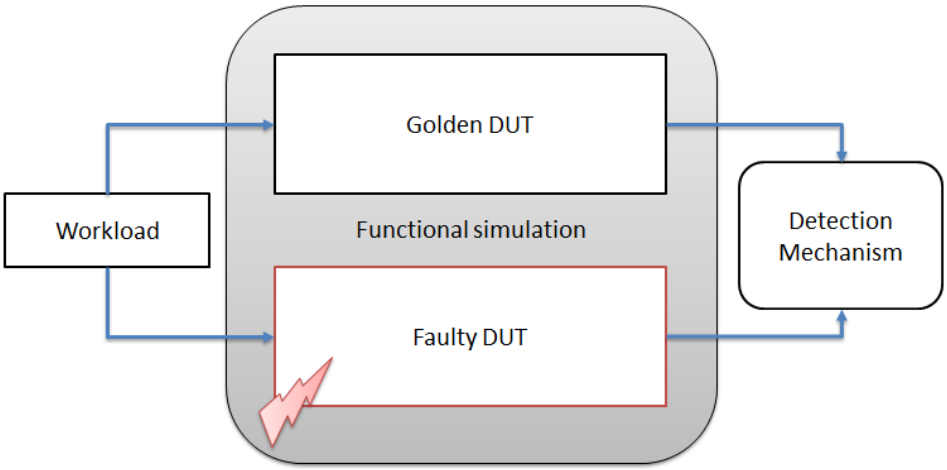


Figure 9: transient injection setup

6.2. About the complexity

As mentioned already, a fault injection campaign is a challenging task, because the number of faults to be injected in a complex system can rise rapidly to a number in the billions.

There are a number of factors affecting the length of a fault injection:

- The number of the faults injected.
- The length of the simulations.
- The frequency of the clock.
- The possibility to parallelize the simulations on many CPUs.
- The way the test bench is built.

Typically, in our experience, completing an injection campaign on an ARM based microcontroller requires 2-3 months of simulations, with up to 100 parallel simulations running in a computing farm.

We can report here a few quantitative figures, from actual projects YOGITECH has been working on.

Table 1: Quantitative examples of time required and parallelization level for different designs sizes.

	Size (Mgates)	Overall time to complete the campaign	Parallel CPUs
Example 1	5	2 months	10
Example 2	15	3 months	15
Example 3	30	3 months	100

For this reason, the fault injection setup must be done carefully.

The information required to properly set-up the experiment are:

1. Elementary Parts and Flip Flops to be injected (where to inject faults)
2. Observation Points (in order to distinguish safe and dangerous faults)
3. Window Of Opportunity (when to inject faults)
4. Fault distribution
5. Estimation of the number of faults to inject

Inputs 1 and 2 are related to the hardware, while input 3 is related to the workload. The fault distribution (input 4) is Uniform. The number of transient faults to be injected (input 5) is based on the following formula [2]:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}}$$

Figure 10: Number of transient faults injected

Where:

- t is derived from the confidence level = 90%
- p = 0,5
- N is calculated based on:
  - number of elementary parts (EP) to be injected
  - duration of the WOO (Window Of Opportunity)
- e is calculated based on the overall acceptable margin to reach the claimed metric = 1%

Using the typical parameters (e = 1% and confidence level = 90% or 99%) the number of faults can vary from 4106 to 13530.

The following section presents implemented strategies to speed-up the fault injection campaign.

6.3. Improvement of the fault simulation performances

The previous considerations make it very difficult to be able to run an exhaustive injection campaign in an acceptable time problematic, not only from CLERECO project perspectives.

There are however a number of areas where it is possible to intervene to decrease dramatically the number of injected faults, and, therefore, the number and the length of the simulation. A set of approaches has been already considered to deal with it.

6.3.1. Saving the vectors produced by the Golden Simulation

The conceptual model of running the golden and faulty simulation for each fault can be improved by:

- Running the golden simulation once,
- Saving the observation signals as vector and,
- Using these vectors for the comparison with the simulation of the faulty machine.

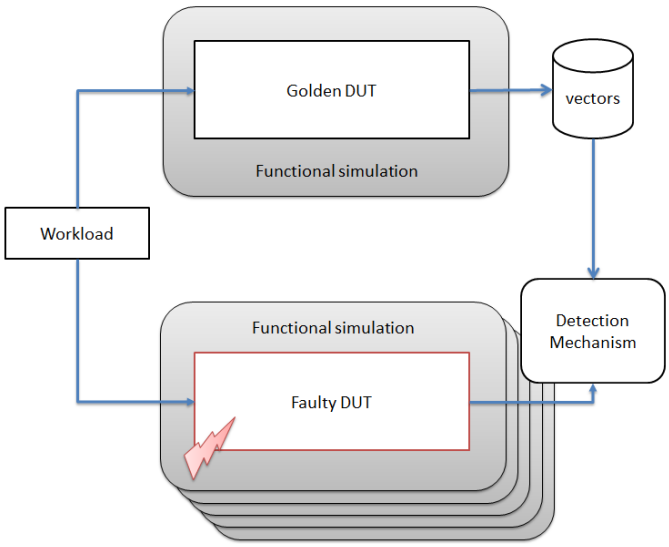


Figure 11: running the golden machine just once

The setup is shown in Figure 11 and allows decreasing the load on the computing machine by practically halving the number of simulations.

6.3.2. Excluding the faults in the non-functionally relevant portions of the design

A second area of improvement requires an architectural knowledge of the platform being simulated, and involves removing from the list of the faults to be injected all faults that are injected on portion of the hardware that is not functionally significant (e.g., test logic) or is not affected by the workload being simulated.

6.3.3. Saving the snapshot of the simulation

As mentioned in Section 5.1 one of the requirement for the validation platform is to build a test environment that allows using the target simulator's features for saving and restoring simulation snapshot.

This feature is pivotal because it allows huge improvements in the simulation time, since it makes it possible to run just a portion of the simulation in the interval near the time of injection, rather than the full injection for each fault.

When running the golden simulation, a number of snapshots will be saved at predetermined intervals. Once a fault will be injected at a certain time, instead of running the simulation from the beginning, the snapshot closest in time to the time of injection of the fault will be chosen (see Figure 12).

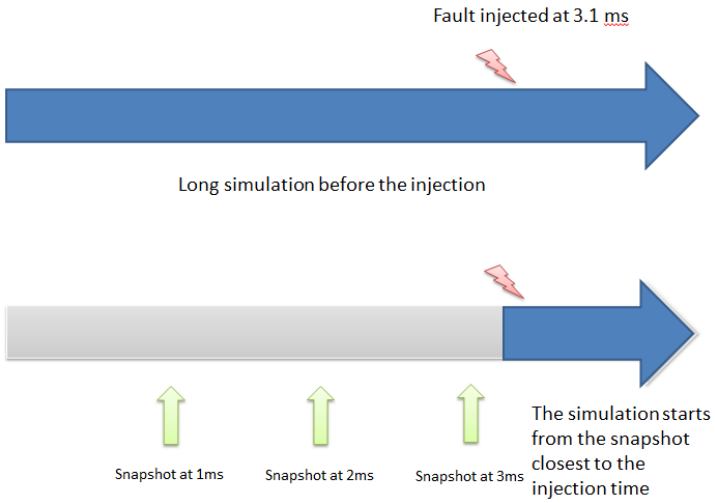


Figure 12: Use of snapshots

The most significant source of optimization, however, comes from the development and use of the Operational Profiler, as described in the next section.

6.3.4. Operational Profiler

The operational profiling is a solution to dramatically reduce the number of faults to be injected during a simulation and offers a big improvement in meeting the performance challenges of the low-level fault injection.

This technique is being refined, engineered and improved, with respect to the original concept [1] in order to specifically target the challenges of the CLERECO validation environment.

The basic goal is to determine upfront in which instants a fault at a certain location may produce any visible consequence at output pins or not. This information can be used to identify the **Windows of Opportunity** (WoO) in which it makes sense to inject a fault knowing that that the fault has a probability to propagate to the observation points (see Figure 13).

The simulation is pre-analyzed in order to identify the "inactivity windows", i.e., all those intervals where it is guaranteed that the fault is not propagated (e.g., a fault injected on a register is not propagated to its fan-out due to the selection of a different input of a multiplexer).

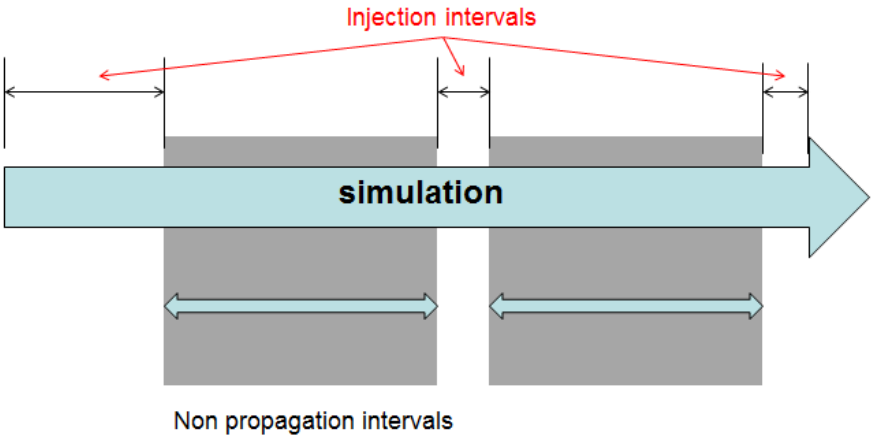


Figure 13: Identifying the WoO

The technique combines both static (analysis of the design database) and dynamic (simulation) analysis to identify the "inactivity windows". This analysis allows detecting the masking condition of the device without having to actually run a full simulation and allows a fairly good level of simplification.

This mechanism can be made hierarchical, starting from sub-blocks and propagating the faults to the HW top, enlarging the non-propagation intervals as the analysis moves up the stack.

Operational Profiler Concept

The idea behind the technique works on a concept called "parasitic simulation": given a particular test bench or set of stimuli driving an RTL design, the related fault tolerance is assessed using data generated by the RTL simulation – the vectors u and x – in order to check the observability conditions and hence screening unobservable faults and potentially observable faults.

The vectors u and v can be explained considering a logic module as a Mealy machine, expressed as a pair of logic equations as reported in Figure 14.

$$\mathrm{x}(\mathrm{k}+1) = \mathrm{f}\big(\mathrm{x}(\mathrm{k}); \mathrm{u}(\mathrm{k})\big)$$

$$\mathbf{y}(k) = \mathbf{g}\big(\mathbf{x}(k); \mathbf{u}(k)\big)$$

Figure 14: logic equation - Mealy machine

Where:

- $\mathbf{x}(\mathrm{k})$ is the space-state array, i.e., the array of flip flops included in the module.
- $\mathbf{u}(\mathrm{k})$ is the array of input Boolean variables.
- $\mathbf{y}(\mathrm{k})$ is the array of output Boolean variables.

$\mathbf{f}$ and $\mathbf{g}$ are combinational functions.

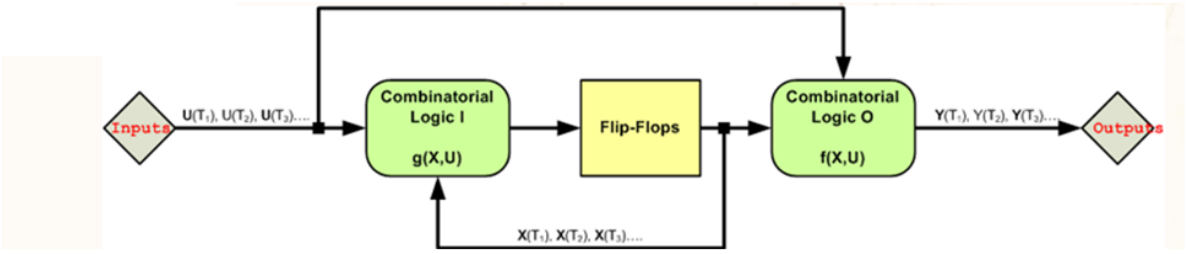


Figure 15: mealy machine

If we know $\mathbf{f}, \mathbf{g}, \mathbf{u}(\mathrm{k})$ and $\mathbf{x}(\mathrm{k})$ at a certain instant k of the simulation, a sufficient condition or the *non-observability* of a fault affecting a certain state at the k-th instant is:

$$\mathbf{f}(0,\mathbf{x_2}(\mathrm{k})\ldots\mathbf{x_N}(\mathrm{k});\mathbf{u_1}(\mathrm{k}),\mathbf{u_2}(\mathrm{k}),\ldots,\mathbf{u_M}(\mathrm{k}))=\mathbf{f}(1,\mathbf{x_2}(\mathrm{k}),\ldots\mathbf{x_N}(\mathrm{k});\mathbf{u_1}(\mathrm{k}),\mathbf{u_2}(\mathrm{k}),\ldots,\mathbf{u_M}(\mathrm{k}))$$

Expression 1

while a necessary condition of the observability of the same fault at the same instant is:

$$g(0,x_2(k)...x_N(k);u_1(k),u_2(k),...,u_M(k)) \neq g(1,x_2(k),...x_N(k);u_1(k),u_2(k),...,u_M(k))$$

Expression 2

These logic conditions are those used to separate observable and unobservable faults occurring at the instant k. It is enough to just solve combinational expressions, under the condition the $\mathbf{f}$ and $\mathbf{g}$ are known.

- **If Expression 1 is satisfied and Expression 2 is not satisfied, the fault does not propagates at instant k.**
- If Expression 2 is satisfied, the fault may or may not be observable depending on how it propagates outside the module.
- If Expression 1 is satisfied and Expression 2 is not satisfied, the fault does not propagates at instant k.

The faults satisfying the second conditions are called **potentially observable**.

The **"parasitic simulation"** will determine the vectors $\mathbf{u}$ and $\mathbf{v}$ at each instant of a RTL simulation, which will be used, together with $\mathbf{f}$ and $\mathbf{g}$, to evaluate the logic expressions.

Operational Profiler Flow

- The Operational Profile analyzes the RTL code and calculates the functions $\mathbf{f}$ and $\mathbf{g}$ for each module in the model and creates a list of relevant signals that are part of $\mathbf{u}$ and $\mathbf{v}$
- An RTL simulation is run – using a standard functional simulator. For each module in the RTL model, $\mathbf{u}$ and $\mathbf{x}$ are stored in vector files (named with a .vcd file extension).
  - A single simulation is needed.
  - Relevant data are stored at clock edges
- Using $\mathbf{x}$, $\mathbf{u}$, the observability conditions are checked using $\mathbf{f}$ and $\mathbf{g}$.
- Unobservable faults are set apart.
- Potentially observable faults are propagated, using the $\mathbf{f}$ and $\mathbf{g}$ functions, until they reach any output port at the top level (or any observation point), until the discarded during the process.

The process turns out to be very fast:

- The observability conditions for each fault (i.e., a pair $\mathbf{x}, \mathbf{k}$) are checked "locally" within the module the injection point belongs to, by solving $\mathbf{f}$ and $\mathbf{g}$, and using the data stored after the RTL simulation.
- When a fault is potentially observable, it is checked for actual observability by propagating it only to neighbor modules, again using the simulation data, and solving $\mathbf{f}$ and $\mathbf{g}$.
- The check only takes place at clock edges.
- Simulation may be limited to a subset of all the clock instants.

Inactivity Windows

In a complex system, there are many components interconnected one another, but it is unlikely that all these components are active at the same time. In general, only a subset of these components is switching at a certain instant. Moreover, we may encounter periods when the state vector $\mathbf{x}(k)$ and the input vector $\mathbf{u}(k)$ do not change. Consequently, during these intervals no changes may occur at the outputs $\mathbf{y}(k)$ or the next state vector $\mathbf{x}(k+1)$.

Finally, considering complex tasks executed by the system over a long period of time, involving the cooperation of many components, it is likely that for most components, each of them will be working for a small fraction of the whole system activity (see Figure 16).

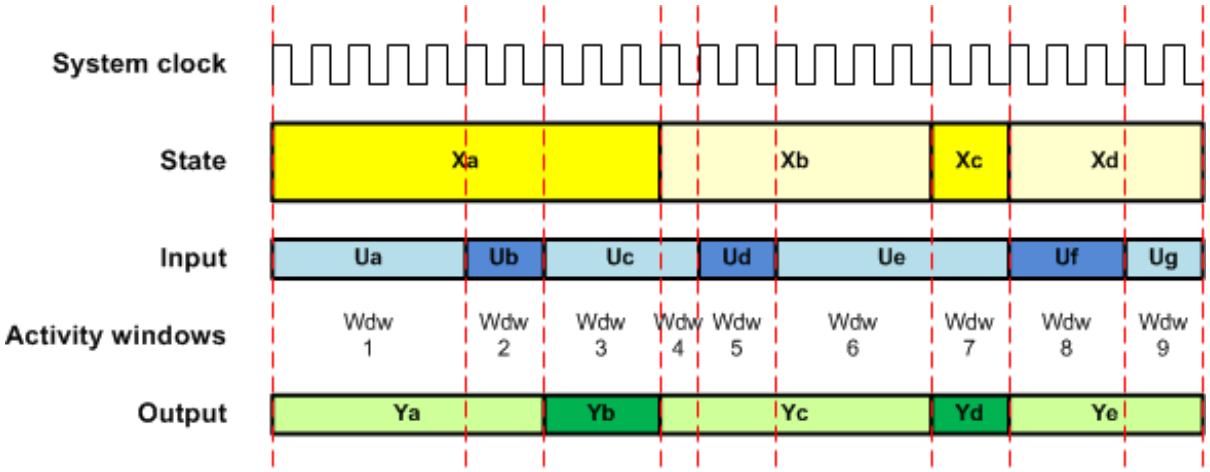


Figure 16: Inactivity Windows

Based on these considerations, we can introduce a great simplification: observability conditions need to be checked only when some changes occur, at $\mathbf{u}$ or at $\mathbf{x}$.

The boundaries of inactivity windows are a limited subset of the whole set of clock cycles. If the process limits its computations only to those windows, we have a significant simplification with respect to the standard RTL simulation.

It is also worth noticing that a .vcd file, given its structure intended to store differential information, automatically provides a simple way of building the inactivity windows of a certain module.

Computation of f and g

We introduce the concept of "leaf component" and "grey component".

A "leaf component" (Figure 17) is a module in the design hierarchy that includes:

- Input signals.
- Output signals.
- State elements (flip-flops).
- Combinational logic.
- No instances of sub modules.

A grey component (Figure 18) also includes instances of sub modules. In this case:

- The inputs of the sub modules are considered as output of the including module.
- Dually the output of the sub modules are considered as inputs of the including module.

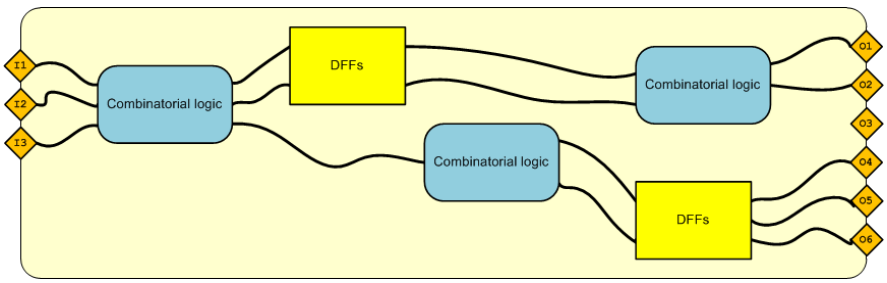


Figure 17: leaf component

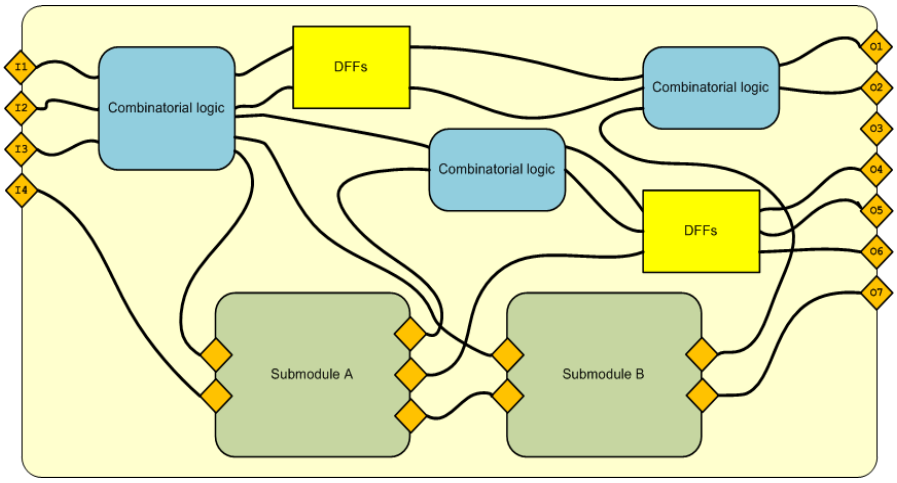


Figure 18: grey component

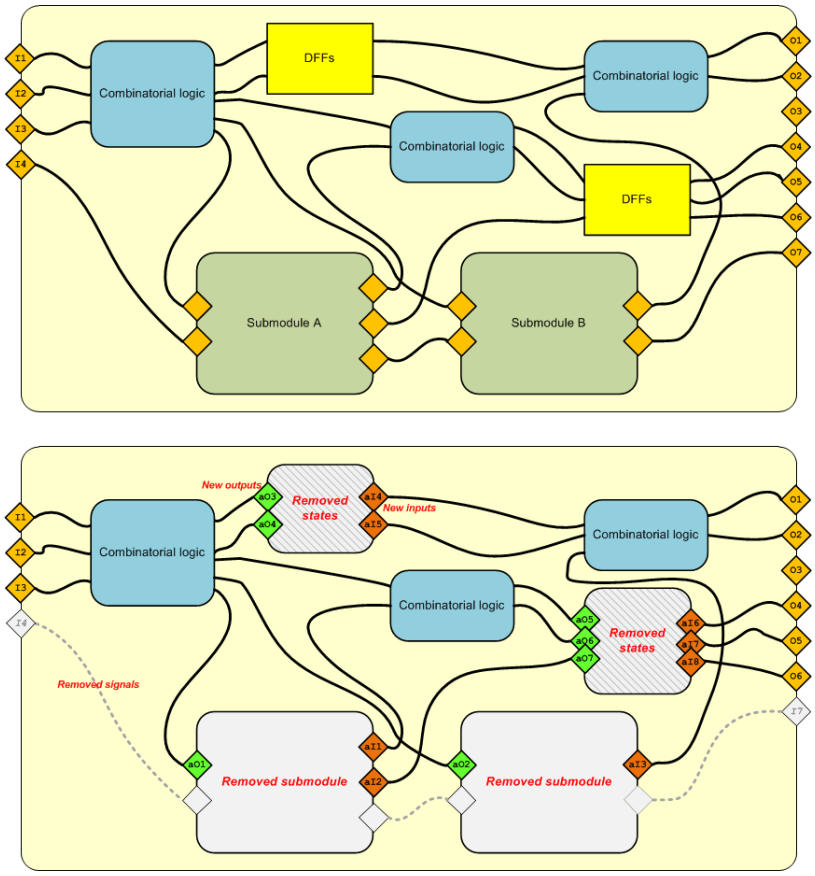


Figure 19: Substitution

Each module can be reduced to a piece of standalone combinational logic by removing internal sub modules and internal states (clocked process). In order to guarantee logic equivalence, the removed parts need to be replaced by the related signals at their boundaries (Figure 19 and Figure 20). During this process, some new I/O ports are added while others may be removed. I/O ports directly mapped to other I/O ports, without any combinational functions in the middle, may be removed.

The remaining logic is combinational logic and it is used to build the functions **f** and **g**:

- The union of the fan-in of all the registers is used to build **f** (state function).
- The union of the fan-in of all the output ports is used to build **g** (output function).

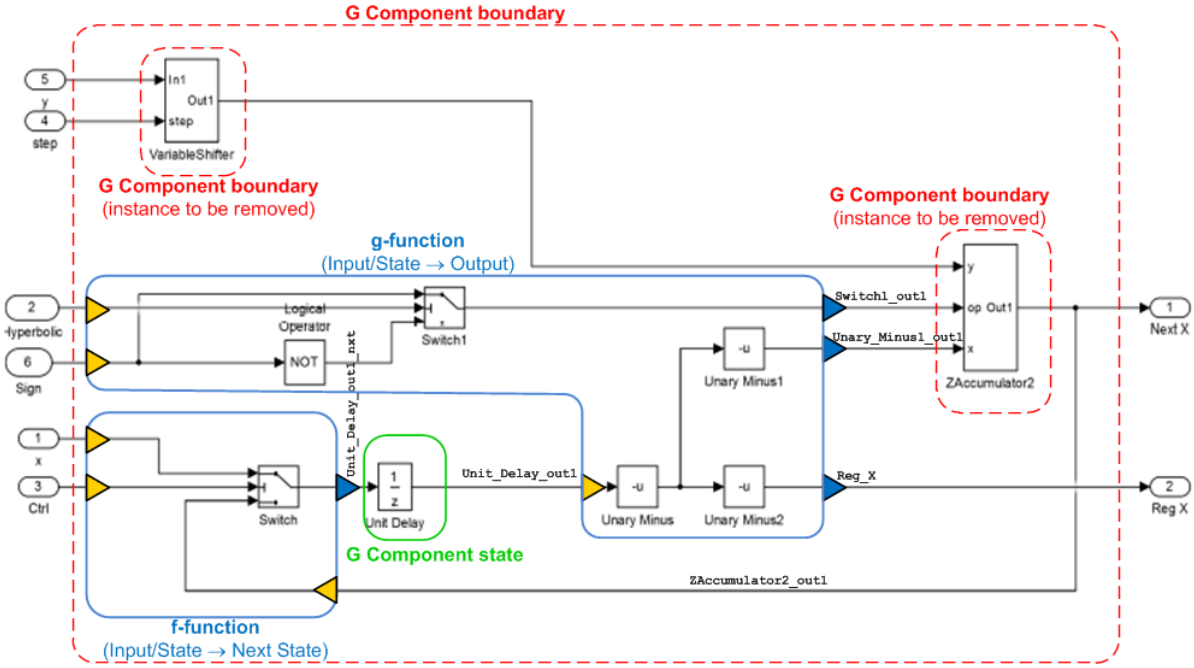


Figure 20: Actual Example

f and **g** are built with a manipulation of the original RTL code (Figure 21) by:

- Removing the sub modules,
- Adding the related I/O ports,
- Reducing all the clocked processes to the related combinational logic, and,
- Adding the related I/O ports to module definition.

Some attention must be paid to the transformation where asynchronous signals are present, because the logic is transformed into the equivalent of a synchronous reset.

At the end of the process **f** and **g** are simple and purely combinational chunks of RTL code that can be solved individually by a simple logic expression solver.

```
always @(posedge clk or posedge reset)
   begin : Unit_Delay_process
     if (reset == 1'b1) begin
       Unit_Delay_out1 <=
24'sb000000000000000000000000;
     end
     else begin
       if (enb) begin
         Unit_Delay_out1 <= Switch_out1;
       end
     end
   end
```

```
always @(*)
   begin : Unit_Delay_process
     if (reset == 1'b1) begin
       Unit_Delay_out1_nxt <=
24'sb000000000000000000000000;
     end
     else begin
       if (enb) begin
         Unit_Delay_out1_nxt <= Switch_out1;
       end
     end
   end
```

Figure 21: transformation of the clocked processes.

7. Validation workflow

Figure 22 provides the detailed validation work workflow, with the detail of all used tools.

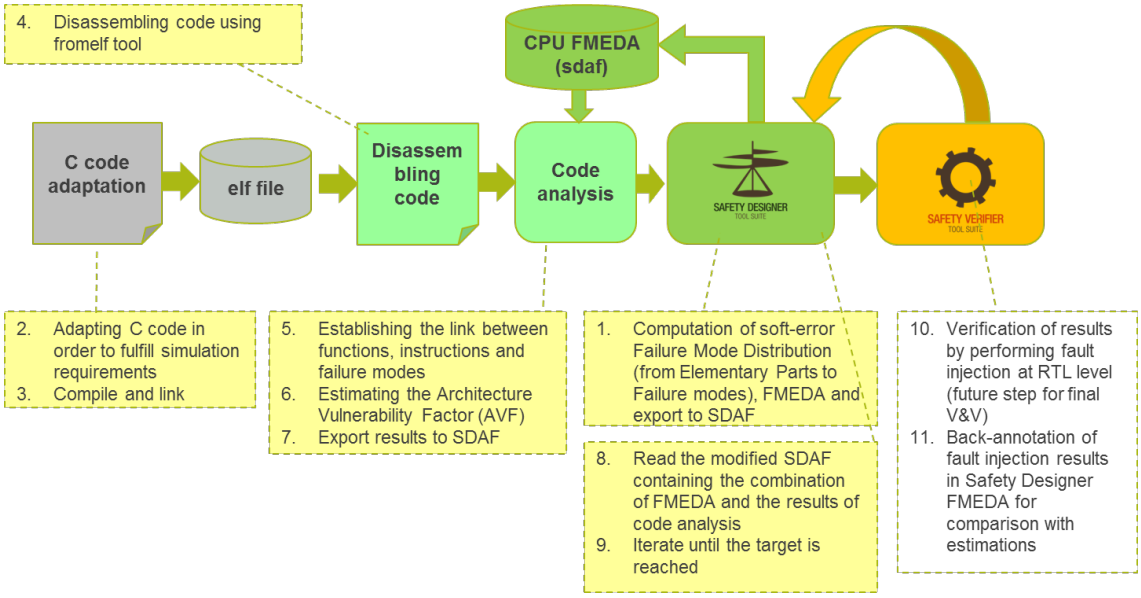


Figure 22: Methodology flow and tools

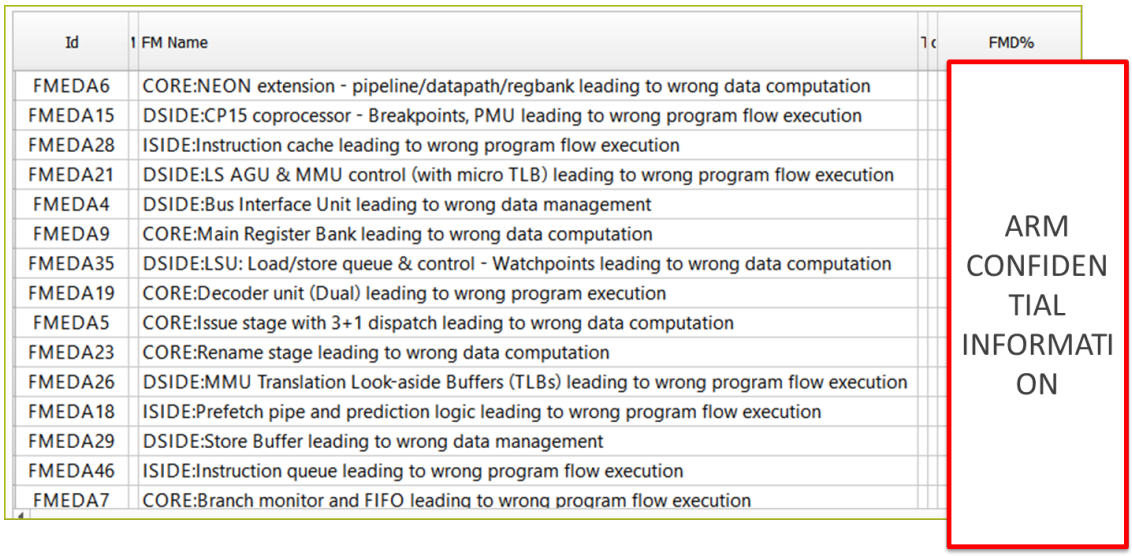
1. Failure Mode Distribution and FMEDA

The step 1 is performed using the YOGITECH Safety Designer (SD) tool. It consists into the computation of soft-error distribution (from Elementary Parts to Failure modes) and FMEDA. The SD takes as input the CPU netlist and provides as output all the needed safety metrics. The output comes from the association between Elementary Parts (EP, flip-flop + logic cone) and Failure Modes (FM). Figure 23 and Figure 24 show the FMEDA analysis performed on the ARM Cortex A9, respectively, including and excluding L1 caches. The output of this step is also called CPU FMEDA.

Id	FM Name	FMD%
FMEDA20	RAM: Memory cells failure	
FMEDA6	CORE:NEON extension - pipeline/datapath/regbank leading to wrong data computation	
FMEDA15	DSIDE:CP15 coprocessor - Breakpoints, PMU leading to wrong program flow execution	
FMEDA28	ISIDE:Instruction cache leading to wrong program flow execution	
FMEDA21	DSIDE:LS AGU & MMU control (with micro TLB) leading to wrong program flow execution	
FMEDA4	DSIDE:Bus Interface Unit leading to wrong data management	
FMEDA9	CORE:Main Register Bank leading to wrong data computation	
FMEDA35	DSIDE:LSU: Load/store queue & control - Watchpoints leading to wrong data computation	
FMEDA19	CORE:Decoder unit (Dual) leading to wrong program execution	
FMEDA5	CORE:Issue stage with 3+1 dispatch leading to wrong data computation	
FMEDA23	CORE:Rename stage leading to wrong data computation	
FMEDA26	DSIDE:MMU Translation Look-aside Buffers (TLBs) leading to wrong program flow execution	
FMEDA18	ISIDE:Prefetch pipe and prediction logic leading to wrong program flow execution	
FMEDA29	DSIDE:Store Buffer leading to wrong data management	
FMEDA46	ISIDE:Instruction queue leading to wrong program flow execution	

ARM CONFIDENTIAL INFORMATION

Figure 23: FMEDA analysis including L1 caches



Id	FM Name	FMD%
FMEDA6	CORE:NEON extension - pipeline/datapath/regbank leading to wrong data computation	
FMEDA15	DSIDE:CP15 coprocessor - Breakpoints, PMU leading to wrong program flow execution	
FMEDA28	ISIDE:Instruction cache leading to wrong program flow execution	
FMEDA21	DSIDE:LS AGU & MMU control (with micro TLB) leading to wrong program flow execution	
FMEDA4	DSIDE:Bus Interface Unit leading to wrong data management	
FMEDA9	CORE:Main Register Bank leading to wrong data computation	
FMEDA35	DSIDE:LSU: Load/store queue & control - Watchpoints leading to wrong data computation	
FMEDA19	CORE:Decoder unit (Dual) leading to wrong program execution	
FMEDA5	CORE:Issue stage with 3+1 dispatch leading to wrong data computation	
FMEDA23	CORE:Rename stage leading to wrong data computation	
FMEDA26	DSIDE:MMU Translation Look-aside Buffers (TLBs) leading to wrong program flow execution	
FMEDA18	ISIDE:Prefetch pipe and prediction logic leading to wrong program flow execution	
FMEDA29	DSIDE:Store Buffer leading to wrong data management	
FMEDA46	ISIDE:Instruction queue leading to wrong program flow execution	
FMEDA7	CORE:Branch monitor and FIFO leading to wrong program flow execution	

Figure 24: FMEDA analysis excluding L1 caches

2 & 3. C code adaptation

The original workloads written in C code must be adapted in order to fulfill simulation requirements, e.g.: I/O operations are replaced with in-memory operations and input parameters are hard-coded. After these negligible modifications the code is compiled and linked using the ARM tool chain.

4. Disassembling code using fromelf

ELF file generated by the previous step is disassembled using ARM *fromelf* utility. The resulting output is fed to the YOGITECH codeprofiler tool. This tool performs a static code analysis, counting the instruction used by the algorithm.

5. Linking functions, instructions and FMs

The step 5 is the most critical one. In this step is performed the link between functions, instructions and failure modes. The link is based on:

1. Identifying instructions used in each function and "counting" how many times each instruction occurs in that function;
2. Associate each instruction to the failure modes (FMs) identified in the CPU FMEDA
3. Link functions and failure modes by combining the previous steps (1+2 → 3)

6. Architecture Vulnerability Factor (AVF)

The step 6 provides an estimation of the Architecture Vulnerability Factor (AVF) combining three factors: architectural safeness, application safeness and Frequency of Use (i.e. exposure or lifetime).

The architectural safeness is determined by Safety Designer and is related to the hardware (i.e., it is independent of the workload). The application safeness is a predetermined amount of safeness assigned to each function. For example, a not safety relevant function can have a high application safeness. The Frequency of Use describes the amount of time each failure mode is exposed to soft-errors, derived from the link between functions, instructions and failure modes established in the previous steps. It also considers the expected pipeline performance (i.e. the duration of pipeline stage, in terms of clock cycles) and the fact that certain logic has a longer "life time", i.e. it causes errors and failures even if not activated by a certain instruction.

7 & 8 & 9. Updated FMEDA

The results of the previous steps are read back into Safety Designer in order to combine HW and SW workload metrics. In other words, the CPU FMEDA analysis (hardware related) is combined with the code analysis (software related). An example is shown in Figure 25.

Id	FM Id	FM Name	Fsafe	Krf	Klat	λSR	λsafe	λns	λSPF
FMEDA43	FM_C-1	RAM:Instruction side RAM - drive logic leading to wrong program execution	99.00%	39.8...	100.0...	3.716...	3.679E-...	3.716...	0.0
FMEDA37	FM_C-2	RAM:Data side RAM - drive logic leading to wrong data computation	85.46%	65.3...	100.0...	4.908...	4.194E-...	7.134...	0.0
FMEDA8	FM_C-3	RAM:Data side RAM - clock gating leading to wrong data computation	85.46%	14.2...	100.0...	4.533...	3.874E-...	6.589...	0.0
FMEDA14	FM_C-4	RAM:TLB RAM - drive logic leading to wrong program execution	99.00%	85.4...	100.0...	8.146...	8.065E-...	8.146...	0.0
FMEDA6	FM_C-8	RAM:Instruction side RAM - clock gating leading to wrong program execution	99.00%	70.5...	100.0...	5.391...	5.337E-...	5.391...	0.0
FMEDA19	FM_C-9	RAM:TLB RAM - clock gating leading to wrong program execution	99.00%	91.1...	100.0...	1.514...	1.499E-...	1.514...	0.0
FMEDA20	FM_C-10	CLOCK:CPU clock module leading to wrong program flow execution	40.70%	10.8...	100.0...	4.869...	1.982E-...	2.887...	0.0
FMEDA7	FM_C-11	CORE:Execution mode control leading to wrong program flow execution	44.82%	70.7...	100.0...	2.055...	9.212E-...	1.134...	0.0
FMEDA21	FM_C-12	CORE:Decoder unit (Dual) leading to wrong program execution	44.08%	0.14%	100.0...	1.679...	7.401E-...	9.388...	0.0
FMEDA13	FM_C-13	CORE:Rename stage leading to wrong data computation	44.82%	0.21%	100.0...	1.168...	5.234E-...	6.443...	0.0
FMEDA31	FM_C-14	CORE:Rename stage - late rename leading to wrong data computation	44.82%	0.00%	0.00%	1.443...	6.467E-...	7.961...	0.0

Figure 25: FMEDA combining HW and SW metrics

10 & 11. Verification & Validation (V&V)

In the V&V step the workload is simulated with CPU RTL and soft-errors are fault injected by using YOGITECH Safety Verifier (SV). A Safety Verification Plan (SVP) is created by Safety Designer based on the updated FMEDA. At the end of the fault injection campaign, the Safety Verifier results are then back-annotated into Safety Designer for comparison with estimations.

8. Preliminary experimental setup

As a preliminary validation setup we set-up and experiment working on a system based on the ARM Cortex A9 running a set MiBench applications [3] often as benchmarks for reliability analysis. In this preliminary experiment the validation campaigns focuses on faults in the L1 cache memory and in the Register File, composed of, respectively, 32 Kb of memory and 56 integer registers.

The fault injection campaign is conducted using 194.7E-7 FIT/bit as raw FIT (single bit FIT rate) for both logic and memory, extracted from the CLERECO technology library for a 22nm Bulk Planar technology node. Moreover, the observation points used are the output of the related modules (outputs of L1 and outputs of Register File).

The number of injected faults is computed using these parameters:

1. Confidence level: 90%
2. Error margin: 1%

After the computations, the injected faults are 4161 and 4352 for, respectively, the Register File and the L1 cache memory.

The analyzed MIBench applications are:

1. String search
2. Susan smooth
3. Susan edges
4. Susan corner
5. Aes encryption
6. Quick sort
7. Fft

8. Sha

8.1. Safety Verifier setup

The YOGITECH Safety Verifier tool allows setting a great number of parameters. One of the most relevant is T1. T1 represents the number of clock cycles, starting from the injection time, which there will be visible effects on the observation points. If after T1 clock cycles there are no visible effects on the observation points than the injected fault is considered safe. For the Register File fault injection campaign T1 is set to 1000 while for the L1 cache case T1 is set to 2000. This difference is due to the delay introduced by the memory access. Is clear that these numbers follow a conservative approach.

8.2. Results

Figure 26 shows failure rates for the different benchmarks computed using the YOGITECH workflow (identified with the YT label) for each workload, while Figure 27 focuses on the Register File injected (YT3) given that it is out of scale on Figure 26.

For each benchmark the figures show:

1. YT1: L1 cache and Register File injected
2. YT2: L1 cache injected
    - Logic is not considered (its FIT is removed)
3. YT3: Register File injected
    - L1 cache and logic is not considered (their FIT is removed)

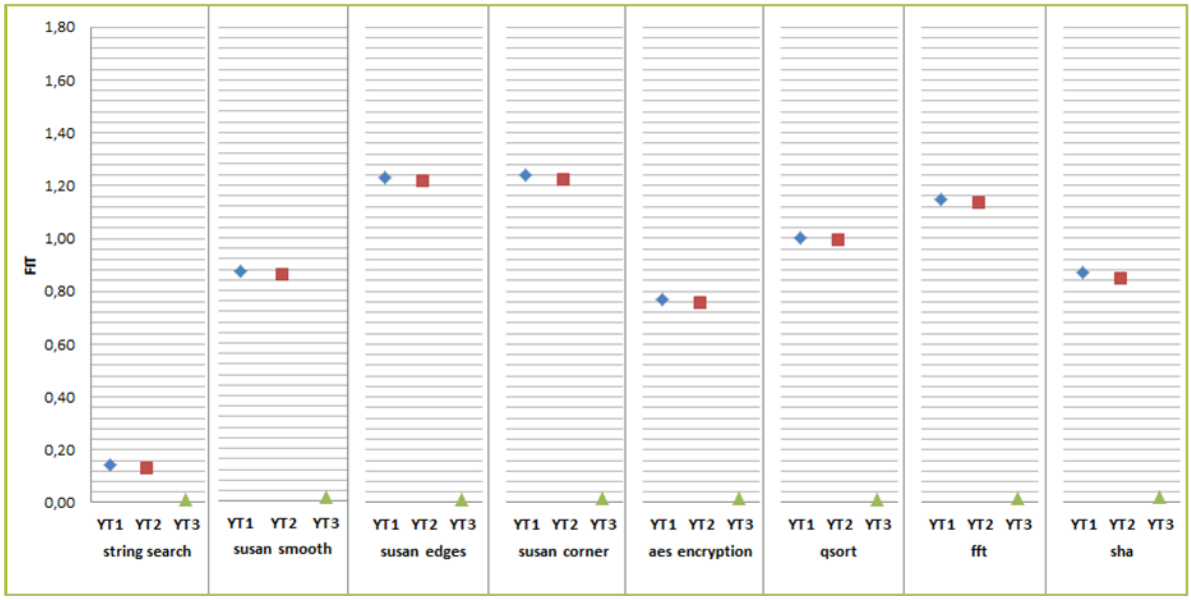


Figure 26: Fault injection results with Safety Verifier (overall view)

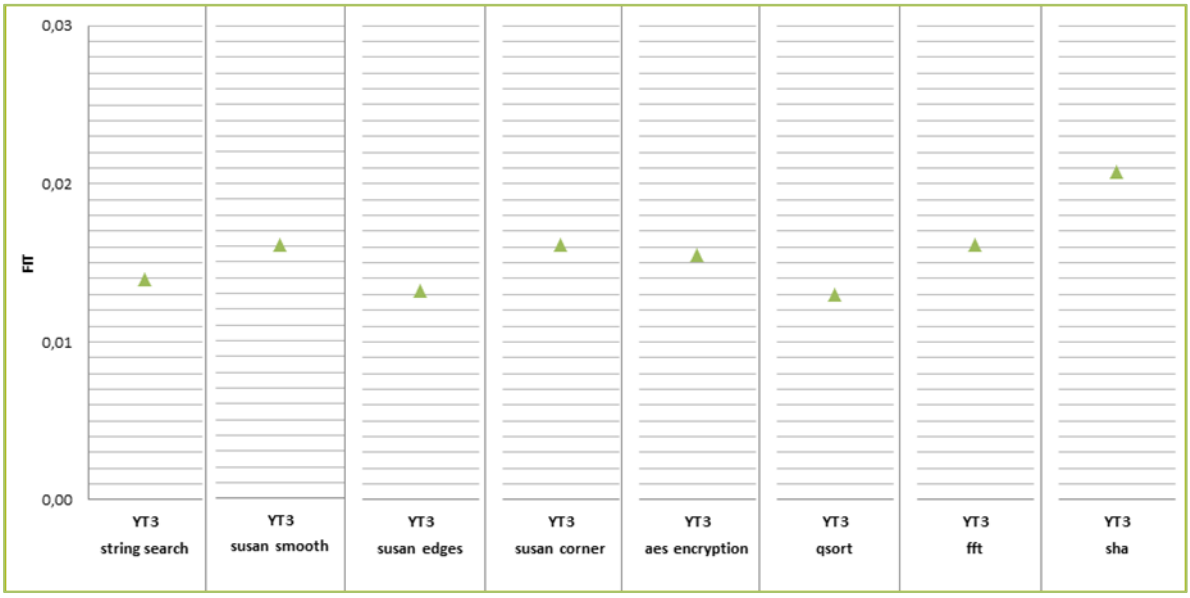


Figure 27: Fault injection results with Safety Verifier (zoom for Register File – YT3)

8.3. RTL Injection vs. CLERECO system level analysis

This section compares results obtained by performing RTL injection using YOGITECH workflow and the results provided by CLERECO System Reliability Analyzer described in deliverable D5.2.2. There are some differences between YOGITECH setup and CLERECO setup. Actually, the observation points used by YOGITECH are measuring the masking factor related to the usage of the module under test while CLERECO System Reliability Analyzer the overall masking factor at the end of the algorithm. Moreover the Register Files under test have different sizes:

4. YOGITECH: 56 integer registers
5. CLERECO: 256 integer registers

Figure 28 shows that the trend between the two methods is the same, which is a very positive result. The figure also confirms that results obtained using the YOGITECH workflow are greater (most pessimistic) than CLERECO values, as expected, due to the different masking factor.

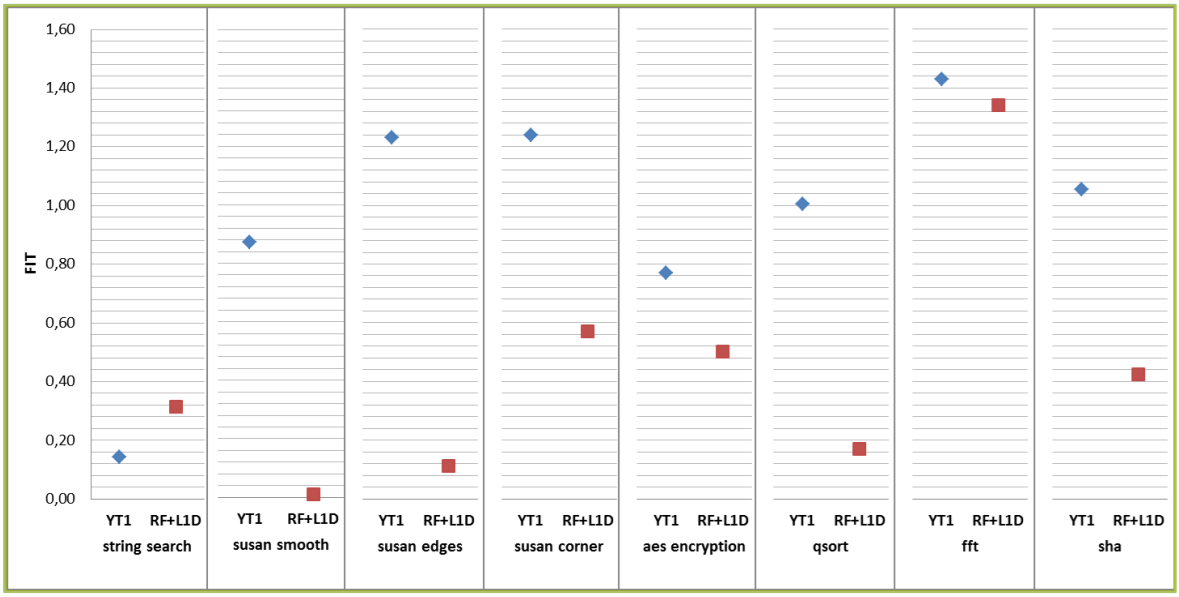


Figure 28: Comparison YT1 vs. "results RF + L1D"

Figure 29 is quite similar to Figure 28 because the L1 contribution is dominant. As Figure 28, the trend is preserved among algorithms.

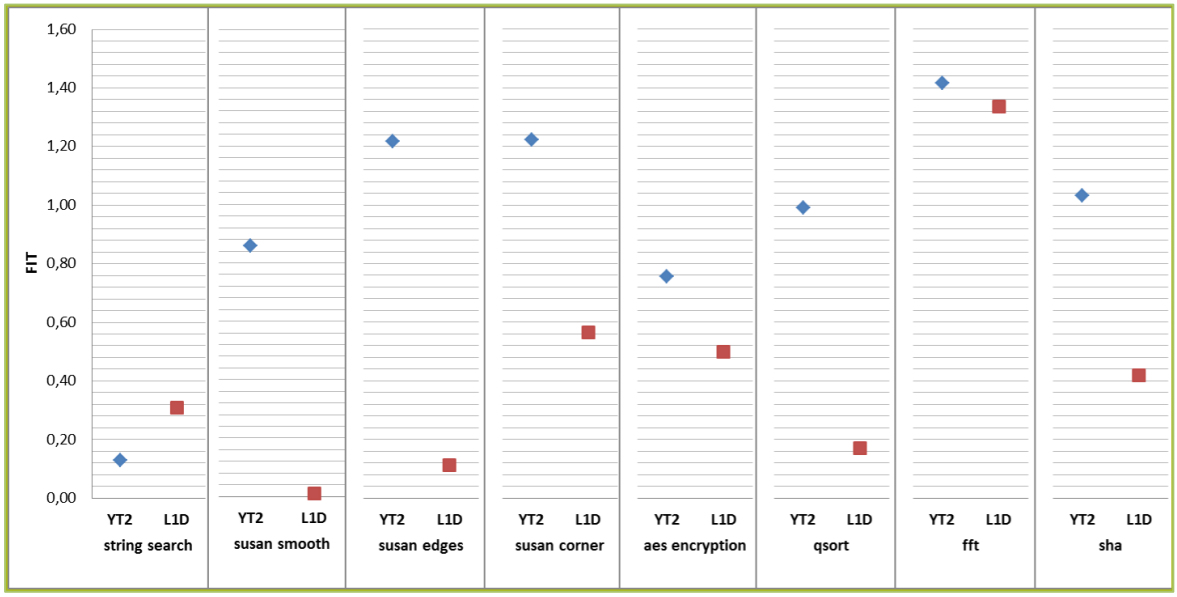


Figure 29: Comparison YT2 vs. "results L1D"

Given that the Register File injected by CLERECO and YOGITECH are different the following comparison provides two figures: Figure 30 shows the comparison between numbers *as is*, while Figure 31 shows a *normalized* version of the CLERECO numbers. The *normalized* version is computed as RF/ (256/56). Obviously, this is a simplification given that the FIT is considered linear with respect to the hardware size.

The trend showed by Figure 28 and Figure 29 is confirmed also in Figure 30 and Figure 31, as expected. However, in this case the difference between YOGITECH and CLERECO numbers is lower than the previous case (in terms of absolute FIT) because the size of the Register File is some order of magnitude lower than L1 cache size. This means that a small variation on measurements implies a great variation on final FIT.

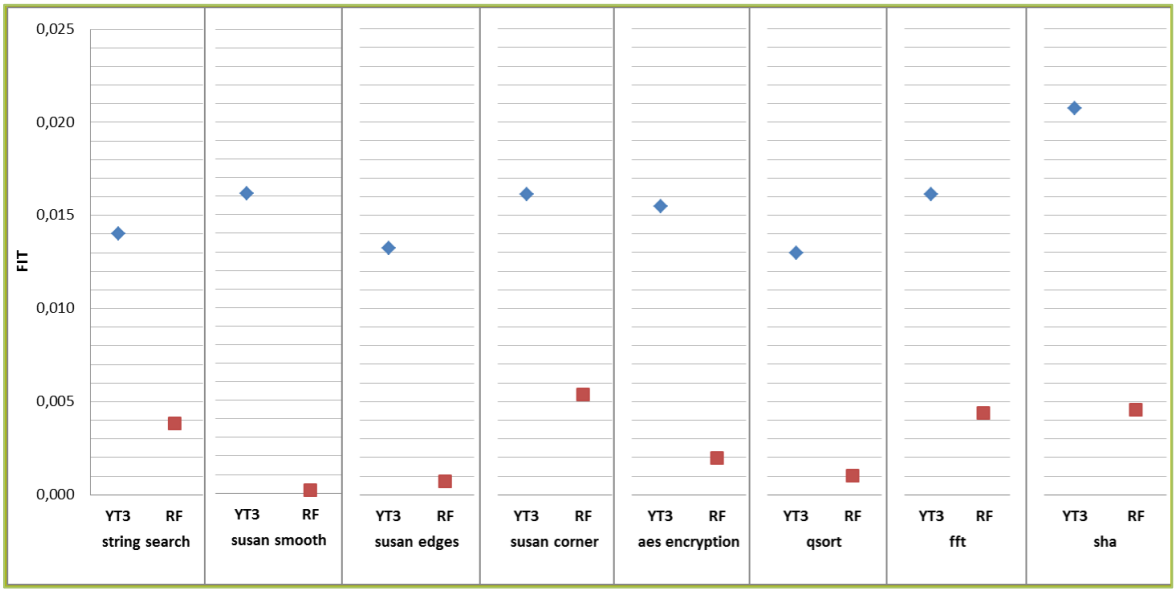


Figure 30: Comparison YT3 vs. "results RF"

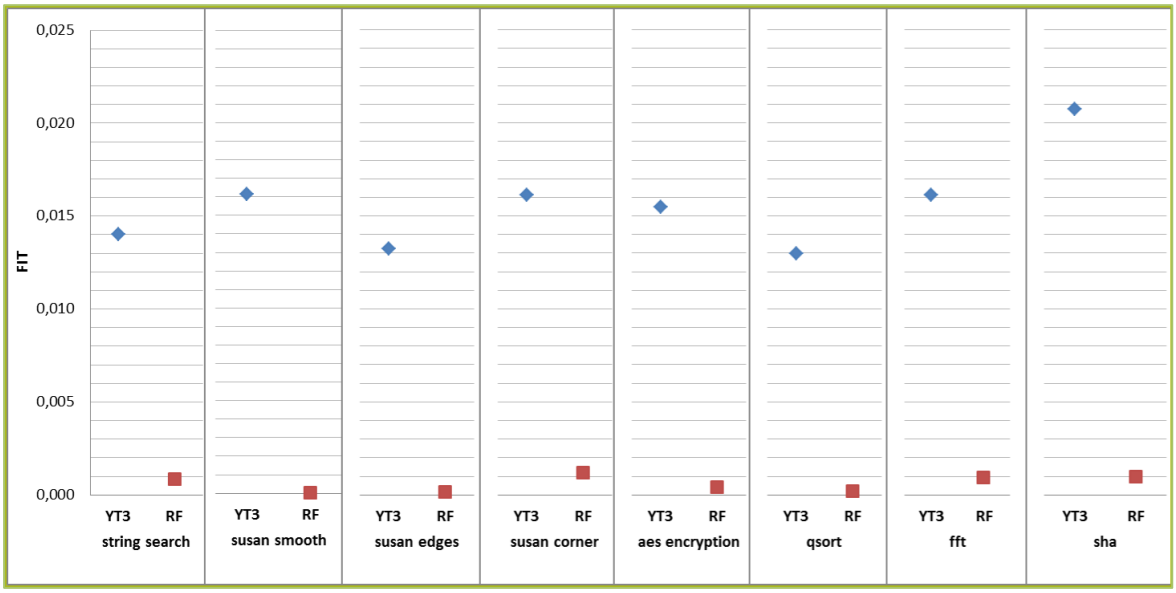


Figure 31: Comparison YT3 vs. "results RF" – normalized

As last comparison session, Figure 32 and Figure 33 show the estimation computed by the YOGITECH Safety Designer tool vs. Register File injected (YT3) vs. CLERECO RF. The estimation is performed, mixing code analysis with hardware considerations. The time required by the Safety Designer estimation is very small compared with the time required by a fault injection campaign. As for the previous case, there is the *as is* version (Figure 32) and the *normalized* version (Figure 33).

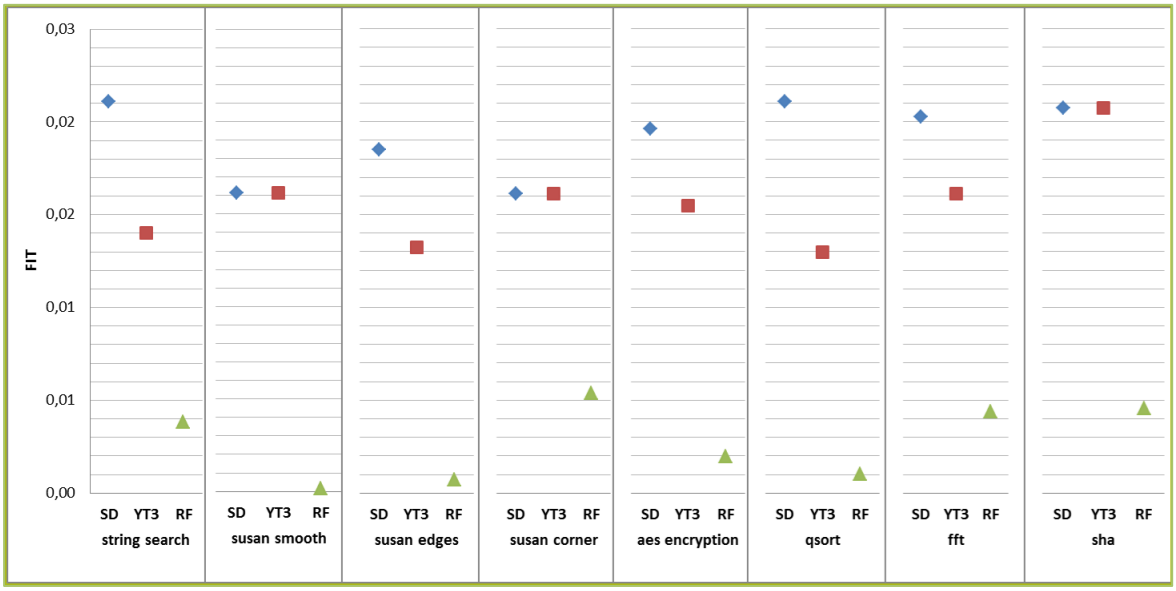


Figure 32: Comparison Safety Designer (estimation) vs. YT3 vs. "results RF"

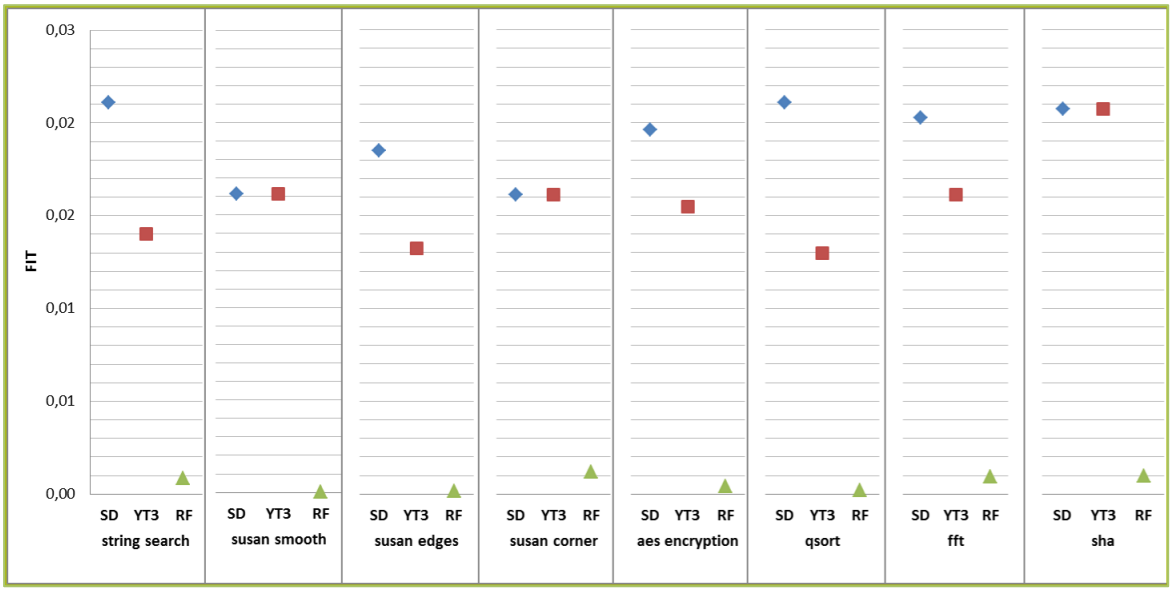


Figure 33: Comparison Safety Designer vs. YT3 vs. "results RF" – normalized

9. Next steps

The results provided in this document shows that there is a small distance between the FIT computed by YOGITECH and CLERECO. The main causes of those differences are the observation points. For this reason, the fault injection campaign conducted by YOGITECH will be performed again using different observations points:

1. CPU overall outputs
2. Algorithm overall outputs

The step 1 can be performed changing the observation points and using the YOGITECH methodology as is, while the step 2 introduces some issues due to the modification of the approach. The implementation of the Step 2 implies that the simulation must run until the end, on

faulty cases. This means that the injection time increases abnormally. For example, for the string search algorithm, the injection time (per fault) changes from few seconds to several minutes.

Using this new setup, YOGITECH results should be closer to CLERECO results. Figure 34 shows the link among the methodology and the final FIT. The fault injection case can provide different values based on the chosen observation points.

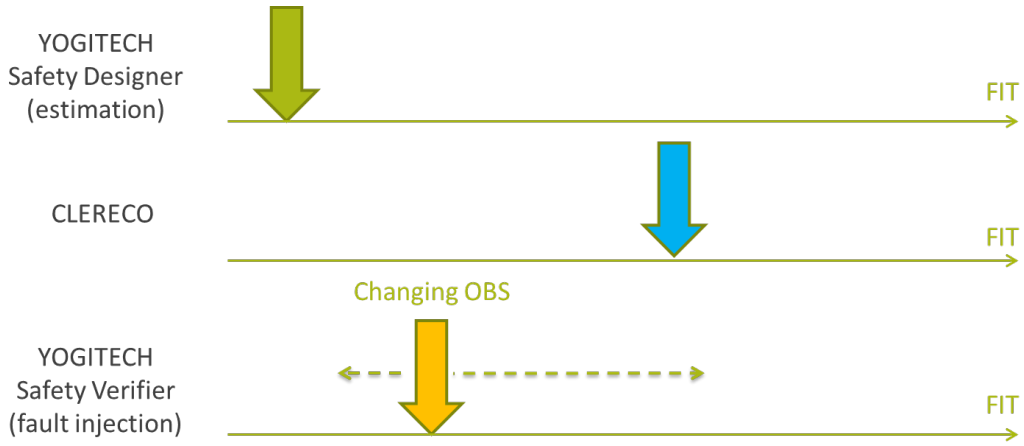


Figure 34: Summary view of results (to be confirmed by final injections)

Figure 35 shows an estimation of the simulation time / effort with respect to the results accuracy. The figure indicates that the time required by the fault injection campaign is exponential with respect to the provided accuracy, while the time required by CLERECO should be order of magnitude lower than the fault injection methodology ensuring, however, an elevated accuracy. Lastly, the time required by the Safety Designer (estimation) is the smallest of all, as its accuracy.

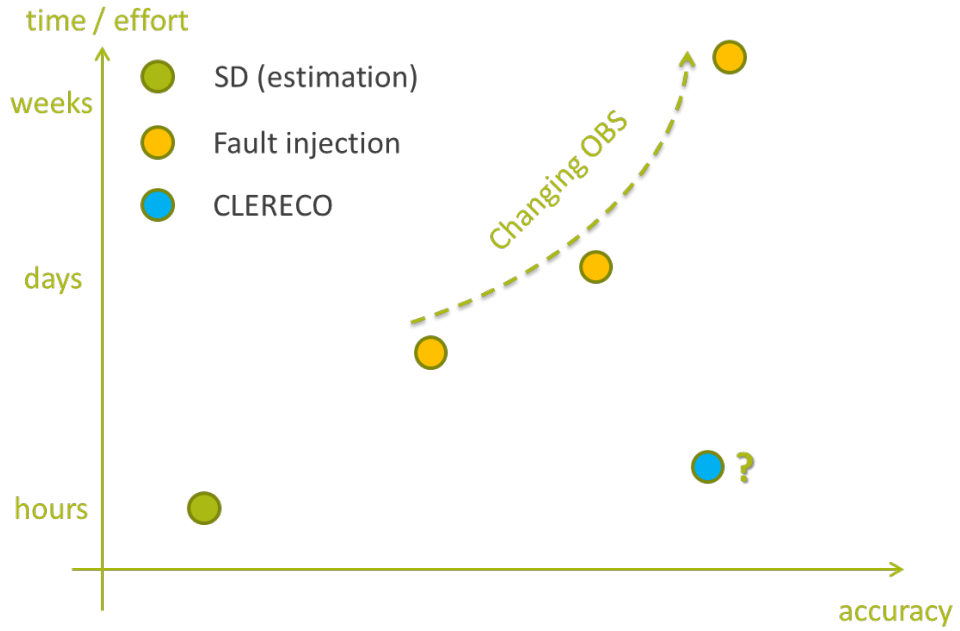


Figure 35: Simulation time / effort vs. accuracy

10. Bibliography

[1] Benso, A.; Bosio, A.; Di Carlo, S.; Mariani, R., "A Functional Verification based Fault Injection Environment," Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on , vol., no., pp.114,122, 26-28 Sept. 2007, doi: 10.1109/DFT.2007.31

[2] R. Leveugle, A. Calvez, P. Maistri e P. Vanhauwaert, «Statistical fault injection: Quantified error and confidence,» Design, Automation & Test in Europe Conference & Exhibition, 2009.

[3] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, 2001. WWC-4. 2001, pp.3-14, 2 Dec. 2001