A Novel Transient Fault Injection Methodology Based on STE Model Checking

Ashish Darbari†, Bashir Al Hashimi†, Peter Harrod†† and Daryl Bradley††
†*ECS Department, University of Southampton, Southampton, England*
††*ARM Ltd.,Cambridge, England*

***Abstract*— The aim of this paper is to propose a new transient fault injection methodology based on Symbolic Trajectory Evaluation (STE) model checking, that addresses the limitations of the recently proposed techniques on sudying fault injection using property checking [1–3]. To demonstrate the capability of our proposed methodology, we instrument the necessary properties, and verify them using the STE model checker Forte, to analyse the effect of transient faults on the fetch unit of a 32-bit multi-cycle processor, synthesized using Altera Quartus II. Our approach can be applied generally to any faulty processor, to compute precise inter-dependencies among different functional units, thereby lending better understanding of the faulty behaviour in an efficient way.**

I. Introduction

The goal of fault tolerance is to make the hardware work correctly in the presence of faults. This usually entails conducting a study on fault injection on a design, to learn about the possible effect of faults, and then devise a technique to mask the effect of these faults.

In a formal verification approach we pose the question: Is a given property satisfied by a circuit model? The answer obtained is usually in the form of a Boolean expression, which can be simply *true*, or *false*, or a compound formula built over some Boolean variables known as a counter-example. A counter-example is a precious by-product of a formal verification run because it encapsulates vital debugging information, that can be used by the designers judiciously to fix the source of the bug.

We use a formal property checking approach based on Symbolic Trajectory Evaluation (STE) based model checking to evaluate the effect of transient faults (often known as single-event-upsets (SEU)), on the fetch unit of a CPU. We demonstrate how one can compute and monitor the trajectory of a fault from the source to different points of interest in the fetch unit. In this paper whenever we use the term fault, we mean an SEU.

II. Related work

Simulation based fault injection is a special case of software fault injection [4–6] that "can support a variety of system abstraction levels — architectural, functional, logical and electrical" [7], and for this reason has been used extensively to investigate fault injection. Recently, there has been an emergence of using formal property checking to study fault injection [1–3]. The basic technique employed is to instrument the RTL in a controlled way to incorporate fault injection, and then check the behaviour of the faulty RTL whilst running some benchmark programs, using a formal property language. Property checking based methodology is an instance of a simulation based fault injection technique, but because it computes simulation traces by doing model checking underneath, it provides comprehensive coverage — wider than any conventional simulation that relies on testing on a finite set of patterns.

Hazelhurst et al. [1] were the first to note that one can use formal property checking to specify and verify the effects of stuck-at faults. They proposed using STE to accomplish this, and for this reason their work is most similar to ours. But there are notable differences between our approach and the one Hazelhurst et al. proposed. In their method, faults were modelled by decomposing the finite-state machine representation into smaller components, and each smaller model is modified to incorporate faults that they investigated. The authors deal with three different FSMs, one that represents the RTL under investigation, one that injects the faults, and the third that monitors the effect of faults.

However, manipulating low-level FSMs is *compute intensive*, is *less abstract* and *less intuitive*, and is therefore not desirable. Another shortcoming of their approach was that they hand-converted the VHDL code into an FSM representation, which obviously limits the applicability of their approach.

Leveugle [2] was the first to show how IBM's Property Specification Language (PSL) based property checking can be used together with controlled mutations, for transient fault injection. The basic idea is that the RTL that has to be investigated for faults is instrumented with extra hardware to model faulty behaviour. The author uses mutants [7] for this precise instrumentation technique. Much care was taken in developing these mutants, since having too many of them, will result in state-space explosion. The author noted that using PSL rules in practice was difficult, and often the rules were long and complex. In addition the tools that are used for model checking were not able to say much in case of a contradiction, besides saying "over-constrained". Also the tool Leveugle used was unsound, meaning the tool okayed the properties whilst there were known counter-examples.

The author stesses that in spite of property checking being used it may be important to think of applications running on the target RTL so that one investigates the behaviour of the faulty design keeping in view the target application. This is however a limitation, since the faults being characterised are influenced by the specific benchmark program running on the

processor.

Krautz et al. [3] also used PSL based property checking to evaluate the effect of transients on the coverage of error-detection logic. The basic approach used by them is very similar to Leveugle — instrument the VHDL with faulty hardware, and then use PSL property checking to evaluate the behaviour of the RTL while it executes specific benchmark programs. The authors prefer dealing with two separate RTLs — one the golden RTL which is fault-free and other the fault injected model. Tolerance in the form of error-detection-and-correction is encoded in the RTL and PSL is used to express properties that will check if the tolerance is adequate to mask the faults. The checking is done by comparing the fault-free RTL with the faulty RTL which the authors suggest is important in order to avoid developing complex PSL properties.

Whereas the fault injection scheme in the model by Krautz et al. is slightly simpler than Leveugle, because an extension of VHDL called BugSpray is used, it is still a limiting factor. This is because bloating the RTL with extra hardware makes formal model checking much harder to perform due to extra state variables, leading to huge BDDs.

In summary, Figure 1 shows how fault injection has been studied traditionally. In the next section we present an overview of our methodology that will overcome the limitations pointed out above.

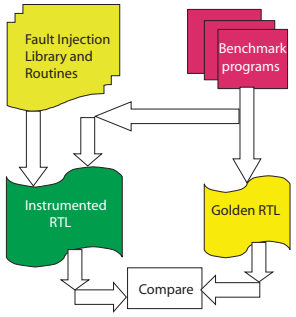

Fig. 1. **Existing fault injection approaches** Traditionally fault injection is carried out by comparing the behaviour of two different RTLs, one that has been instrumented to incorporate faults, and the other – golden RTL – which expresses the intention of the designer in a fault-free case. Both these RTLs are simulated whilst running specific binary program images. In the instrumented RTL, faults are injected using a fault library and injection functions. The comparison is usually done by comparing the output values of gates, registers, memory elements and so on. In a property checking approach, one compares the output of a property checking run of the golden RTL against the instrumented RTL.

III. Overview of our methodology

Our fault injection methodology relies on architecting faulty properties, and verifying them against a fault-free RTL. This offers several benefits. Firstly, it is using the language of STE for faulty property specification which enables one to express properties easily. Secondly, using properties to express faults, rather than adding more hardware to the RTL, we are able to avoid the state-space explosion problem experienced by others who have used property checking. By keeping the original RTL intact, one can evaluate the effect of faults on the design more faithfully – there remains only one RTL – the one that has to be investigated for faults. A final benefit is that our methodology describes properties about the general architecture of the processor, and does not rely on characterising the faults in the presence of specific program instances as is done in fault injection both by people who have used property checking and those who have used testing.

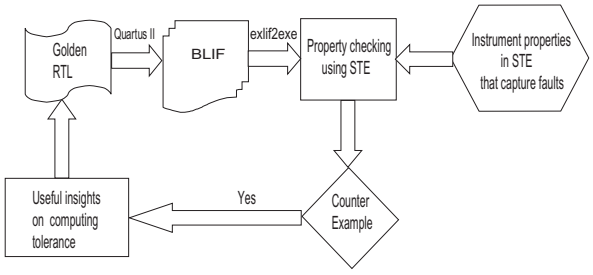

Fig. 2. **Proposed Methodology** Instrument STE properties to express transient faults, and then verify the golden RTL (obtained through a formal verification of a fault-free model) against the faulty properties to obtain a counter-example that provides useful insights on how to obtain tolerance.

We use the STE syntax (shown in the next section) to represent our properties and use the model checking engine built in the model checker Forte [8] to check if the properties are satisfied by the circuit model. These properties can describe a fault-free scenario for the correct functioning of a processor, or a faulty scenario for the same processor design. In fact, the first step in formally analysing the fault-tolerance is to verify if the processor works correctly without the faults. The outcome of the first step is a golden RTL — one that works according to the intended specifications that the designer had in mind for the processor to work correctly in the absence of any fault.

We instrument the STE properties to capture precisely the transient faults we want to model, and then check the modified properties against the golden RTL. This provides us a counter-example, which is very useful for the designer to gain insights into the behaviour of the processor in the presence of faults, and thus a suitable fault tolerance scheme can be designed based on this. An overview of our approach is shown in Figure 2. Our processor is designed in VHDL, and is a 32 bit multi-cycle unpipelined RISC architecture adapted from [9]. The choice of this processor is merely to demonstrate a proof-of-concept. What we show in terms of functionality of this processor is common to any processor - presence of a PC, a branch prediction logic and so on. It is then synthesized (see Figure 2) using the Altera Quartus II tool to a BLIF model (Berkeley Logic Interchange Format) which is then used for model checking using STE. The BLIF model generated from Quartus II is converted to a finite-state machine (FSM) using *exlif2exe* that is provided as a part of the Forte distribution by Intel. An FSM is represented in Forte by a file with a *.exe* extension.

We combine theorem proving, with STE model checking similar to what has been successfully used elsewhere [10] and in major industrial processor verification cycles [8]. The

general strategy[1] is to use STE inference rules [10], to decompose properties about CPU verification into properties about different functional blocks such as *fetch*, *decode*, *execute*, *write-back* and *control*, and check for faults in each block separately. Once properties about each block have been examined independently, the integrated design with all the blocks – the full processor — is re-checked for faults. However, when this is done at the level of the full processor, the symbolic property checking itself does not have to be done again for each and every unit; only the interfaces have to be done. The properties that characterise each faulty block simply become instances of the properties that we had checked earlier for each stand-alone unit, modulo node re-naming. The overall set of properties for the full processor in the presence of faults, are stitched from all the component smaller properties about each functional unit.

IV. STE Model Checking

We provide a brief note on the preliminaries that are required to understand the details presented in later sections. In this section we assume familiarity with logical connectives that appear in propositional logic such as $\neg$, $\wedge$, $\vee$, $\supset$. We use $\forall$ quantifier from first-order logic to express the fact that something holds "for all values", and use the $\vdash$ notation to denote a theorem. We also use the notation $(a \rightarrow b \mid c)$ to express "if a then b else c".

STE [11] is a model checking technique that combines the ideas of *ternary modelling* (using 0,1 and X) with *symbolic simulation* (using symbolic variables). Circuit models are defined on lattice states, and are constructed on-the-fly during simulation, from the FSM (.exe) representation of the circuit. Specifications in STE, take the form of what are known as *symbolic trajectory formulas*. Formally, we define the syntax of formulas [8, 11] as follows:

Definition 1. *Syntax of STE formulas*

$$
\begin{array}{rcll}
f & \triangleq & n \text{ is } 0 & \text{- node } n \text{ has value 0} \\
 & \mid & n \text{ is } 1 & \text{- node } n \text{ has value 1} \\
 & \mid & f_1 \text{ and } f_2 & \text{- conjunction of formulas} \\
 & \mid & f \text{ when } G & \text{- } f \text{ is asserted only when } G \text{ is true} \\
 & \mid & \mathsf{N}f & \text{- } f \text{ holds in the next time step}
\end{array}
$$

where f_1 and f_2 range over formulas, $n \in \mathit{string}$ ranges over the nodes of the circuit, and G is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a *guard*. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent inter-dependencies among node values. For example, we can associate an arbitrary propositional formula G with a node using the construct 'n is G' defined by

$$n \text{ is } G \triangleq ((n \text{ is } 1) \text{ when } G) \text{ and } ((n \text{ is } 0) \text{ when } \neg G)$$

We also use a convenient form of expressing the temporal formula, by using *from* and *to* functions.

$$f \text{ from } i \text{ to } j \triangleq \mathsf{N}^i f \text{ and } \mathsf{N}^{i+1} f \text{ and } \ldots \text{ and } \mathsf{N}^{j-1} f$$

[1]This strategy is used for processor verification in the absence of faults as well.

where the convention is that $\mathsf{N}^0 f = f$. The interested reader is referred to [11].

Verification takes place by testing the validity of an *assertion* or property, of the form $(A \Rightarrow C)$, where both A and C are trajectory formulas. In practice every successful STE run i.e., a run that returns the value True, is a theorem that holds for all the Boolean variables mentioned in the property. However, in those cases where the STE model checking returns a counter-example, it can still be turned into a theorem where the counter-example becomes the assumption under which the STE property is valid.

V. Experimental Results

In this section we shall examine the fault injection in the fetch unit of the processor. The fetch unit as shown in Figure 3. The adder computes the address of the next instruction when there is no branch detected, and the output of the ALU is not zero. In case a branch is detected the address comes from the result of another adder (not shown in the figure) shown as an AddResult, and the multiplexor selects between this value and PC+4.

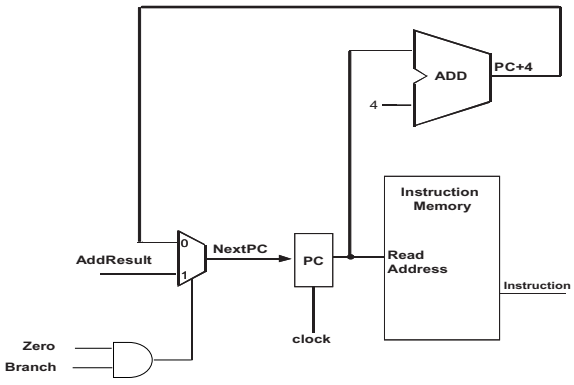

Fig. 3. A simplified fetch unit.

Classes of faults we have looked at are the SEUs affecting

1) a random bit in the PC
2) multiple bits in the PC
3) a random bit in the Instruction Memory
4) multiple bits in the Instruction Memory

We first show that the fetch unit works correctly in the absence of any fault, and then show the effects of faults.

We begin by showing that the program counter works correctly, and then we will show what happens when we inject faults in a random bit, or in multiple bits of the PC.

A. Program Counter Works Correctly

We declare symbolic variables `AddResult`$[7:0]$ to place on the AddResult bus. We assert `AddRes` to denote the vector of new variables `AddResult`$[7:0]$. We assert that the higher six bits of the PC take on symbolic values between time 1 and 2. We define this here to verify this later in case of a branch.

`let NextPC_is_AddResult` = "`PC`$[7:2]$" `is AddRes from 1 to 2;`

We declare the assertions that state that the nodes Branch and Zero take on symbolic values (Branch and Zero respectively), and the bus AddResult takes on symbolic values.

```
let Zero_asserted = "Zero" is Zero from 0 to 10;

let Branch_asserted = "Branch" is Branch from 0 to 10;

let AddResult = "AddResult[7 : 0]" is AddRes from 0 to 10;
```

We declare symbolic values for higher (PC[7 : 2]) and lower bits of the PC ([F, F]), and assign the higher six bits to the predicate PC_initial. We don't need to assert any values to the lower two bits of the PC since we have already clamped them to logic 0 in the RTL (to increment PC by 4).

```
let PC_initial = "PC[7 : 2]" is PC[7 : 2] from 0 to 1;
```

We state the trajectory formula that the next state of the PC is incremented by 4. We use the built-in add function (ADD_int_bvn_fix) on integer vector.

```
let NextPC_is_PC_plus_4 =
   (("PC[7 : 2]" is (ADD_int_bvn_fix 1 PC_MSB) and
      (["PC[1]", "PC[0]"] is [F, F])) from 1 to 2);
```

We declare a clock below.

```
let clock = "clock" is F from 0 to 1 and
            "clock" is T from 1 to 2 and
                        ⋮
            "clock" is T from 9 to 10;
```

We then verify the property that either the PC is incremented by 4 or takes on the branch address.

```
⊢ clock and PC_initial and reset_is_low and AddResult
and Zero_asserted and Branch_asserted
⇒  (NextPC_is_AddResult when (Zero ∧ Branch))
    and
   (NextPC_is_PC_plus_4 when (¬Zero ∨ ¬Branch));
```

The reader should note that we have not quantified all the free Boolean variables in the theorem for the sake of presentation clarity.

B. Fault Injection in the PC

We have developed a library of functions for injecting faults in different units of the CPU and they have been written in the functional language FL. A sample function that injects the fault in the upper six bits of the PC is shown in the property below, it takes the 5-tuple representation of the PC and inverts the symbolic values placed on the PC.

Injecting a bit-flip fault in the higher six bits of the PC, we will get a counter-example BDD — Branch ∧ Zero. We still expect to see that the next value of PC is either the result of branch or a normal PC increment, but we don't anymore because of the fault. This is shown as an assumption of the theorem that represents the outcome of the STE run. What this means is that if we assert the Branch and the node Zero to be a logic 1, the PC will work correctly in spite of the fault. This seems to suggest that in the presence of a branch, the PC is less susceptible to SEUs in the PC.

```
⊢ (Branch ∧ Zero) ⊃
(clock and (map(λ(a, b, c, d, e).(a, b, ¬c, d, e)) PC_initial)
   and reset_is_low and AddResult and Zero_asserted
   and Branch_asserted) ⇒
    ((NextPC_is_AddResult when (Zero ∧ Branch)) and
    (NextPC_is_PC_plus_4 when ((¬Zero) ∨ (¬Branch))));
```

We show the effect of injecting a bit-flip in any single bit of the top six bits of the PC. The outcome of the fault is similar to the one when we inject the fault in all bits of the PC.

```
let PC_fault i = clock and (fault PC_initial i)
  and reset_is_low and AddResult and Zero_asserted
  and Branch_asserted ⇒
    ((NextPC_is_AddResult when (Zero ∧ Branch)) and
    (NextPC_is_PC_plus_4 when ((¬Zero) ∨ (¬Branch))));

⊢  (Branch ∧ Zero)  ⊃ ∀i.(2 ≤ i ≤ 7) ⊃ (PC_fault i)
```

When we inject a fault in bits 1 and 0 of the PC we still expect to see that the next value of PC is either the result of branch or a normal PC increment, but we get the following counter-example: AddResult[1] ∧ Branch ∧ AddResult[0] ∧ Zero. What this suggests is that when an SEU hits only the lower two bits of the PC, the effect of that can be anulled only if the node Zero is logic 1, and there is a Branch instruction, and the lower two bits of the Branch instruction (AddResult) should be both logic 1.

Now we investigate if the PC values successfully get loaded onto the Read Address Bus of the Instruction memory in a fault-free case.

```
let ReadAddress1 =
      "ReadAddress[7 : 0]" is AddRes from 1 to 2;

let ReadAddress2 =
    (("ReadAdress[7 : 2]" is (ADD_int_bvn_fix 1 PC_MSB)) and
    ("ReadAddress[1 : 0]" is [F, F] from 1 to 2));
```

The following property demonstrates that the interface from the PC to the Instruction Memory is sound.

```
⊢ clock and PC_initial and reset_is_low and
   AddResult and Zero_asserted and Branch_asserted
   ⇒ ((ReadAddress1 when (Zero ∧ Branch)) and
       (ReadAddress2 when (¬Zero ∨ ¬Branch)));
```

C. Verifying that instruction memory works correctly

Our Instruction Memory is 256 deep and 32 bits wide. Here we show how we verify that the memory works correctly[2]. We assume we have declared a vector of BDD variables denoted by WD that would be written onto the Write data port of the memory. We then express the fact that the Write port of the memory takes on this vector from time 0 to 1.

```
let WriteData = "WriteData[31 : 0]" is WD from 0 to 1;
```

Similarly we assume the existence of symbolic BDD variables for ReadAddress (RAddd) and WriteAddress (WAdd) ports.

```
let WriteAddress = "WriteAddress[7 : 0]" is WAdd from 0 to 1
let ReadAddress = "ReadAddress[7 : 0]" is RAdd from 1 to 10;
```

[2]The reader should note here that in practice the memory verification runs were carried out using symbolic indexing [8], not quite literally the way they have been shown in this paper for presentation clarity.

We define assertions that state the Memory Write and Read is asserted.

```
let MemWrite_is_asserted = "MemWrite" is we from 0 to 1
                                  and
                           "MemWrite" is F from 1 to 10;

let MemRead_is_asserted = "MemRead" is F from 0 to 1 and
                          "MemRead" is T from 1 to 10;
```

We define 256 distinct constants $\texttt{IMem}_0 \ldots \texttt{IMem}_{255}$ that denote 32-bit names for each of the 256 rows of the memory. Further, we assume that we have declared 256 distinct 32-bit symbolic BDD variables to initialise the memory. We will denote this vector of variables as $\texttt{mem}_0 \ldots \texttt{mem}_{255}$.

Memory is initialised by assigning symbolic values to the names of the memory elements.

```
let Initial_memory =
        IMem0 is mem0 from 0 to 1 and
                      ⋮
        IMem255 is mem255 from 0 to 1;
```

We express that new data (WD) is placed at the Write Data port and writes it to a location selected by the write address (WAdd), which itself ranges from 0 (Zero) to 255 (TwoFiftyFive).

```
let Write_data_to_memory =
        ("WriteData[31 : 0]" is WD from 0 to 1)
            when (we ∧ WAdd = Zero) and
                      ⋮
        ("WriteData[31 : 0]" is WD from 0 to 1)
            when (we ∧ WAdd = TwoFiftyFive);
```

We define the function that constructs the assertion that states that if the write-enabled is asserted high, then the new data is written to the location selected by WAdd, else the old state of the memory ($\texttt{mem}_\texttt{i}$) is retained.

```
let New_state_of_memory =
(IMem0 is (we ∧ (WAdd = Zero) → WD | mem0) from 1 to 2) and
                      ⋮
(IMem255 is (we ∧ (WAdd = TwoFiftyFive) → WD | mem255))
      from 1 to 2
```

We state the property that the Write function to the memory works correctly.

```
⊢ clock and WriteAddress and Write_data_to_memory and
   MemWrite_is_asserted and Initial_memory
                ⇒ New_state_of_memory;
```

We would like to check if the read-after-write operation in the memory works correctly. For this we define a symbolic function below that says that if the read takes place from the same location where a new data (WD) has been written, it should fetch the new data, else the old state of the memory ($\texttt{mem}_\texttt{i}$) is retained.

```
let Read_after_write_fn =
(RAdd = Zero) → ((we ∧ (WAdd = Zero)) → WD | mem0)
| (RAdd = One) → ((we ∧ (WAdd = One)) → WD | mem1)
                      ⋮
| (RAdd = TwoFiftyFive)
   → ((we ∧ (WAdd = TwoFiftyFive)) → WD | mem255)
```

The following simply assigns the symbolic read-after-write function to the Read port of the memory.

```
let ReadData =
      "ReadData[31 : 0]" is Read_after_write_fn from 3 to 5;
```

We now show that the read-after-write property works correctly.

```
⊢ clock and WriteAddress and MemWrite_is_asserted
and WriteData and ReadAddress and MemRead_is_asserted
and Initial_memory ⇒ ReadData
```

D. Fault Injection in Instruction Memory

We have injected several bit-flip faults in the memory to examine the path the fault takes from the source to the output instruction stream coming from the read port of the memory. The faults we will discuss are single-bit errors (only one bit anywhere in the memory is infected), and several bits are infected at once across different parts of the memory. What we have observed is that for a given row if we inject a bit-flip in any of the columns of the memory, the counter-example remains the same. However, from one row to another the counter-examples vary, and this is for a good reason. Our memory is organised as a grid of 256 rows (0 to 255) and 32 columns (0-31 bits). There are two different kinds of faults we will discuss here. First is the case when we inject a fault in an entire row. We will call these row faults. The other kind is when we arbitrarily choose a bit anywhere in the memory and flip its value. We shall refer to this as a bit-fault.

```
//Checking if read after write property is correct
//when we inject an SEU in row 255

((WriteAddress[7] ∧ WriteAddress[6] ∧ WriteAddress[5]
∧ WriteAddress[4] ∧ WriteAddress[3] ∧ WriteAddress[2]
∧ WriteAddress[1] ∧ WriteAddress[0] ∧ we)∨
¬ ReadAddress[7] ∨ ¬ ReadAddress[6] ∨ ¬ ReadAddress[5]
∨ ¬ ReadAddress[4] ∨ ¬ ReadAddress[3] ∨ ¬ ReadAddress[2] ∨
¬ ReadAddress[1] ∨ ¬ ReadAddress[0]) ⊃
⊢ clock and WriteAddress and MemWrite_is_asserted
and WriteData and ReadAddress and MemRead_is_asserted
and Initial_memory and (fault_row Initial_memory 255)
            ⇒ ReadData
```

Let us look at the counter-example in a bit more detail. The counter-example states in a mathematically concise form what a designer would expect the memory to do in the presence of these faults. For example, when we inject the fault in the entire 32 bits of row 255, and check the read-after-write property again in presence of this fault, we can only obtain the correct values of memory during the read-after-write operation, if we do not read from the faulty locations of the memory — which is exactly what the counter-example for this fault states. The counter-example says that as long as we write to the address location indexed by the symbolic values WriteAddress[7 : 0], and all of them take on a logic 1 value, we have written the new data at location 255. If we subsequently read from any other location but location 255, (at least one of the read address bits amongst ReadAddress[7 : 0] take on a logic 0 value), we will be able to assert that the read-after-write works correctly. This is exactly the case with each of the

other memory locations when we fault other rows between 0 and 254. Checking the read-after-write for the fault at any particular bit j in a given row, for each different j between 0 and 31, the counter-example generated for each j is identical, and in fact is the same as the case for the fault in entire row i. The reason is exactly the same as explained above for the fault in each row. We verified 8192 single-bit-flip cases, and it comes as no surprise that the bit-faults are a special case of 256 row faults. The benefit of using symbolic simulation and model checking is that we did not have to conduct $2^{(m \times n)}$ simulations (for m rows and n columns) for analysing single-bit errors in the memory; instead we only did $m \times n$ simulations. Whilst performing a single-bit injection, we could also pick any other arbitrary row to inject a fault, and we only needed to conduct $m \times n$ simulations rather than $2^n \times m$.

VI. Conclusions and Future Work

This paper presented a new methodology of fault injection that used STE model checking. The novelty lies in how the fault injection was carried out using the simple, formal and powerful framework of STE. Not only did this allow us to use a formal property checking methodology that is underpinned by strong mathematical semantics, but also the way we used property checking to perform fault injection, we alleviated the problem of state-space explosion for fault injection faced by Leveugle [2], and Krautz et al. [3]. Again, compared to both Leveugle and Krautz et al. we showed that our language of choice for property specification i.e., STE is able to easily express properties — we therefore minimised the possibility of having complex rules as was the case with PSL in [2, 3]. However, with this ease-of-use comes also the limitation on expressivity. With the restricted synatx of STE, it is not as expressive as PSL. What we have found though, is that for our task, STE is very well suited. Also, it is clear that by keeping the RTL intact, one can evaluate the effect of faults on the design more faithfully – there remains only one RTL – the one that has to be investigated for faults. Whereas both Leveugle and Krautz et al. attempted to formally check a piece of RTL with fault injection hardware embedded in it, our focus has been to assess the architectural vulnerabilities in an unprotected and untarnished RTL in the presence of faults which we created by architecting faulty properties. Another strength of our approach has been that it is completely independent of any applications or benchmark programs, and is a generic method of analysing an architecture. We have identified an automated tool flow to obtain FSMs from RTL which we believe is a useful contribution of our work, since this enables us and others to conduct formal verification and fault injection more easily.

We have shown through our case study how using our approach, one can learn useful aspects about the processor's behaviour in the presence of faults, and these can be used to provide guidelines to architects, for designing a fault-tolerant microprocessor. As part of our ongoing work we are looking to investigate another problem which is to work out the *minimal* bit of redundancy that is required to accomplish the *maximum* fault tolerance for the most *critical* parts of a processor. This entails identifying an optimal set of properties that can reveal the overlap amongst different functional units, during fault simulation. A substantial challenge here is to identify which properties define a complete set for fault diagnosis, and how many properties are adequate, a significant problem that has been recently studied by Fummi et al [12].

VII. Acknowledgement

The authors would like to thank Petru Eles and Zebo Peng at Linköping University for fruitful discussions, and EPSRC (UK) for funding this research under grant EP/D057663/1.

References

[1] S. Hazelhurst and J. Arlat, "Specifying and Verifying Fault-Tolerant Systems," in *Proceedings of DCC'02*, M. Sheeran and T. Melham, Eds., 2002.

[2] R. Leveugle, "A New Approach for Early Dependability Evaluation Based on Formal Property Checking and Controlled Mutations," *IOLTS 2005*, vol. 00, pp. 260–265, 2005.

[3] U. Krautz, M. Pflanz, C. Jacobi, H. W. Tast, K. Weber, and H. T. Vierhaus, "Evaluating coverage of error detection logic for soft errors using formal methods," in *Proceedings of the DATE 2006 conference*, 2006, pp. 176–181.

[4] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems," 1995.

[5] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Exploiting FPGA for Accelerating Fault Injection Experiments," in *IOLTW 2001*. IEEE Comp. Soc., 2001, p. 9.

[6] Mei-Chen Hsueh and Timothy K. Tsai and Ravishankar K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[7] Jeffrey A. Clark and Dhiraj K. Pradhan, "Fault Injection: A Method for Validating Computer-System Dependability," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[8] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification," *IEEE TCAD*, vol. 24, no. 9, pp. 1381–1405, 2005.

[9] James O. Hamblen and Michael Furman, *Rapid Prototyping of Digital Systems: A Tutorial Approach*, 2nd ed. Springer, 2001.

[10] S. Hazelhurst and C.-J. H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs," *IEEE Tran. on CAD of Integrated Circuits*, vol. 14, no. 4, pp. 413–422, 1995.

[11] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Journal of FMSD*, vol. 6, no. 2, pp. 147–189, 1995.

[12] Franco Fummi and Graziano Pravadelli, "Too few or Too Many Properties? Measure it by ATPG!" *Journal of Electronic Testing*, vol. 23, no. 5, Oct 2007.