A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems

Christian Bartsch, Nico Rödel, Carlos Villarraga, Dominik Stoffel, Wolfgang Kunz
Electronic Design Automation Group
University of Kaiserslautern
{bartsch,roedel,villarraga,stoffel,kunz}@eit.uni-kl.de

Abstract. With the advent of new microelectronic fabrication technologies new hardware devices are emerging which suffer from an intrinsically higher susceptibility to faults than previous devices. This leads to a substantially lower degree of reliability and demands further improvements of error detection methods. However, any attempt to cover all errors for all theoretically possible scenarios that a system might be used in can easily lead to excessive costs. Instead, an application-dependent approach should be taken, i.e., strategies for test and error resilience must target only those errors that can actually have an effect in the situations in which the hardware is being used.

In this paper, we propose a method to inject faults into hardware and to *formally* analyze their effects on the software behavior. We describe how this analysis can be implemented based on a recently proposed hardware-dependent software model called *program netlist*. We show how program netlists can be extended to formally model the behavior of a program in the event of one or more hardware faults. First experimental results are presented to demonstrate the feasibility of our approach.

1. Introduction

In the design of Systems-on-Chip and Embedded Systems measures for increasing error resilience can benefit from an application-dependent approach when determining good trade-offs between effectiveness and costs. Only those errors should be targeted that can actually have an effect in the situations in which the hardware is really used. These situations are defined by the software. Fortunately, as a result of the application-specific nature of embedded systems, most parts of the software do not undergo major changes during the system's lifetime. This is true in particular for low-level software components controlling the communication between the application software and the hardware, implementing important functions for chip management and, not rarely, replacing traditionally hardware-implemented control functions of the system. When taking measures for increasing the system's reliability with respect to HW faults, it seems wise to take this SW into account since different HW faults may have different relevance in their effects on this software layer and, thus, on the entire system.

Fault injection is a well-known technique to evaluate the fault tolerance of a component or system against specific faults [3, 7]. While the standard application for fault injection is determining the coverage of implemented resilience techniques, our work has a slightly different focus and uses fault injection to increase the efficiency of resilience techniques by determining which faults should be covered.

For this purpose, a computational model, called *program netlist* [17], recently developed for hardware-dependent software verification, has been extended to formally model the effects of hardware faults on the software. Under all possible runs of the software the effect of an injected fault on the program state is precisely determined, including corner cases which are difficult to find for non-formal approaches like simulation. The proposed method does not only enable the modeling of simple faults like single bit flips but also of more complex fault scenarios consisting of multiple faults occurring at different points in time. Such complex fault scenarios may result from practical observations of test engineers or from a preceding analysis, as described in Section 4.1.

In cases where the protection of specific variables against faults is considered critical and should be increased, we propose to analyze the data dependencies between assembler instructions. In this way, it is possible to understand the possible root causes of faults and the possible fault propagations through the program so that appropriate countermeasures can be taken.

The paper is structured as follows. In Section 2 we briefly discuss already published work related to our topic. Then, we explain in Section 3 how our model is generated and how it is extended to model faulty program behavior. In Section 4 we present the proposed analysis of the modeled effects as well as the analysis of data dependencies. In Section 5 experimental results are presented and discussed. This is followed by a conclusion where we summarize our work and point out further applications.

2. Related Work

Most existing approaches for HW/SW cross-layer fault analysis, such as [9, 11], are based on simulation. It is in the nature of simulation-based approaches that full confidence can never be gained on the absence of errors. For the same reason a complete understanding on how a fault can propagate and how it can affect the program execution is not possible. The proposed formal approach can provide this confidence and an improved understanding of possible fault propagations by exploring all possible program runs under a given fault assumption.

Our approach is therefore also useful to formally certify the effectiveness of resilience measures or to prove that one resilience measure is more effective than another based on some metrics. In [14] related goals are pursued and specific code transformations were proposed to increase the robustness of software against hardware faults. However, the verification in [14] relies on a simulation-based method. Verifying the effectiveness of such approaches in a formal way can increase the confidence about the applied resilience measures and allows for formulating safety guarantees for the system.

There is also previous work in which formal techniques were used to analyze the effects of HW faults on the system behavior. In [5], for example, model checking is used to prove that specific fault tolerance properties hold. A use case of this technique was presented in [4]. In order to perform the fault analysis a labeled transition system (LTS) representing the considered system has to be generated. Model checking was also used in [15], where the fault tolerance of a startup algorithm for a time-triggered architecture had been proven. Like in [4] a manual conversion of the HW/SW system into a highly abstracted state transition system is conducted. Such approaches are promising when a manual translation of the concrete system into a highly abstract model is doable. However, in some industrial settings a higher degree of automation is required. Moreover, the standard fault models for HW implementations like stuck-at faults and bit flips do not have a direct correspondence at the abstract level. For the same reason fault effects on the detailed I/O behavior are difficult to model at the abstract level such that their analysis may be difficult or even completely impossible.

In [10] a formal framework is proposed which is able to analyze the effects of transient HW faults on the program behavior. The proposed framework enumerates the effects of an error for every possible error location, e.g., every register in the register file, at one or more instructions of the program. However, it cannot handle undefined inputs to the program, so that concrete input values have to be chosen. In addition to that the assembly program under consideration has to be converted into a custom-built assembly language supporting only a small set of instructions. Analyzing how a fault affects the temporal behavior of a program is therefore very difficult. The same difficulties occur when trying to derive low-cost resilience measures by exploiting the knowledge about the used instruction set architecture and possible program states. Another drawback is that the framework operates on the word level, leading to over-approximation and incomplete knowledge on the program's control flow. This may lead to complexity problems (e.g., by introducing infinite loops) as well as to false alarms when attempting to certify the effectiveness of resilience measures.

Our approach can be seen as complementary to previous work that has evaluated the effects of HW faults on the architectural processor state, such as [6]. There, it is elaborated on how intermittent HW faults on the RT level affect the behavior of processor components, including program-visible components like the register file. Knowledge about how physical defects propagate through the layers can be used to develop realistic fault models on the architectural level and provide a basis for methods like the one proposed in this work. Another promising approach developing realistic fault models is to derive them from a meta-model [13].

Finally, the output of our analysis can be used as input for techniques that inject faults directly into the software, e.g., [8], to examine their effect in higher software layers than the ones considered here.

3. Program Netlist

The underlying model of the proposed fault analysis method is called *program netlist* (PN) [12]. A PN formally models the behavior of a processor with respect to a specific software program. An important property of this model is that it is of entirely combinational nature so that Boolean reasoning by SAT solving can be employed for its analysis. This was exploited by [17] to develop an efficient equivalence checker that will be used in the proposed fault analysis, as described in Section 4.

3.1. Model Generation

The process for the generation of program netlists is completely automated and consists of two steps. The first step is to unroll a control flow graph (CFG) representing the software program. The CFG can be obtained by extraction from either machine or assembly code of the program. In our model, each node of the graph represents an individual instruction. The unrolled CFG is called *execution graph* (EXG) and is the basis for the second step, where each node of the EXG is replaced by a corresponding logic block describing the behavior of the processor for this particular instruction. Such a logic block is called *instruction cell* (IC). It has an input and an output which are connected to the preceding and succeeding instruction cell, respectively. The input of an IC represents the current program state, i.e., the values of the program variables in memory and the contents of the CPU registers before the corresponding instruction is executed. Its output represents the next program state, i.e., the situation after the instruction was executed.

The CFG used as the starting point for model generation can be incomplete (e.g., branch targets may be unknown because of indirect addressing), as is often the case when a CFG is generated from a real software program. This incompleteness is acceptable because the missing information is generated during the model generation process. This is done by interleaving the unrolling process with a SAT-based analysis to fill in the missing information. The interleaved analysis also supports a compaction of the model [12].

```
ADD(const Rm, const Rn, in PS, out PS')
{
    PS' = PS;
    PS'.RegisterFile[Rn] = PS.RegisterFile[Rn] + PS.RegisterFile[Rm];
}
```

Figure 1: Instruction Cell

An example of an instruction cell template is shown in Fig. 1 by using pseudo-code. (Information about bit widths is abstracted in this and the following examples to make them more readable.) It depicts the behavior of an ADD instruction of the SuperH2 instruction set architecture (ISA). As can be seen, the instruction cell needs to know on which registers the operation should be performed. This information is encoded in the specific assembler instruction of a program. The identifiers Rm and Rn in the template will be replaced with the actual register addresses when the instruction cell is instantiated during PN generation. The instantiated instruction cell has only one input, the current program state, and one output, the next program state. The body of the instruction cell basically contains a forwarding of the program state from the input to the output, with the exception of the register that contains the result of the performed addition. This register is changed according to the ISA specification.

3.2. Fault Description

When analyzing the behavior of software with respect to hardware faults a model of the system is required which describes for each fault the time at which the fault occurs, how long it lasts and how its logical behavior affects the execution of an instruction. The logical behavior of a hardware fault can be modeled by describing its effects on the program state, i.e., how a faulty instruction execution deviates from the correct one. This can be accomplished by changing the corresponding instruction cell description in an appropriate way and is explained in the next section.

In order to model the temporal behavior of a fault, a cycle-abstract representation of time was used in this work. This reflects the need for a time-abstract view on the program execution in order to handle larger processors with unpredictable execution times. Abstraction was performed in the way that time is represented by the order and position of the instructions in the program. The proposed method, however, is easily adaptable for time-accurate instruction cells that can be created for processors with predictable execution times. In [16] a time-accurate version of an instruction cell called *timed instruction cell* was introduced for this purpose.

In the case a single hardware fault has an effect on multiple processor instructions, the effects can be modeled by creating a corresponding fault description for every affected instruction cell. Correct modeling of multiple faults and their effects on different instructions is more challenging. As will be elaborated below, this can be achieved by adding additional constants, registers and ports to the fault description of an instruction cell. (In our modeling of PNs, ports represent the memory interface of I/O instructions [12].)

```
preliminary_ADD_inject_fault_1(const Rn, in PS, out PS')
{
    PS'.Fault_Cond += 1;
    if(PS.Fault_Cond.bit(LSB) == 0)
    {
        PS'.RegisterFile[Rn].bit(MSB) = PS.Fault_Register.bit(MSB);
    }
}
```

Figure 2: Modeling fault injection – preliminary

The example shown in Fig. 2 illustrates the most simple case of a fault description that can be integrated into the description of an instruction cell. In this example, a stuck-at fault is modeled which is activated only every second ADD instruction and which affects the most significant bit (MSB) of the addition. (This may model, e.g., a situation where only one out of two adders in a superscalar pipeline is affected by the stuck-at fault.) For this purpose the architectural state was extended by two registers: Fault_Cond and Fault_Register. The former has an initialization value of zero while the latter is left uninitialized. The Fault_Cond register is incremented every time the ADD instruction is executed, and the fault becomes active whenever the least significant bit (LSB) of the Fault_Cond register is zero, i.e., on every second incrementation of Fault_Cond. Then, the MSB of the target register is assigned the value of the MSB of the unspecified register Fault_register. In effect, the MSB of the target register is treated like an open input in our formal analysis. This way, both faults, stuck-at-0 and stuck-at-1, can be considered at the same time.

Note that it is possible to describe more than one fault for a particular instruction. A fault description with more than one fault can serve two purposes. It can be used to model multiple faults and to examine their combined effect on the program. The second purpose is to model several faults (single or multiple) in the same program netlist, thus avoiding the effort of re-generating the program netlist for every fault to be examined.

In order to support such complex fault descriptions for fault lists with a large number of faults and to separate the activation and deactivation of faults from the computation of the internal processor state, the occurrence of a fault given in the fault list is encoded into the data of an auxiliary memory at a specific location addressed through auxiliary ports. These auxiliary ports do not correspond to variables of the original software but are only used in our computational model to gain better control on the activation conditions.

During fault analysis the values of these ports are set appropriately so that specific faults and combinations of faults can be activated or deactivated. In fact, using this construction, it is sufficient to generate a single PN to analyze the effects of several single faults and/or several multiple faults together with the original fault-free behavior.

```
ADD_inject_fault_1(const Rn, in PS, out PS', Fault_Port Port)
{
    Port.Address = 0xABCD;
    PS'.Fault_Cond += 1;
    if((PS.Fault_Cond.bit(LSB) == 1) && (Port.Data == 1))
    {
        PS'.RegisterFile[Rn].bit(MSB) = PS.Fault_Register.bit(MSB);
    }
}
```

Figure 3: Modeling fault injection – allowing for several faults in a single model

In Fig. 3 the code of Fig. 2 was modified such that a memory access was added to the fault description. Now the fault is active only when also the read data is equal to 1.

3.3. Fault Injection

Faults are injected into instruction cells by inserting their description at the end of the corresponding cell. The injected fault changes the behavior of the original instruction cell by either performing additional changes of the program state or by overwriting changes of the fault-free part.

```
ADD(const Rm, const Rn, in PS, out PS', Fault_Port Port)
{
    PS' = PS;
    PS'.RegisterFile[Rn] = PS.RegisterFile[Rn] + PS.RegisterFile[Rm];

    ADD_inject_fault_1(Rn, PS, PS', Port);
    ADD_inject_fault_2(Rm, Rn, PS, PS', Port);
    ADD_inject_fault_3(Rm, Rn, PS, PS');
    ...
}
```

Figure 4: Instruction cell with fault injection

The example in Fig. 4 shows an instruction cell with several fault injections. Adding both the temporal activation conditions and the descriptions of the logical fault behaviors to the instruction cells has the advantage that during the model generation process no complex fault injection is required. As described in [12], the PN model generation steps are interleaved with a SAT-based analysis to prune the control space of the program. This analysis is now extended to the instruction cells with their fault descriptions so that all possible fault scenarios are also included into the generated model. Based on the obtained PN the faults are injected simply by making value assignments to the addresses of the auxiliary memory.

Note that since the PN represents all fault behaviors of the fault list, it is also possible to perform a global reasoning over all faults or sets of faults. For example, the set of all faults could be determined that lead the program into a specific program state.

Obviously, a PN modeling a large number of possible faults may turn out to be more complex than the corresponding PN for the fault-free case or the PN for only a small subset of these faults. Depending on the complexity of the model it may therefore be advisable to partition the fault list and to analyze each partition in a separate PN.

4. Fault Analysis

The resulting model can be used to analyze the effects of HW errors on the SW behavior including program states, I/O sequences and the control flow.

Note that all registers modeled in the PN are program-visible registers of the design and create a direct link to the gate level. All faults that are modeled in the PN registers are also modeled in the corresponding gate-level registers. This means that insights on the effects of faults modeled in the PN registers also hold for the corresponding faults at the gate level. These insights can also be transferred to HW faults in the gate-level combinational logic by a testability analysis based on these registers. For example, a redundant stuck-at fault at a register bit will cause also other faults in its fanin logic to be untestable. A general analysis by SAT or combinational ATPG that computes the consequences of testability conditions at the program-visible registers for faults in the combinational logic is, however, beyond the scope of this paper.

This work takes HW faults as modeled in the PN as starting points and explores their effects in the SW. Two approaches for fault analysis are possible. The first one is to compare the behavior modeled by the PN against an abstract specification using a HW property checker. The second approach is to compare the PN containing faults with its fault-free counterpart. The latter approach is preferred here since it can benefit from sophisticated optimizations used in standard hardware equivalence checking. A method to check the equivalence of two different PNs was already proposed in [17].

We define two programs to be equivalent iff they produce the same output sequence for any applicable input sequence. Programs with activated faults represent mutations, i.e., new programs different from the original. Equivalence checks can be performed on the same PN but with a different set of faults activated in each check. The PN with all faults deactivated can be used as the fault-free reference.

There are two possible outcomes of the equivalence check. The first possibility is that the PNs are equivalent, i.e., the corresponding programs produce the same I/O sequences. In this case, the considered fault has no effect on the program behavior regardless of what values the inputs have. We denote such faults as *application-redundant*. In the other case, if the PNs are not equivalent, this means that they differ in either data or address of one or more I/O accesses, in the number of I/O accesses, their order or any combination of these. In such cases, a subsequent analysis may be used to categorize the error. For example, a simple structural analysis of the two PNs can yield the information on whether the considered fault affects only data or modifies the control flow of the program.

4.1. Dependency Analysis

In some applications, resilience measures are desirable which do not protect the entire program but only specific functions or instruction sequences inside a function. For example, a loop counter may be considered more critical than some variable within the loop. In order to ensure the correct execution of these critical instructions, however, resilience measures only protecting these particular instructions might not be sufficient. Due to the nature of the program's computation a fault activated during the execution of instructions with low criticality might actually propagate to critical instructions. Therefore, we propose to perform an analysis to determine the data dependencies of critical instructions. This calculation can be done by analyzing the PN and provides a precise description of all dependencies. Note that pure machine code would not be sufficient for this analysis since it yields only incomplete CFGs and therefore would lead to an over-approximation of possible dependencies.

As an example, Fig. 5 shows an excerpt of the results from a dependency analysis. The analysis was performed on the PN of the Traffic Alert and Collision Avoidance System (TCAS) developed by Siemens which is part of the Software-artifact Infrastructure Repository [2]. For demonstration of our analysis, an instruction was selected that delivers the value for a variable which is important for the result calculation of the overall algorithm. The instruction is shown as the bottom node (with address 1432) in Fig. 5.

The figure shows a part of the dependencies existing for the considered instruction. These dependencies are extracted from all reachable program paths leading to this instruction and are represented by a graph as shown. Each node represents an instruction, its address and information on what registers or memory location the instruction reads from (R) or writes to (W). The annotation "`R: @R1 (0x0000)`", for example, indicates that the particular instruction reads from the memory address 0x0000 stored in register R1. Similarly, "`W: R1`" means that the particular instruction writes to R1.

As mentioned before, only an excerpt of the dependency analysis is shown. Parts which were removed are indicated by a dashed line. Solid lines indicate dependencies and are labeled with a type according to Tab. 1. "Type 0" indicates a direct data dependency where one instruction writes to a register which is used by another. "Type 1" also indicates a data dependency but in this case one that exists through a memory value rather than register content. The last type, "Type 2", indicates that the particular instruction depends on a correctly executed jump or branch instruction.

It is worthwhile noting that the uppermost node (with address 1432) represents a jump instruction which needs the value of a register to calculate the jump target address. Due to the characteristics of the used model all possible target addresses are known so that it is possible to trace in the PN both in forward and backward direction to extract the relevant dependencies.

As can be noted, the paths in the dependency graph of Fig. 5 are not numbered with consecutive instruction addresses. In fact, in our experiments it could be observed that the topology of the computed dependency graph is not identical and not even in a simple relationship with the topology of the program's execution graph. This demonstrates that indeed additional information is obtained from the proposed analysis which may be valuable when designing cost-efficient resilience solutions. Their effectiveness can be certified by proving equivalent behavior of the protected code segment for a given fault list.

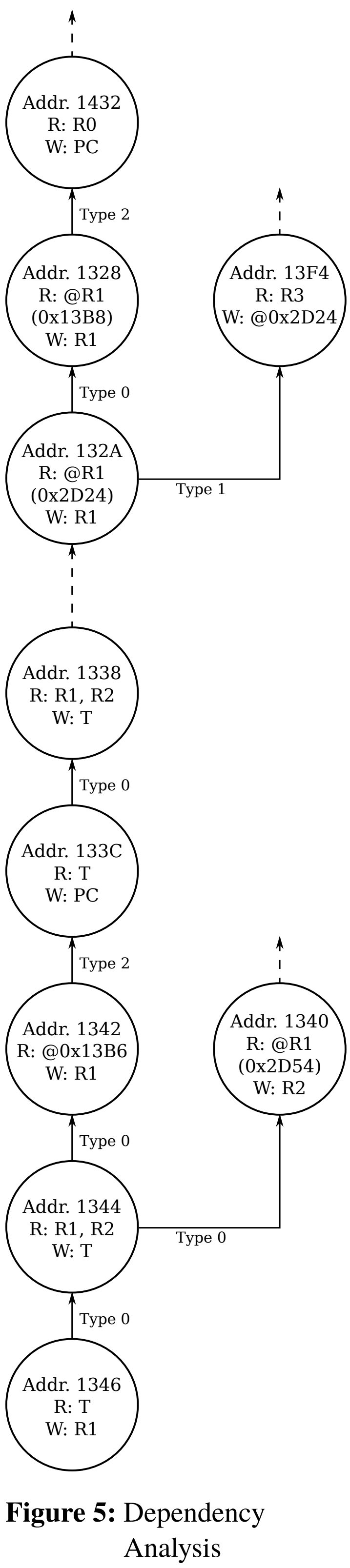

Figure 5: Dependency Analysis

Dependency Type	Explanation
Type 0	Data Dependency (Register)
Type 1	Data Dependency (Memory)
Type 2	Control Flow Dependency

Table 1: Explanation

5. Experimental Results

Experimental results have been conducted on a HW platform containing a 32-bit processor (Aquarius, SuperH2 instruction set) running two different programs. One is a software-implemented industrial driver for a master node of the automotive protocol LIN. The other is the Traffic Alert and Collision Avoidance System (TCAS) [2] mentioned earlier.

An evaluation on how the fault injection affects the model generation runtime and complexity was conducted. For each considered test program two PNs were generated. The first PN, referred to as fault-free, was generated without injecting any faults, while in the second run different faults were injected into the PN. Different types of faults such as stuck-at faults and bit flips have been selected manually from a general fault list and have been injected into the HW.

In the case of TCAS, the safety-critical variable mentioned in Section 4.1 was identified in the source code and the proposed dependency analysis was used to create a list of instructions affecting the variable. The dependency analysis took less than 2 seconds (measured by using the profiling tool gprof) to recursively determine control and data dependencies. For a first assessment of our approach, based on this analysis, four instructions were selected from the generated list. Transient as well as permanent faults were injected.

In the case of LIN, permanent faults were injected in all used shift instructions. Such a fault scenario can be used to model a faulty shifting unit.

It is worthwhile noting that the performed fault injection process was exhaustive. For all bits used in arithmetic and logic operations as well as in read/write processes from/to the register file or from/to the ports all possible faults of types stuck-at-1, stuck-at-0 and bit flip have been injected. In this way, 353 bits were identified in the case of TCAS and 258 in the case of LIN that can be affected by any of these faults. All faults of the different fault types can be analyzed either independently as single faults or in arbitrary combinations as multiple faults. All faults were injected into registers of the PN model and have a direct correspondence to registers at the gate level. In this preliminary experiment, we ignored multiple fault scenarios and restricted our analysis to finding out which of the injected single faults can possibly have an effect on the program behavior. The analysis was done by performing equivalence checks using the commercial tool [1].

Program	Bit Flips	Stuck-at 0	Stuck-at 1
TCAS	353	353	353
LIN	258	258	258

Table 2: Injected Faults

Tab. 2 shows the number and types of faults injected for each test program. The number of injected faults also represents the number of analyzed single bit faults for the particular fault model. This makes a total of 1059 single bit faults for the TCAS example and 774 single bit faults for the LIN example. We used GCC to compile the test programs. All experiments were performed on an Intel i7-4790 CPU at 3.6 GHz with 16 GB RAM. The timing measurements were performed using the profiling tool gprof.

Program	**CPU time (s.)**	
	Fault-Free	**Faults Injected**
TCAS	10.13	22.72
LIN	76.07	120.99

Table 3: CPU times for model generation

Tab. 3 shows the time needed to generate the PNs. It can be observed that fault injection has a significant effect on the runtime of the PN generation process. However, when taking into account that a huge number of different single bit and multiple bit faults are modeled in the PN the increase in runtime seems acceptable.

Program	**# of Instructions**	
	Fault-Free	**Faults Injected**
TCAS	655	660
LIN	1862	2182

Table 4: Number of instructions of models

Tab 4 shows how many instructions each generated PN contains. It can be observed that the fault-injected PN generated for the TCAS program is only 5 instructions or less than 1% larger than the fault-free PN. Obviously, the faults that have been injected, in most cases, do not have an effect on the control flow of the program, although one of the four selected instructions directly affects a branch instruction. We believe that the reason for this is that the program behavior is mostly data-driven such that most branches of the program are already included in the fault-free PN. For the LIN program the fault-injected PN is by 320 instructions (17%) larger than the original version. As a result of injecting the faults the program was able to take program paths which were previously unreachable, adding several new instructions to the PN.

Analyzing the results of the proposed fault analysis can provide important insights into the effects of the considered faults at the software level. For example, in the case of TCAS only 394 injected single bit faults proved to actually have an effect on the value of the selected variable.

Taking into account the entire I/O behavior of the system allowed to check application redundancy for all injected faults. CPU times for conducting the required equivalence checks between the fault-free and the faulty PNs were nearly the same for all faults in both designs. The commercial tool [1] reported CPU time requirements of 0.76 seconds on average in case of TCAS and 3.26 seconds on average in case of LIN, per fault. We were able to prove application redundancy for 561 faults in the case of TCAS and 552 faults in the case of LIN. These fairly large numbers demonstrate the value of the proposed analysis and suggest that the effect of HW faults at the SW level varies widely. Beyond application redundancy also other fault scenarios of interest to the user can be explored. For example, for certain faults of the LIN bus we could observe that the LIN node was virtually disconnected from the bus.

6. Conclusion and Future Work

The paper has demonstrated the feasibility of analyzing the effects of HW faults at the SW level by using formal methods. Formal methods provide the advantage that they can actually *certify* the absence of errors for a given application or the effectiveness of fault resilience measures. This can form the basis for developing test strategies, for example by exploiting the knowledge about application-redundant faults, as well as for designing cost-efficient and effective fault resilience mechanisms both at the HW and the SW level. Such applications of the proposed techniques are subject to our future work.

The scope of the proposed techniques is to analyze safety-critical software components with the size of tens of thousands of lines of C code. Larger software has to be decomposed and each part has to be analyzed individually. This also will be addressed in future work.

References

[1] *OneSpin 360 DV*. http://www.onespin-solutions.com/.

[2] *Software-artifact infrastructure repository*. http://sir.unl.edu. Accessed: 2015-09-01.

[3] Arlat, J., M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell: *Fault injection for dependability validation: a methodology and some applications*. IEEE Transactions on Software Engineering, 16(2):166 – 182, Feb 1990, ISSN 0098-5589.

[4] Bernardeschi, Cinzia, Alessandro Fantechi, and Stefania Gnesi: *Formal validation of the guards inter-consistency mechanism*. In *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 420–430. Springer Berlin Heidelberg, 1999, ISBN 978-3-540-66488-8.

[5] Bernardeschi, Cinzia, Alessandro Fantechi, and Stefania Gnesi: *Model checking fault tolerant systems*. Software Testing, Verification and Reliability, 12(4):251 – 275, 2002.

[6] Gracia-Moran, J., J.C. Baraza-Calvo, D. Gil-Tomas, L.J. Saiz-Adalid, and P.J. Gil-Vicente: *Effects of intermittent faults on the reliability of a reduced instruction set computing (risc) microprocessor*. IEEE Transactions on Reliability, 63(1):144–153, March 2014, ISSN 0018-9529.

[7] Hsueh, Mei Chen, T.K. Tsai, and R.K. Iyer: *Fault injection techniques and tools*. Computer, 30(4):75 – 82, Apr 1997, ISSN 0018-9162.

[8] Larsson, Daniel and Reiner Haehnle: *Symbolic fault injection*. In *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21*, volume 259, pages 85 – 103, 2007.

[9] Li, Man Lap, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou: *Understanding the propagation of hard errors to software and implications for resilient system design.* In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265 – 276, 2008, ISBN 978-1-59593-958-6.

[10] Pattabiraman, K., N.M. Nakka, Z.T. Kalbarczyk, and R.K. Iyer: *Symplfied: Symbolic program-level fault injection and error detection framework.* IEEE Transactions on Computers, 62(11):2292 – 2307, Nov 2013.

[11] Rashid, L., K. Pattabiraman, and S. Gopalakrishnan: *Characterizing the impact of intermittent hardware faults on programs.* IEEE Transactions on Reliability, 64(1):297–310, March 2015, ISSN 0018-9529.

[12] Schmidt, Bernard, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz: *A new formal verification approach for hardware-dependent embedded system software.* IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013), 6:135–145, 2013.

[13] Schwarz, Michael, Moomen Chaari, Bogdan Andrei Tabacaru, and Wolfgang Ecker: *A meta-model-based approach for semantic fault modeling on multiple abstraction levels.* In *Design and Verification Conference and Exhibition Europe*, November 2015.

[14] Sharma, A., J. Sloan, L.F. Wanner, S.H. Elmalaki, M.B. Srivastava, and P. Gupta: *Towards analyzing and improving robustness of software applications to intermittent and permanent faults in hardware.* In *International Conference on Computer Design*, pages 435 – 438, October 2013.

[15] Steiner, W., J. Rushby, M. Sorea, and H. Pfeifer: *Model checking a fault-tolerant startup algorithm: from design exploration to exhaustive fault simulation.* In *International Conference on Dependable Systems and Networks*, pages 189–198, June 2004.

[16] Villarraga, Carlos, Bernard Schmidt, Binghao Bao, Rakesh Raman, Christian Bartsch, Thomas Fehmel, Dominik Stoffel, and Wolfgang Kunz: *Software in a hardware view: New models for HW-dependent software in SoC verification and test (invited paper).* In *Proc. International Test Conference (ITC'14)*, 2014.

[17] Villarraga, Carlos, Bernard Schmidt, Christian Bartsch, Joerg Bormann, Dominik Stoffel, and Wolfgang Kunz: *An equivalence checker for hardware-dependent software.* In *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 119–128, 2013.