Cross-Layer Fault-Space Pruning for Hardware-Assisted Fault Injection

Christian Dietrich, Achim Schmider, Oskar Pusz, Guillermo Payá Vayá, Daniel Lohmann
Leiniz Universität Hannover, Germany
{dietrich,pusz,lohmann}@sra.uni-hannover.de {schmider, guipava}@ims.uni-hannover.de

ABSTRACT

With shrinking structure sizes, soft-error mitigation has become a major challenge in the design and certification of safety-critical embedded systems. Their robustness is quantified by extensive fault-injection campaigns, which on hardware level can nevertheless cover only a tiny part of the fault space.

We suggest Fault-Masking Terms (MATEs) to effectively prune the fault space for gate-level fault injection campaigns by using the (software-induced) hardware state to dynamically cut off benign faults. Our tool applied to an AVR core and a size-optimized MSP430 implementation shows that up to 21 percent of all SEUs on flip-flop level are masked within one clock cycle.

ACM Reference Format:
Christian Dietrich, Achim Schmider, Oskar Pusz, Guillermo Payá Vayá, Daniel Lohmann. 2018. Cross-Layer Fault-Space Pruning for Hardware-Assisted Fault Injection. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196019

1 INTRODUCTION

Due to shrinking transistor sizes and operating voltages, transient hardware faults caused by single event upsets (SEUs) have become an emerging challenge for safety-critical real-time systems [6]. Functional safety standards, such as ISO 26262 or IEC 61508 [14, 15], cope with this problem by demanding explicit measures to assess (and, if necessary, mitigate) the effect of soft errors on safety and robustness. This is commonly done by performing extensive fault injection (FI) experiments on the target system [2, 4] that try to mimic either the physical *causes* for soft errors (by exposing the system to, e.g., heat or radiation [11]) or their *effects* (by changing logic signals) and then observing the system's behavior with respect to its functional specification.

Compared to radiation or heat experiments, FI on the logic level is a lot cheaper to carry out and has the huge advantage of experiment controllability and repeatability: At discrete points in time in the running system, *faults* are injected for discrete logic signals and the system's execution continues from there on until a predefined terminal state or a timeout is reached. If the result is still correct, the fault was *benign*, otherwise it has turned into an *error*, which can lead to an *failure* of the system. By repeating the same experiments with different software-based hardening schemes, their impact on the overall system's robustness can be quantified.

Logic faults can be injected on the pin [16], flipflop [5, 28], or ISA level [12, 24, 25] and it is an open question, which level is "best" to assess a system's robustness: Higher levels provide for much higher fault-space coverage [24], but lower levels are closer to the physics. Some researchers have shown that the injection level of the commonly assumed SEUs can have quite an impact on the results [5, 28], while others have argued that the reported difference is probably less drastic due to calculation issues [23]. Nevertheless, the lower the injection level (e.g., flipflop-level vs. ISA-level), the more precisely we mimic the effects of real SEUs in the hardware.

1.1 Hardware-Assisted Fault Injection

In practice, however, low-level FI faces severe scalability issues. The run-time costs of fault experiments are high, as the injection of faults on the level of logic elements is typically only possible in circuit simulators. Moreover, the fault space is huge, as an SEU could affect every element at every clock cycle.

One way to mitigate the first problem is hardware-assisted fault injection (HAFI) [9, 17], which takes advantage of the advent of high-density FPGA devices to emulate the target logic circuit (i.e., the netlist) including the injection of faults, so basically the complete FI campaign (including the detection of errors) runs on the FPGA. For this purpose, HAFI approaches instrument the target logic circuit to integrate fault injection and monitoring capabilities directly into the FPGA design. This can yield a speedup for fault experiments by three orders of magnitude [19].

However, even with hardware assistance, the fault space is still way too large to inject every element in every cycle. Hence, to increase the effectiveness of FI campaigns, it is of utmost importance to *prune* the fault space by deriving equivalence classes for (possibly) effective faults and cutting off points that beforehand can be proven to be benign [24].

Fault-space pruning (FSP) is often performed offline on a recorded execution trace. However, especially for HAFI-enabled FI platforms, an online fault-list generation and FSP that is integrated into the FPGA, reveals several advantages: (1) If one FI controller distributes the FI campaign over several FPGAs, injection commands can be more coarse grained ("inject(cycle=500)" vs "inject(cycle=500, wire=42)"). (2) If recording a trace is infeasible due to indeterminism or long-running programs, online FSP is still able to avoid ineffective fault injections. (3) If several programs are examined on the same HAFI platform, we continuously benefit from a reduced fault list without paying the cost for offline campaign planing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196019

1.2 About This Paper

We present a new method for online fault-space pruning in HAFI campaigns. In a nutshell, we exploit the (software-induced) execution state and the internal logic of the target element to dynamically determine (and cut-off) injections that will become benign in the further execution. Technically, we analyze the netlist for all possible fault propagations and derive a fault masking term (MATE) for each logic element that describes input combinations that would mask a fault within one cycle. We employ a heuristic over an exemplary execution flow to find and select MATEs that cut-off a high number of injections. In particular, we claim the following contributions:

- We introduce the concept of MATEs, which dynamically prune the fault space on a per-cycle level by examining the current state of the synchronous circuit.
- We present a heuristic to find and select MATEs and provide a prototype implementation as open source.
- We evaluate the concept with two real-world CPU designs and quantify the fault-space reduction for two test programs.

Our results with an AVR 2-stage RISC core and a multi-cycle MSP430 implementation show that up to 21 percent of all faults on flipflop level can be eliminated as benign by MATEs, which themselves will have only a moderate impact on FPGA resources.

2 SYSTEM MODEL

Our fault model is SEUs in synchronous circuits with internal state, employed in special-purpose (deeply embedded) HW–SW systems. Without loss on generality, we focus on the CPU part. An SEU is assumed to manifest as a state change of any flipflop at any point in time, so the underlying fault space is (flipflops × cycles). Figure 1b depicts our example fault space, which spans over 5 flipflops and 8 clock cycles. We mimic these faults with HAFI for a concrete execution trace in an FPGA-based emulation of the circuit.

Our goal for this paper is to prune the space of possibly effective faults in this setting. A fault is considered as possibly effective, if it could eventually propagate to externally visible state (i.e., visible on an output pin or by the software on ISA level). Otherwise (i.e., if it gets masked by the current state within one clock cycle), it is surely considered as benign.

Note that our definition of benign faults is sufficient, but not complete: A possibly effective fault might still be masked and never propagated to the higher level (e.g., ISA) and, thus, might never yield an error with respect to the system's specification. However, a fault that is benign already on the logic level could never lead to an error on the system level.

3 FAULT-MASKING TERMS

When a transient hardware fault occurs in a logical circuit, it can propagate to a large number of elements and, therefore, lead to a cascade of faulty secondary signals and erroneous behavior. In a synchronous circuit this propagation happens in a time-discrete manner: Faulty signals influence gate outputs, which are input to other gates, and more and more wires become faulty down the road. If such a faulty signal reaches a flipflop, it is stored at the end of the current clock cycle and then has the possibility to poison the circuit behavior in the next time interval. However, not every primary fault comes that far and reaches a flipflop within the first clock cycle after the fault. Some are masked right away by the combination of the current (unfaulty) state and the circuit's structure. We want to detect situations, where a fault is masked within one clock cycle to avoid its injection in larger testing campaigns.

More formally, the logic of a synchronous circuit is a boolean function N that takes a vector $\vec{i}$ of external inputs and the current state (flipflops) and calculates a new state and some external outputs. A fault function f becomes benign within one clock cycle, if the faulty input vector $f(\vec{i})$ results in an unchanged output vector:

$$N(f(\vec{i})) = N(\vec{i}) \rightarrow f \text{ is benign for } \vec{i}$$

Within the physical realization of N, a fault propagates only to a certain area of gates and wires, the *fault cone*. Within this fault cone, we have to distrust all signals, until proven otherwise, as they are potentially faulty. However, there are signals that cross the fault-cone border from the outside and bring new hope to the fault cone as only they can mask the fault. If the values of these *border wires* are appropriate, the fault is masked on *all* paths from the faulty input to the outputs and the fault becomes benign.

Definition (Fault-masking term, MATE). *A fault-masking term M_f for a fault function f is a boolean function of primary inputs and intermediate signals of a circuit. If it becomes true, the fault is surely benign and $N(f(\vec{i})) = N(\vec{i})$. Otherwise, we have no knowledge about the effect of f in this cycle.*

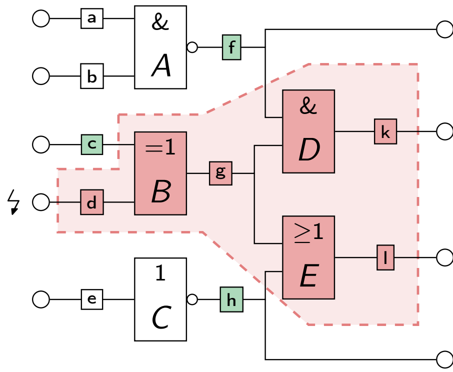
Let us illustrate this concept with an example (Figure 1a): The fault cone for the input d contains the wires {d, g, k, l} and the gates {B, D, E}. There are two paths ($[B, D]$, $[B, E]$) for the fault to propagate from the primary input d to one of output wires {k, l}. The fault-cone has three border wires {c, f, h}, each influencing a single cone gate. As B is an XOR gate, it has no *fault-masking capabilities* and a fault will always propagate through B, regardless of the value of c. However, both D (an AND gate) and E (an OR gate) have a fault-masking capability if at least one input is known to be unfaulty. Therefore, a fault in d is masked within one cycle if (f, h) have the values $(0, 1)$. So one MATE M_d is $(\neg f \wedge h)$; another one is $(a \wedge b \wedge \neg e)$. For the input e, there exists no MATE, as there is a path ($[C]$) where no gate has any fault-masking capabilities.

Figure 1b depicts how a MATE set for the circuit from Figure 1a can be used to prune the fault space. Starting from an offline-recorded execution trace or directly at run time in a HAFI platform, the MATEs are connected to the input and internal wires of the circuit. On a per-cycle basis, all MATEs are evaluated and the benign faults are removed from the fault space. For example, in the first two cycles, the MATEs $\neg b$ and $\neg a$ trigger and, therefore, the wires a and b are removed form the fault space.

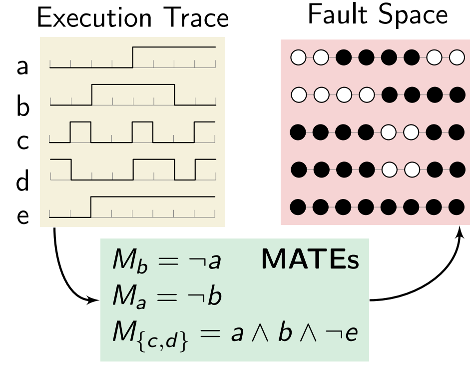
A MATE for a fault can be formulated over any wire that is not within the fault cone. However, as every MATE can be reformulated to contain only border wires as terms, we will concentrate on these *border* MATEs. Border MATEs reuse parts of the already existing circuit and can, therefore, be integrated into a HAFI platform with little effort.

4 HEURISTIC TO FIND HIGH-IMPACT MATEs

Formulating a MATE $M_{f(i)}$ for an input wire i is easy as we could simply duplicate the fault cone, feed it with the unfaulty value of i



(a) Fault Cone for input wire d. If a fault occurs at d, it can propagate within the entire fault cone (d,g,k,l) and the output wires (k, l) is potentially faulty. However, the fault can be stopped from propagating, when *border* wires (c,f,h) have the values f=0, h=1.



(b) Fault-space pruning with fault masking terms (MATEs). The current state of the circuit is fed into a set of MATEs to detect whether a fault could lead to an error (filled dot) or becomes benign within one clock cycle (empty dot).

Figure 1: Fault cone for a single faulty wire and fault-space pruning capabilities for the respective MATEs of the whole circuit.

and compare the results of all output wires. Although this MATE is the most precise one and detects all situations where a fault in i becomes benign, it is inefficient to implement this strategy in hardware for every input. Therefore, we look heuristically at the whole circuit to find several smaller MATEs that detect as many faulty-but-benign wires as possible. With this whole-circuit view, we also find MATEs that detect more than one benign fault.

The starting point for our heuristic search is the netlist of a synthesized circuit and the used gate library. As a first step, we analyze all gates within the library for their fault-masking capabilities. For every gate type, we iterate over all combinations of faulty input wires and find all input-pin assignments that will mask the current faulty-input set. For example, for a 1-bit multiplexer $MUX(x, a, b)$ and the faulty-input set {x}, we will find the gate-masking terms $GM(MUX, \{x\}) = \{(\neg a \wedge \neg b), (a \wedge b)\}$. Phrased differently: If the select signal of a multiplexer is faulty, the fault is stopped at this gate given that the selected inputs are not faulty but equal.

In the second step, we iterate over every wire that can be faulty according to the fault model. For each possibly-faulty wire, we search for MATEs independently. We enumerate all paths of gates in the fault cone up to a given depth (heuristic parameter). Enumerating longer fault-propagation paths enlarges the search space, but also unveils more gates that can potentially stop the fault. However, masking a fault early in the fault cone is easier, as fewer wires are potentially affected by the fault.

For the gates covered by the enumerated paths, we collect all gate-masking terms, while we assume that every wire within the fault cone is to be mistrusted. If there is a path where no gate can mask the fault, we abort the search early and proceed to the next faulty wire. Otherwise, we generate all combinations of gate-masking terms as MATE *candidates* up to a given number of terms (heuristic parameter) and check their conjunction against the propagation paths. If a MATE candidate masks the fault on every propagation path, it is an actual MATE for the investigated faulty wire.

As a third step, we collect and summarize all MATEs for all possibly-faulty wires. Oftentimes, one active MATE indicates the masking of more than one fault. For example, most CPU execution stages have an operation where they select only one operand to implement the `mov`/`ld` instruction. Therefore, a MATE that triggers on this operation can mark all faults in the other operand as benign.

At this point, we have a large set of actual MATEs. However, we do not know whether these MATEs can trigger at all or whether they trigger only in very specific and rarely seen situations. Therefore, we select a subset of all discovered MATEs by replaying an exemplary trace of the circuit in action. For every cycle, we calculate which MATEs would trigger. Beginning from the MATE that masks the most faults, we increase a hit counter for every MATE that masks an additional input wire. After replaying the trace, we select the top-N MATEs with the highest hit counter.

5 EVALUATION

For the evaluation, we apply the heuristic MATE search for two fully synthesized processors with different instruction-level architectures, namely an 8-bit RISC AVR/Atmel-compatible microcontroller, implementing a two-stage pipeline design, and a 16-bit multi-cycle MSP430-compatible microcontroller. On both processors, we simulate two test programs and record a wire-level trace of all internal and external signals. With the calculated MATE sets and the recorded trace, we quantify what proportion of the potential fault-space is surely benign.

5.1 Experimental Setup

First, we performed an ASIC synthesis for both processors using Synopsys Design Compiler 2017.09-SP1. For that, the freely available 15nm FinFET-based Open Cell Library [18] was chosen, optimizing both designs for area. Then, two test programs (i.e., a Fibonacci sequence computation and a convolution function), which use different instruction subsets, were implemented for both processors. Finally, both test programs were executed on both processors by means of a netlist simulation. During the simulation, we recorded a VCD (value change dump) trace file for each program/processor that describes the values of all wires for every clock cycle.

We implemented the MATE search as a Python program and executed the prototype with the optimizing just-in-time compiler

Table 1: Statistic for the heuristic MATE search.

	AVR		MSP430	
	FF	FF w/o RF.	FF	FF w/o RF
Faulty Wires	383	135	743	519
Avg. Cone [#gates]	656	840	287	151
Med. Cone [#gates]	547	581	236	27
Run Time [s]	164	34	126	90
#Unmaskable	81	57	96	70
#MATE candid.	$3 \cdot 10^7$	$7 \cdot 10^6$	$4 \cdot 10^7$	$2 \cdot 10^7$
#MATE	24 536	3226	19 180	17 649

PyPy 5.9.0. The search was executed on a Intel i7-6600 @ 2.60 Ghz (2 cores, 2 hardware threads) machine. We utilized the parallel processing capabilities of the machine, by running the MATE search for different faulty flipflops in parallel.

The input for the MATE search are the generated netlists and the used standard cell library (i.e., the logical function of each gate). For the following MATE selection and the quantification of the fault-space reduction, the VCD trace files were used. Furthermore, we executed the MATE search twice for every processor model: 1. for all flipflops in the netlist ("FF"). 2. for all flipflops in the netlist that are not located in the register file ("FF w/o RF"). We added the second set of faulty wires, since most CPU operations write only a single register and, therefore, most faults in the register file will naturally live longer than one clock cycle. These register-level faults are more likely to be pruned on an inter-cycle pruning strategy.

5.2 MATE Search Performance

Table 1 characterizes the four input sets (2 processors, 2 FF sets) and the performance of our MATE search. For the small AVR core, most flipflops are located in the register file as it contains 31 8-bit registers. Only 135 flipflops are used in the pipeline logic. Since the multi-cyle MSP430 holds more state between cycles, the impact of the register file (14 registers a 16 bit) is much smaller. However, the average (and median) fault-cone for both processors is quite different. Although the MSP430 is the more powerful CPU, the multicycle implementation results in fault cones with about half as many gates.

We ran the MATE search with the following heuristic parameters: (1) Search 8 gates deep into the fault cone to enumerate the fault-propagation paths. (2) Use at most 4 gates to stop the propagation of a fault. (3) Abort the search after trying at most 100 000 MATE candidates per faulty wire. With these parameters, it always took less than 3 minutes to calculate the MATE sets used in the rest of the evaluation. During all search operations, the RAM allocation remained below 1 GiB.

5.3 Fault-Space Reduction

As a first step to quantify the impact of our (complete) MATE sets, we replayed the wire-level traces of both test programs and recorded, for every cycle, which MATE would have triggered and what faulty wires would be detected as benign. The selection of high-impact MATEs is done and evaluated in a second step.

Table 2: AVR MATE Performance. We selected the top N MATEs according to one test program and calculated the percentage of the full fault space was detected as benign by the MATE set. Both programs ran for 8500 clock cycles.

	AVR MATEs	fib()		conv()	
		FF	FF w/o RF	FF	FF w/o RF
	#Effective MATEs	279	209	390	247
	Avg. #inputs	5.4± 1.6	4.9± 1.2	5.8± 1.8	4.9± 1.2
	Masked Faults	7.15 %	14.37 %	7.90 %	16.48 %
selected for fib	Top 10	2.61 %	7.32 %	2.35 %	6.04 %
selected for fib	Top 50	5.79 %	13.77 %	5.30 %	14.23 %
selected for fib	Top 100	7.06 %	14.29 %	7.17 %	14.68 %
selected for fib	Top 200	7.15 %	14.37 %	7.48 %	15.39 %
selected for conv	Top 10	2.27 %	5.93 %	2.58 %	7.05 %
selected for conv	Top 50	4.62 %	13.86 %	5.90 %	15.86 %
selected for conv	Top 100	6.84 %	14.12 %	7.79 %	16.43 %
selected for conv	Top 200	7.04 %	14.22 %	7.89 %	16.48 %

Table 3: MSP430 MATE Performance. We selected the top N MATEs according to one test program and calculated the percentage of the full fault space was detected as benign by the MATE set. Both programs ran for 8500 clock cycles.

	MSP430 MATEs	fib()		conv()	
		FF	FF w/o RF	FF	FF w/o RF
	#Effective MATEs	387	386	441	437
	Avg. #inputs	3.2± 1.8	3.2± 1.8	3.4± 1.9	3.4± 1.9
	Masked Faults	14.63 %	20.91 %	14.32 %	20.45 %
selected for fib	Top 10	4.97 %	7.12 %	4.97 %	7.11 %
selected for fib	Top 50	13.37 %	19.14 %	13.28 %	19.01 %
selected for fib	Top 100	14.28 %	20.42 %	13.95 %	19.93 %
selected for fib	Top 200	14.63 %	20.91 %	14.30 %	20.43 %
selected for conv	Top 10	4.97 %	7.11 %	4.97 %	7.11 %
selected for conv	Top 50	13.19 %	18.88 %	13.11 %	18.77 %
selected for conv	Top 100	14.18 %	20.26 %	14.01 %	20.02 %
selected for conv	Top 200	14.61 %	20.88 %	14.32 %	20.44 %

The first section of Table 2 and Table 3 gives an overview of the complete MATE set on both traces. For the complete AVR and the Fibonacci (fib()) program, 279 MATEs triggered at all, and we could reduce the fault space by 7.15 percent. Without considering the register file, and although the number of effective MATEs reduces, the fault-space shrinkage rises up to 14.37 percent. For the convolution (conv()), the results were slightly better and significantly more MATEs (+39.78 %) triggered at least once. For the MSP430 (see Table 3), the general trends for the fault-space reduction are similar, although the overall performance of our approach is much better, and we achieve at least 14 percent of fault-space reduction. For both programs the results are similar to the AVR results, while more MATEs are triggered by the execution trace. However, the fault-space reduction for conv() is about the same size as for fib(). From the average number of MATE inputs, we can conclude that the hardware overhead to implement a single MATE is negligible

and, with less than 6 input signals (on average), also friendly for an FPGA implementation.

As the discovered MATE sets are quite large (at least 3226 elements), we use the described method (see Section 4) to subset the high-impact MATEs. By replaying the execution traces of one test program, we rate the MATEs for their impact on the fault space and selected a subset of the most efficient MATEs. For the evaluation, we took the simulation trace for fib() (respectively conv()) and generated a top-n set. This subset was then evaluated against both traces to investigate: (a) How well does a small subset perform? (b) How well are MATE sets transferable between different hardware-usage patterns? The second and third section of Table 2 and Table 3 contain these results, as well as the results for the cross validation. For example, the 50 most effective MATEs for all flip flops selected for the fib() trace, reduce the fib() fault space by 13.37 percent. Applied to the conv() trace, the fault space is pruned by 13.28 percent.

As a general trend, both for AVR and MSP430, we see that already 50 MATEs exhibit a fault-space reduction that is very close to the results of the complete MATE set. For MSP430, we see that the achieved fault-space reduction is not significantly dependent on the trace that was used in the selection process. However, the experiments where selection and evaluation trace differ are slightly worse for the AVR core. We consider this as an indicator that MATE selection remains (mostly) stable, even in case the software is slightly changed and the MATE-enriched HAFI platform can be used for several different programs.

6 DISCUSSION

6.1 Integration of MATEs into HAFI Platforms

As our experiments have shown the average MATE-complexity is rather small and, thus, well suited for synthesis into FPGA LUTs. With their average input size of less than 6 wires, one MATE fits into one or two LUTs. Compared to the size of current HAFI FPGA-based platforms, which utilize between 1500 and 6000 LUTs [9, 19] only for the fault-injection control unit, or the capacity of midrange Virtex-6 FPGA (XC6VLX240T, 150k LUTs), the extra LUTs required by 50 to 100 MATEs are negligible.

6.2 Higher Fault Models

Our fault model is based in the common assumption of single, uncorrelated faults caused by SEUs that each induce a logic flip in a single element (Section 2) — an assumption that is frequently criticized, but still considered valid [26]. However, despite the requirement that at most one input is faulty, MATEs do actually not rely on any further assumption about the fault characteristics: Our approach works out of the box also with upsets that hold more than one cycle or cause an input to become logically unstable. Conceptually, also 2-bit faults (or more) could be considered in the construction of MATEs – as well as MATEs for faults that are masked only within more than one clock cycle. Depending on the gate types, such multi-bit/multi-clock MATEs will become more expensive on the FPGA, but might also have a much higher impact on fault space pruning. This underlines the necessity of efficient heuristics to select high-impact/low-cost MATEs and is a topic of further research.

6.3 Extension to ISA Level

The fact that currently the derived MATEs only address fault-masking within a single clock cycle means that faults in flipflops not overwritten in the next cycle could never be masked. Hence, MATEs are very effective if applied to, for instance, stage buffers or the status register, but mask only few faults in the general register file, as on most architectures at most one register is overwritten per clock cycle. This is also observable in the numbers from Table 2 and Table 3, respectively: The multi-cycle architecture of the MSP430 implementation seems to be more beneficial in this respect than the RISC-style architecture of the AVR. However, independently of the architecture, the number of faults masked within one clock cycle is considerably higher if we exclude the register-file flipflops from HAFI. We consider the numbers excluding the register file as more relevant, as faults in general-purpose registers (or even external memory) affect ISA-visible state, so they are on the border of our system model (Section 2). At this stage, ISA-level software-based pruning techniques could take over, which have proven as a very effective as means for FI experiments that target bit flips in registers or volatile memory [12, 25] and achieve full coverage of the fault-space of single-bit flips in typical embedded control systems [24]. Hence, with respect to the question which level is "best" to get realistic results when assessing a system's robustness against SEUs [5, 28], we envision the combination of HAFI on flipflop level with software-based FI taking over at ISA level as the ideal combination.

7 RELATED WORK

Fault-space pruning on the hardware level is done with two different goals in mind. First of all, chip manufacturers want to reduce the number of test patterns that is required to asses the functional correctness of all gates on an actual chip die. Fault collapsing is a technique to statically analyze a netlist for possible faults that are equivalent in their error behavior. Several techniques [1, 10, 20] were proposed to find such equivalence classes, including approaches that utilize specialized binary decision diagrams [27] and the hierarchical nature of hardware designs [21]. However, fault collapsing does not take the current state during a fault-injection campaign into account and is targeted at stuck-at faults. Furthermore, the combination of MATEs and fault collapsing could be profitable when all wires are subject to injection.

Software reliability assessment is the second large area, where fault-space pruning is employed to make large injection campaigns feasible. Here, the fault space is spanned by the program-under-test as it is executed on the hardware. Due to the huge fault spaces, sampling heuristics are used to concentrate on the most important faults [8]. Grouping faults into classes with similar error behavior is done by means of data-structure dependencies [7], address bounds [22], and memory states [13]. However, these pruning techniques are more coarse grained as they span multiple execution cycles. Furthermore, as we already discussed, such inter-cycle pruning techniques complement the presented intra-cycle MATE approach.

Similar to our approach, Asadi et al. [3] consider the error propagation of single gates in a circuit when used with a concrete software. From an execution trace, they calculate a propagation probability to estimate the chance of a whole-system failure. In contrast

to our approach, they do not focus on fault injection or the fault propagation on a per-cycle basis.

Finally, one way to accelerate fault injection campaigns is the use of HAFI tools, which basically instrument the netlist of the SUT to allow fault injection while emulating the SUT on an FPGA platform. An extra control unit is required, which can access in parallel or serially the gates of the instrumented netlist to inject single or multiple faults in the desired cycle and to read the state for subsequent evaluation. The functionality of the control unit can easily be implemented by means of a soft-core processor [17] or a dedicated finite state machine [9]. Moreover, information about the physical location of the gates extracted from the layout of the SUT can be used to emulate realistic multi-event upsets (MUE) [19].

8 CONCLUSION

Experimental fault injection (FI) is a broadly accepted means to assess the functional correctness of safety-critical embedded systems with respect to transient errors. With hardware-assisted fault injection (HAFI) the throughput of gate-level FI campaigns is drastically increased by injecting the faults online during the FPGA-based emulation of the circuit's netlist. However, even with HAFI techniques, the fault space is way too large to achieve full coverage, while on the other side a significant number of faults get logically masked (and, thus, benign) in the real execution.

We introduced fault masking terms (MATEs) as an additional means to prune the fault space in HAFI campaigns. MATEs exploit the software-induced flipflop state to detect faulty inputs that, however, would not propagate to the next clock cycle. The MATEs for all logic elements of the circuit are derived offline from the netlist; a heuristic selects a subset to be integrated into the FPGA-based emulation. Our evaluation results with two test programs on two embedded CPU architectures show that already a small number of 50 MATEs with very moderate hardware costs is sufficient to cut off nearly 20 percent of all flipflop-level faults as benign, which leads to a significant reduction in the number of FI experiments to carry out.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and Horst Schirmeier for their feedback. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/4-1.

The source code of our tool, the used CPU netlists, and all raw data used for this paper are available under an open-source license at:
`https://scm.sra.uni-hannover.de/source/dac-pruning-data/`

REFERENCES

[1] Vishwani D Agrawal, AVSS Prasad, and Madhusudan V Atre. 2003. Fault collapsing via functional dominance. In *ITC*.

[2] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. 1990. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. on Software Engineering* 16, 2 (1990). https://doi.org/10.1109/32.44380

[3] Ghazanfar Asadi and Mehdi Baradaran Tahoori. 2005. An analytical approach for soft error rate estimation in digital circuits. In *ISCAS 2005*. IEEE.

[4] Alfredo Benso and Paolo Ernesto Prinetto. 2003. *Fault injection techniques and tools for embedded systems reliability evaluation*. Kluwer Academic Publishers.

[5] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *DAC '13*.

[6] C. Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *Micro, IEEE* 23, 4 (2003). https://doi.org/10.1109/MM.2003.1225959

[7] Mojtaba Ebrahimi, Mohammad Hadi Moshrefpour, Mohammad Saber Golanbari, and Mehdi B Tahoori. 2016. Fault injection acceleration by simultaneous injection of non-interacting faults. In *DAC '16*. ACM.

[8] Mojtaba Ebrahimi, Nour Sayed, Maryam Rashvand, and Mehdi B Tahoori. 2015. Fault injection acceleration by architectural importance sampling. In *10th Intl. Conf. on Hardware/Software Codesign and System Synthesis*. IEEE.

[9] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil. 2012. Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection. *IEEE Trans. on Computers* 61, 3 (2012). https://doi.org/10.1109/TC.2010.262

[10] Tigranuhi Grigoryan, Heghineh Malkhasyan, Gevorg Mushyan, and Valery Vardanian. 2015. Fault collapsing for digital circuits based on relations between stuck-at faults. In *Computer Science and Information Technologies (CSIT)*. IEEE.

[11] Ulf Gunneflo, Johan Karlsson, and Jan Torin. 1989. Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In *19th Intl. Symp. on Fault-Tolerant Computing*. IEEE. https://doi.org/10.1109/FTCS.1989.105590

[12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *ASPLOS '12*. ACM Press. https://doi.org/10.1145/2150976.2150990

[13] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, Vol. 47. ACM.

[14] IEC. 1998. *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. Intl. Electrotechnical Commission.

[15] ISO 26262-9. 2011. *ISO 26262-9:2011: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*. Intl. Organization for Standardization.

[16] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. 1994. RIFLE: A general purpose pin-level fault injector. In *1st European Dependable Computing Conf. (EDCC)*. Springer-Verlag. https://doi.org/10.1007/3-540-58426-9_132

[17] W. Mansour and R. Velazco. 2013. An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs. *IEEE Trans. on Nuclear Science* 60, 4 (2013). https://doi.org/10.1109/TNS.2013.2267097

[18] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell Library in 15Nm FreePDK Technology. In *Intl. Symp. on Physical Design (ISPD '15)*. ACM. https://doi.org/10.1145/2717764.2717783

[19] Rochus Nowosielski, Lukas Gerlach, Stephan Bieband, Guillermo Payá-Vayá, and Holger Blume. 2015. FLINT: Layout-oriented FPGA-based Methodology for Fault Tolerant ASIC Design. In *Design, Automation & Test in Europe Conf. (DATE '15)*. EDA Consortium.

[20] AVSS Prasad, Vishwani D Agrawal, and Madhusudan V Atre. 2002. A new algorithm for global fault collapsing into equivalence and dominance sets. In *Intl. Test Conf.* IEEE.

[21] Raja K. K. R. Sandireddy and Vishwani D. Agrawal. 2007. Using Hierarchy in Design Automation: The Fault Collapsing Problem. In *11th VLSI Design and Test Symp.*

[22] Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. 2017. Light-Weight Techniques for Improving the Controllability and Efficiency of ISA-Level Fault Injection Tools. In *Pacific Rim Intl. Symp. on Dependable Computing (PRDC)*. IEEE.

[23] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors. In *45th Dependable Systems and Networks (DSN)*. IEEE.

[24] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *11th European Dependable Computing Conf. (EDCC)*.

[25] D. Skarin, R. Barbosa, and J. Karlsson. 2010. GOOFI-2: A tool for experimental dependability assessment. In *39th Dependable Systems and Networks (DSN)*. IEEE. https://doi.org/10.1109/DSN.2010.5544265

[26] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. 2013. Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM Press, Article 22. https://doi.org/10.1145/2503210.2503257

[27] Raimund Ubar, Lembit Jürimägi, Elmet Orasson, and Jaan Raik. 2015. Scalable algorithm for structural fault collapsing in digital circuits. In *Very Large Scale Integration (VLSI-SoC)*. IEEE.

[28] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *44th Dependable Systems and Networks (DSN)*. IEEE. https://doi.org/10.1109/DSN.2014.2