An Exploration of Microprocessor Self-Test Optimisation Based On Safe Faults

Anuraag Narang[1], Balaji Venu[2], Saqib Khursheed[1], and Peter Harrod[2]

[1]Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, United Kingdom
[2]Arm Ltd., Cambridge, United Kingdom

***Abstract*—Microprocessor software test libraries (STLs) must provide maximum fault coverage with minimum overhead. Pruning safe faults, which cannot cause errors in the output of the processor, from the fault list can increase fault coverage without adding test overhead. Applying more application-specific constraints can lead to the identification of more safe faults, and some such constraints are yet to be explored.**

This work explores the use of signal combination-based constraints alongside well-known constant signal-based constraints for identifying safe faults. Also, for the first time, information on safe faults is utilised during test compaction in order to further minimise test overhead. Results for an OpenRISC processor design show up to 2.33% improvement in fault coverage with the use of the proposed constraints. In one test program, a code segment contributing only to the coverage of safe faults is identified, with its removal providing a 1.09% code size reduction on top of existing compaction techniques. The results may vary for a larger and more complex commercial design with greater scope for redundant logic.

***Index Terms*—software-based self-test, software test library, test quality, test compaction**

I. Introduction

In the safety-critical domain, functional safety standards for electronic systems impose major technical and economic challenges on engineers, who look to meet safety goals without compromising the functionality and profitability of their product. This motivates research into the optimisation of safety mechanisms with respect to the aforementioned parameters.

A. STL Optimisation

With microprocessor-based embedded systems increasingly used in safety-critical applications such as advanced driver-assistance systems (ADAS) and autonomous vehicles, the optimisation of safety mechanisms that are used in these systems, such as software-based self-test (SBST), is of great importance. The fault coverage or test quality (ratio of faults covered by a test to the total faults targeted by the test) of the software test library (STL) deployed as part of an SBST mechanism must comply with the targeted standard. Also, the STL must not violate the in-field resource constraints such as memory footprint, test application time (TAT) and power consumption.

Instead of writing additional test sequences and potentially violating resource constraints, fault coverage requirements may be met by classifying faults not covered by the STL in a way such that they can be discounted from the fault list.

To further improve resource utilisation, a test program may be compacted by removing redundant code, i.e., code that does not contribute to the coverage of faults and thus unnecessarily increases memory footprint and TAT. Additional compaction may be achieved by considering the faults that have been classified and discounted from the target fault list, but this opportunity has not been explored in previous works.

B. Safe Faults

Safe faults, named as such due to their inability to cause errors in the output of the design-under-test (DUT), have been explored in various forms for several years and are generally classed according to what causes them to be safe:

- *Structurally safe faults* are those that are untestable due to the structure of the design itself.
- *On-line functionally safe faults* are those that do not pose a threat during operation mode, such as faults in the debug circuitry.
- *Application-specific safe faults (ASSFs)* are those that do not pose a threat as they are located in logic that is not exercised by the mission application program, such as unused address bits or a floating-point unit (FPU).

The relationship between faults and errors is detailed further in [1].

Safe faults are generally identified with the use of an automatic test pattern generator (ATPG) or formal tool for functional safety verification. On-line functionally and application-specific safe faults are identified by first extracting information about values taken by signals in the microprocessor DUT during application program execution and converting the information into design constraints, then applying the constraints to the design prior to running the ATPG or formal tool to return a list of untestable or “safe” faults. The number of safe faults identified depends, among other things, on how much information regarding the behaviour of the application program is captured in the aforementioned constraints.

When generating constraints, existing techniques for identifying ASSFs typically look at signals whose values are constant during application program execution, and do not give serious consideration to the potential for identifying faults that are safe due to a limited number of combinations being

This work is funded by the Engineering and Physical Sciences Research Council (EPSRC) (UK). Grant number EP/N509693/1 under DTP programme.

used by the application program for various signals in the microprocessor DUT.

C. This Work

The main contribution of this work is the demonstration of the significance of the increase in STL fault coverage when applying signal combination-based application-specific constraints to identify safe faults, and the increase in test compaction when considering safe faults during the compaction process. The proposed methodology was applied to several modules in the OpenRISC 1000-compliant Marocchino microprocessor [10] running representative application program tasks, with the signal combination-based constraints yielding up to 2.33% additional fault coverage compared to the constant signal-based constraints alone. Thus far, the safe faults-aware test compaction campaign has yielded a 1.09% reduction in the size of one test program out of eight that constitute the STL.

In the rest of this paper, Section II surveys existing literature on the use of ASSFs for optimising STLs, and explains the potential improvements offered by the signal combination-based constraints and safe faults-aware test compaction considered in this work. Sections III, IV and V describe the methodology and results obtained using industry standard tools. Section VI concludes the paper with comments on future work.

II. STL Optimisation By Discounting Safe Faults

A. Software Test Library (STL) Optimisation

Software-based self-test (SBST) works by executing test programs alongside functional programs on the processor DUT, allowing for less invasive periodic testing than methods involving scan chains. An STL is the set of test programs used in SBST. The STL must provide enough fault coverage to satisfy the safety goal with minimal resource overhead. If fault coverage is insufficient, it can be increased by writing additional tests to target uncovered faults, or by classifying such faults as "safe" so that they can be removed from the target fault list [5]. Test programs can be compacted to reduce memory footprint, and also to reduce test application time (TAT) which lowers power consumption and makes it easier to schedule test programs alongside mission tasks without violating the latter's real-time constraints [6]–[9].

B. Application-Specific Safe Faults Identification

Previous works extensively consider the phenomenon of idle signals existing in processors while application program tasks are being executed, indicating unused logic such as certain processor states or buffer entries. Various methods have been presented in literature to identify and translate these signals and their constant values into constraints in order to identify more safe faults [2]–[5].

The authors in [2] demonstrate how to identify idle input and flip-flop signals in a given processor module-under-test (MUT), then constrain them to their constant values during the ATPG process which returns a list of safe faults for the given constraints.

In [4], the authors similarly identify and constrain such idle signals at the inputs of the combinational logic in the DUT and then apply a static analysis at the DUT top-level to identify related safe faults. Performing the analysis at the top-level of the processor DUT may increase the complexity and thus reduce the number of safe faults that the flow is able to identify within a given time. On the other hand, performing the analysis at the module-level may result in some faults not being classed as safe since they propagate to the module outputs, even though they may not propagate to the processor primary outputs. An ATPG-based method may be limited to the module-level where the state space can be handled.

Other works have given some attention to signal combination-based constraints [3] [5].

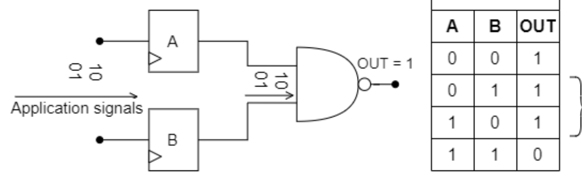
The authors in [3] look at common safe faults categories in microprocessors and how to identify them for different MUTs. When considering the decoder module, it is noted that, due to application programs typically not utilising the full instruction set of the target processor, several combinations of the multi-bit operational code (opcode) input signal to the decoder module are not utilised. Two opcodes (*mul* and *div*) are selected by the authors to represent unused combinations in order to demonstrate the point that more safe faults can be identified by introducing such constraints.

In [5], the authors utilise commercial register transfer-level (RTL) block and toggle coverage tools to identify idle or unused signals in a Controller Area Network (CAN) controller connected in a microprocessor system-on-chip (SoC) while executing test cases (design verification programs). This information is then translated into constraints in the form of SystemVerilog *assume* statements and fault propagation barriers which are applied in the Cadence JasperGold FSV tool, which, in turn, returns a list of safe faults including those caused by the applied constraints. The work detailed in [5] targets logic in the DUT that does not violate safety goals, which may be any parts not utilised in operational mode. In other words, the work primarily targets on-line functionally safe faults. However, the described technique could be deployed to identify ASSFs.

So far, no research has looked at the presence of safe faults across different processor modules derived from signal combination-based constraints. The work described in [3] only considers the decoder module for such constraints, that too without any profiling of a representative application program. The method presented in [5] may be capable of capturing safe faults due to signal combination-based constraints as it looks at unused registers, which are unused as a result of unused address combinations, however only a CAN bus controller design has been considered rather than a processor design. In addition, the increase in the number of safe faults identified due to such constraints is not examined, making it difficult to trade-off the improvement in fault coverage as a result of employing such constraints against the overhead of identifying the said constraints. These points are addressed in this paper.

C. *Application-Specific Signal Combination-Based Constraints*

An illustrative example to explain how considering application-specific signal combination-based constraints can help to identify more safe faults is given below with reference to Fig. 1. In the example, the application program behaviour is such that the combination of flip-flops $\{A,B\}$ only takes the values $\{1,0\}$ and $\{0,1\}$, causing the NAND gate output to be constant at *1* throughout the execution of the application program. This means a stuck-at-1 fault on this node is safe under the target application. If only constant signal-based constraints are considered, where a signal is either a primary input or a flip-flop, the safe fault will not be identified since none of the primary inputs or flip-flops in this example are at a constant value. However if signal combination-based constraints are taken into account then the safe fault at the output node of the NAND gate would be identified as the ATPG or formal tool would have the relevant information regarding the application program's behaviour.



NAND Truth Table		
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 1. Example circuit where signal combination-based constraints would help to identify the safe stuck-at-1 fault at the output of the NAND gate.

The number of safe faults identified is expected to increase with the use of the proposed constraints. The impact of this increase on the fault coverage for various processor modules is to be investigated in this work, and traded off against the additional effort needed to identify such constraints.

D. *Test Compaction*

Several works have looked at STL compaction. Techniques involving test instruction and program re-ordering aim to achieve the target fault coverage as soon as possible during the test, allowing for instructions (or programs) at the bottom end of the re-ordered test program (or STL) to be removed, thereby reducing TAT and improving code density [6] [7]. Compaction methods based on instruction removal and restoration have also been presented [8] [9]. The general methodology for compacting a test program is to remove instructions from the program and leave them out if the fault coverage does not change, otherwise restoring them in the program. In [8], removed instructions are replaced with *NOP* instructions to maintain coverage of length-dependent faults (LDFs), i.e., faults that can only be tested with a pattern of a certain length, while minimising TAT overhead. The granularity at which code is removed and restored can vary from blocks of instructions to individual instructions.

In [4], after the identification of ASSFs, a new STL was generated targeting the reduced fault list. This effectively reduces the STL size based on ASSFs, but includes the overhead of developing a new STL instead of making use of an existing STL provided by the processor IP developer, which is a common scenario in industry.

Thus far, no works have considered to enhance the compaction of an existing STL by identifying and removing code that only helps to cover on-line functionally and application-specific safe faults. This work explores the potential of such compaction. Structurally safe faults would not be considered here as they are not covered by STLs anyway due to being undetectable.

III. Methodology

This section details the proposed methodology for identifying ASSFs and then using the information to optimise STL fault coverage and memory footprint.

A. *Application-specific constraints*

For any module-under-test (MUT), constraints related to two classes of signals are to be extracted:

- *Application constants* - These are bits belonging to MUT primary input and flip-flop signals that do not toggle during application program execution.
- *Application combinations* - These are the combinations of MUT primary input signals seen during application program execution, which are typically a subset of all the possible combinations.

B. *Application-Specific Safe Faults Identification*

For a given MUT and application program task, the process for identifying ASSFs is shown in Fig. 2. The task is simulated on the processor DUT, returning a value change dump (VCD). The application-specific constraints for signals present in the MUT netlist are then extracted from the VCD. Finally, the constraints, MUT netlist, and relevant technology library information are passed to a formal tool to identify related safe faults (alternatively, an ATPG tool can be used for this step).

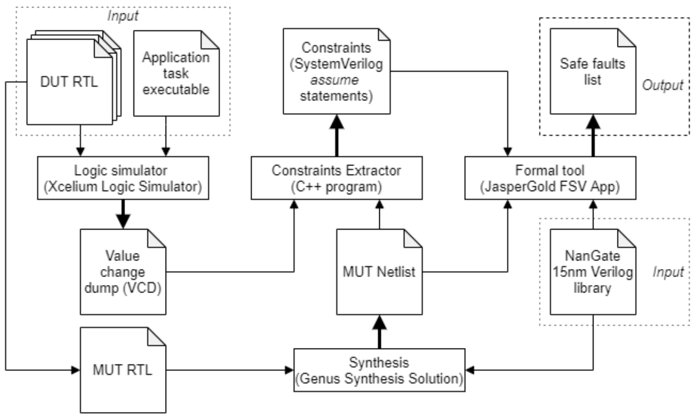

Input
DUT RTL
Application task executable
Constraints (SystemVerilog *assume* statements)
Safe faults list
Output
Logic simulator (Xcelium Logic Simulator)
Constraints Extractor (C++ program)
Formal tool (JasperGold FSV App)
Value change dump (VCD)
MUT Netlist
NanGate 15nm Verilog library
Input
MUT RTL
Synthesis (Genus Synthesis Solution)

Fig. 2. Methodology for identifying ASSFs for a given application task and module-under-test (MUT). Inputs and outputs to the methodology are indicated by the boxes with the dashed line borders. Data being generated in each step of the methodology is shown.

For any application program, the first two steps must be carried out for every possible execution profile, while also ensuring that signals that may vary in-field, such as data and and interrupt signals, are excluded from the analysis. The purpose of excluding such signals is that if any safe faults are derived from such a signal being constrained, then the faults are not guaranteed to be safe in-the-field. Such signals can be excluded either manually or automatically based on available design information. This design time overhead of the methodology has been acknowledged in related works [2].

Each task of the application program shall be put through the safe faults identification flow separately. Then, the intersection of the resulting safe fault lists under each task shall be taken as the safe fault list for the application program as a whole. This is represented in (1), where SF is the list of safe faults for the application program as a whole and SF_i is the safe fault list for the i^{th} task[1].

$$SF \supseteq \bigcap_{i=1}^{n} SF_i = SF_1 \cap SF_2 \cap ... \cap SF_n \tag{1}$$

If a fault is not common to the safe fault lists of each task, then it is not considered to be safe under the application program as a whole.

If the application code is updated after system deployment, the ASSF identification and related STL optimisation flow should be re-run, and the system updated with the correct STL.

Memory limitations of the formal or ATPG tool may necessitate that constraints are simplified. For this, a choice can be made to only monitor specific MUT primary inputs when identifying the different combinations used by the application.

C. Re-evaluating Fault Coverage

Once safe faults have been identified, they can be removed from the fault list and the fault coverage can be re-evaluated to check for improvement. Pruning safe faults from the fault list that are covered by the STL may reduce the improvement in fault coverage. To maximise fault coverage, only safe faults that are not covered by the STL can be removed, as described in [5]. On the other hand, it may be sensible to discount even those safe faults that are covered by the STL since such coverage is not relevant from a safety standpoint. This latter fault coverage calculation method would also be used when compacting an STL based on safe faults and checking for maintained coverage of dangerous faults only. The latter method is deployed in this work.

D. Safe Faults-Aware Test Compaction

The methodology presented in [9] is adapted for safe faults-aware test compaction of a given test program, and is shown in Algorithm 1. The key change is that even if fault coverage is lost after removing a segment of code from the test program, the segment may still be dropped if the coverage of dangerous faults is not compromised. As shown in step 7 of Algorithm 1, if coverage of both dangerous and safe faults is lost, then the segment is broken down further until the loss is only on the latter.

If a redundant test code segment is identified when experimenting on a given MUT, then, prior to dropping the code, the reduced test program is fault simulated on all other MUTs to check that no coverage of dangerous faults is lost.

This is one of many possible schemes for test compaction. Other schemes may be explored in future work.

[1](1) has been adapted from (1) in [13] which discusses uncontrollable lines instead of safe faults.

Algorithm 1: Safe faults-aware test compaction.

```
1 Fault simulate the test program TP; let DF and SF
  be the sets of dangerous and safe faults detected by
  TP respectively;
2 Split TP into m segments $S_0, ..., S_{m-1}$;
Remove segments as long as DF does not decrease:
3 for $i = m - 1$ down to 0; do
4       Set TP' = TP $\setminus$ $S_i$ (i.e., remove $S_i$ from TP);
5       Fault simulate TP' to obtain SF' and DF';
6       if (DF' < DF); then
7           if (SF' < SF); then
                Select one instruction $I$ from $S_i$;
                Set TP' = TP' $\cup$ $I$ (i.e., restore $I$);
                Go to 5 (i.e., refined analysis);
8           else;
                TP = TP' + $S_i$ (i.e., restore $S_i$).
                Go to 3 (i.e., check next segment);
9       else; ($S_i$ does not contribute to DF)
            Set TP = TP' (i.e., drop $S_i$ from TP);
            Go to 3 (i.e., check next segment);
  end for
```

IV. Experimental Setup

Experiments have been conducted to determine the increase in determined safe faults through the inclusion of application-specific signal combination-based constraints, and the corresponding increase in faut coverage of the STL. In addition, the improvement in test compaction through the consideration of safe faults has been explored.

A. Application-Specific Safe Faults Identification

The processor DUT is the superscalar out-of-order Marocchino processor [10]. The application program is represented by a set of four EEMBC Autobench 1.1 [12] benchmarks implementing algorithims for various different functions that may be found in an autononous system - *aifftr01* (Fast Fourier Transform), *bitmnp01* (Bit Manipulation), *canrdr01* (CAN protocol), and *puwmod01* (Pulse Width Modulation).

The Cadence Xcelium logic simulator is used to simulate the benchmarks executing on the DUT and return a VCD file for each benchmark. A C++ program has been developed to parse the VCD files and extract application constants and combinations, which are then formatted as SystemVerilog *assume* statements. Any signals that can change in-field from their simulation values, such as data signals, are specified manually so that they are not considered for constraints. MUTs are synthesised to the NanGate 15nm open cell library [11] using Cadence Genus synthesis solution. The synthesised DUT consists of 14,033 flip-flops, 334,640 logic gates, 22 primary inputs (PIs) and 19 primary outputs (POs). For each MUT, for each benchmark, the netlist and constraints are passed to the Cadence JasperGold Functional Safety Verification (FSV) tool, which tries to activate and propagate faults under the specified constraints in order to determine related safe faults.

The benchmarks are treated as separate tasks that make up an application program as a whole. Then, as per (1), for each MUT, the safe fault lists obtained for each task are merged.

B. Resource Consideration

The time allotted for running the JasperGold FSV check is dependent on the available development time, server memory and disk space. The default timeout is 24 hours, however after experimenting, this has been reduced to 1 hour to enable experiments to run without excessive design time overhead and without burdening the compute resources, while still demonstrating improvement in the number of safe faults identified through the proposed methodology. There is little to no variation in results past the 1 hour mark. The FSV check is run on a shared server, so memory usage and time must be managed in order to avoid interrupting the use of the server by other users. Comparing to related works, authors in [2] cite runtime for safe faults identification on the entire processor DUT "in the order of a few hours", while in [5] the FSV check timeout was set to the default 24 hours.

C. STL Development

The STL is built from existing verification programs for the DUT[2]. Fault coverage for stuck-at-{0, 1} faults is evaluated using gate-level fault injection in Xcelium logic simulator.

D. Safe Faults-Aware Test Compaction

Test program segmentation is performed manually by inspecting the test programs and inserting comments to mark the segments. Then the various test program versions are automatically generated. Fault simulations are performed using the aforementioned method. Further refinements are performed manually to segments where both safe and dangerous faults are being covered, in an effort to isolate code that can be removed without compromising coverage of dangerous faults.

V. Experimental Results

A. Identification of Application-Specific Safe Faults

Table I presents statistics on the number of safe faults identified when the following types of constraints are applied: 1) Structural (the design itself), 2) Structural and Application Constants, 3) Structural, Application Constants and Application Combinations. The data shown for each MUT is for the faults that are safe under every benchmark task, as per (1).

From Table I it can be seen that out of the six modules experimented on, there is an increase in the number of

[2]Available: https://github.com/openrisc/or1k-tests.

TABLE I
The amount of safe faults identified under constraints of various EEMBC benchmarks, in raw terms and as a percentage of the MUTs.

Module Name	Total Faults	Determined Safe Faults Under Different Constraints		
		Structural	***+ App Constants***	***+ App Combos***
Decode	4790	175 (3.65%)	214 (4.47%)	759 (15.85%)
Control	19656	647 (3.22%)	1018 (5.18%)	1046 (5.32%)
Reg File	3836	16 (0.42%)	40 (1.04%)	54 (1.41%)
ALU	8848	78 (0.88%)	632 (7.14%)	646 (7.30%)
Load-Store	86622	5168 (5.97%)	5454 (6.30%)	5454 (6.30%)
Ticktimer	4872	227 (4.66%)	429 (8.81%)	472 (9.69%)

determined safe faults of five of them due to the proposed application program signal combination-based constraints. For the *Decode* module, there is a more than 3X increase in safe faults, which may be because several opcodes are not being used by the application code. Among the unused instructions are *divide*, *rotate right* and *find last 1*. All but one *extend* and floating point instruction variants are unused. An unsigned variant of the *multiply* instruction is also not used.

Variation between benchmarks in numbers of ASSFs is relatively small. Under the *bitmnp01* and *canrdr01* tasks, the *Decode* module had 760 determined safe faults in the '+ *App Signal Combos*' constraint category. Under the *aifftr01* and *puwmod01* programs, the *Control* module had 1025 and 1053 determined safe faults for the '+ *App Constants*' and '+ *App Signal Combos*' constraint categories respectively.

B. STL Fault coverage

The STL was evaluated on the processor modules shown in Table II, which shows the raw fault coverage (*FC*), the fault coverage after discounting structual safe faults (*FC_S*), then after discounting ASSFs derived from constant signal-based constraints (*FC_C*) and, on top of that, signal combination-based constraints (*FC_CC*).

TABLE II
Fault coverage with and without considering safe faults.

Module	FC (%)	FC_S (%)	FC_C (%)	FC_CC (%)	FC_CC -FC_S(%)	FC_CC -FC_C (%)
Decode	82.78	85.92	87.23	89.56	+3.64	+2.33
Control	53.46	55.28	55.82	55.9	+0.62	+0.08
Reg File	84.18	84.42	84.71	84.84	+0.42	+0.13
ALU	83.65	84.39	84.64	84.78	+0.39	+0.14
Ticktimer	72.62	76.17	75.8	76.55	+0.38	+0.75

As shown in Table II, the fault coverage for each module-under-test (MUT) increases after discounting ASSFs, which agrees with the results presented in related works. The improvement in fault coverage due to the proposed application signal combination-based constraints, as opposed to only using application constant-based constraints, is shown in the final column of Table II. Fault coverage increases with each added constraint for all tested modules apart from the *Ticktimer* module. When only constant-based constraints are applied, the fault coverage for the *Ticktimer* decreases from 76.17% (FC_S) to 75.8% (FC_C). In this case, the proportion of additional safe faults identified that are covered by the STL is enough to decrease the fault coverage, indicating that the STL's effectiveness given the application program is not as good as initially thought. However when considering the safe faults identified with the aid of signal combination-based constraints, the fault coverage increases above FC_S to 76.55% (FC_CC), yielding a gain in fault coverage from the safe faults analysis process. It is indeed possible that for some other DUT and STL, the use of the proposed combination-based constraints could lead to a drop in fault coverage for the same reason explained in the case of constant-based constraints on the *Ticktimer*. The results presented in this paper are for a small open-source processor design, and may vary for a larger commercial design. A larger design would have a wider variety of functionalities, increasing the scope for there to be unused logic and thus more safe faults for a given application program.

The overhead of identifying primary input and flip-flop signals that can vary in-field exists whether or not the proposed signal combination-based constraints are used alongside the constant signal-based constraints, so there is no additional design time overhead incurred in that regard. The identification of the proposed constraints has a design time overhead in the order of a few minutes. As shown in Table II, these constraints can lead to an improvement in fault coverage and are worth including in the safe faults identification flow.

C. Safe Faults-Aware Test Compaction

Test compaction experiments were performed on the available test programs. For one program, a segment of 8 instructions was identified to be contributing to the coverage of two safe and no dangerous faults in the integer arithemtic and logic unit (*ALU*), and was thus dropped from the test program. The subject test program consists of several procedures for targeting various structures such as register, arithmetic, shift and logical circuitry. The compacted test program was re-evaluated on the other modules-under-test without a loss in fault coverage. The compaction results are shown in Table III.

TABLE III
TEST COMPACTION DUE TO CONSIDERATION OF SAFE FAULTS.

Module-under-test	ALU
Original program size	670 instructions; 2.936KB
Compacted program size	662 instructions; 2.904KB
Size compaction	8 instructions (1.19%); 0.032KB (1.09%)
Original program runtime	170,345 ns
Compacted program runtime	169,985 ns
Runtime compaction	360 ns (0.21%)

These results only represent the removal of instructions contributing to the coverage of safe faults, and are complementary to any compaction achieved by removing code not contributing the coverage of any faults.

Assuming that ASSFs have already been identified for improving fault coverage, the design time overhead of this safe faults-aware test compaction method is negligible when integrated into an existing test compaction flow. The main change required is that when a code segment is considered for dropping from a program, the criteria is that the code removal does not compromise *dangerous* fault coverage, rather than total coverage of dangerous and safe faults together.

VI. CONCLUSIONS

This work explores the use of safe faults in microprocessor STL optimisation with respect to fault coverage and test overhead. Signal combination-based constraints are employed for improving fault coverage, looking at the impact on modules across the processor that have not been considered in related works. Safe faults are considered during test compaction, where code segments contributing to coverage of ASSFs are identified and removed from the STL without compromising coverage of dangerous faults.

The results show improvements in fault coverage and test overhead with very low additional overhead in the constraints extraction and test compaction process. The effectiveness of the proposed methodology is design- and STL-dependent, and here results have been obtained for a small open-source processor with an STL that was constructed from existing verification programs for the processor architecture.

Future work will investigate the impact of the proposed methodology on a commercial safety-critical processor and STL, and the use of a more representative application program such as software for an autonomous system.

REFERENCES

[1] A. Avizienis, J. Laprie, B. Randell and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” in IEEE Transactions on Dependable and Secure Computing, Jan.-March 2004.

[2] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. S. Reorda and J. Mess, “An analysis of test solutions for COTS-based systems in space applications,” IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), 2018.

[3] C. Gursoy et al., “New categories of Safe Faults in a processor-based Embedded System,” IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2019.

[4] A. Ruospo, R. Cantoro, E. Sanchez, P. D. Schiavone, A. Garofalo and L. Benini, “On-line testing for autonomous systems driven by RISC-V processor design verification,” IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2019.

[5] F. A. da Silva et al., “Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs,” IEEE European Test Symposium (ETS), 2020.

[6] A. Touati, A. Bosio, P. Girard, A. Virazel, P. Bernardi and M. S. Reorda, “An effective approach for functional test programs compaction,” IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2016.

[7] R. Cantoro, E. Cetrulo, E. Sanchez, M. S. Reorda and A. Voza, “Automated test program reordering for efficient SBST,” 32nd Conference on Design of Circuits and Integrated Systems (DCIS), 2017.

[8] R. Cantoro, E. Sanchez, M. S. Reorda, G. Squillerò and E. Valea, “On the optimization of SBST test program compaction,” IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2017.

[9] M. Gaudesi, I. Pomeranz, M. S. Reorda and G. Squillero, “New techniques to reduce the execution time of functional test programs,” in IEEE Transactions on Computers, vol. 66, no. 7, pp. 1268-1273, 1 July 2017.

[10] or1k_marocchino. Available: https://github.com/openrisc/or1k_marocchino (Accessed 26th May 2021)

[11] 15NM OPEN-CELL LIBRARY AND 45NM FREEPDK. Available: https://si2.org/open-cell-library/ (Accessed 21st May 2021)

[12] About the EEMBC AutoBench™ Performance Benchmark Suite. Available: https://www.eembc.org/autobench/ (Accessed 31st March 2021)

[13] N. I. Deligiannis, R. Cantora, M. Sauer, B. Becker, M.S. Reorda, “New techniques for the automatic identification of uncontrollable lines in a CPU core,” 2021. In press.