Improving Fault Tolerance Utilizing Hardware-Software-Co-Synthesis.

Heinz Riener* Stefan Frehse* Görschwin Fey*†

*Institute of Computer Science
University of Bremen, Germany
{hriener, sfrehse, fey}@informatik.uni-bremen.de

†Institute of Space Systems
German Aerospace Center, Germany
goerschwin.fey@dlr.de

***Abstract*—Embedded systems consist of hardware and software and are ubiquitous in safety-critical and mission-critical fields. The increasing integration density of modern, digital circuits causes an increasing vulnerability of embedded systems to transient faults. Techniques to improve the *fault tolerance* are often either implemented in hardware or in software.**

In this paper, we focus on synthesis techniques to improve the fault tolerance of embedded systems considering hardware and software. A greedy algorithm is presented which iteratively assesses the fault tolerance of a processor-based system and decides which components of the system have to be hardened choosing from a set of existing techniques. We evaluate the algorithm in a simple case study using a *Traffic Collision Avoidance System* (TCAS).

***Index Terms*—Fault tolerance, Formal methods, Synthesis, Optimization**

I. Introduction

Embedded systems consist of hardware and software. In safety-critical and mission-critical fields, e.g., automotive engineering and aerospace, the reliability of embedded systems plays a crucial role. In this paper, we focus on assessing and improving the fault tolerance of embedded systems against *transient faults*. A transient fault temporarily modifies the expected functioning of a system causing an unstable, undesired situation [1]. For instance, incoming cosmic radiation may temporarily increase the energetic level of a signal which changes the system state. Although the functionality of the system is only momentarily changed, failure symptoms may be observed after a long period of time.

The vulnerability of embedded systems to transient faults increases with the integration density of digital circuits [2]. Besides of shielding the system, various techniques to detect the effects of transient faults and to improve the fault tolerance of a system have been introduced [3]. We refer to these techniques as *fault tolerance techniques*. For instance, common fault tolerance techniques add redundancy to the system in order to produce multiple copies of critical data [4]. A *failure* can then be *detected* when two copies of the same data block do not agree on their values. Moreover, when a third copy of the same data block is available and two out of three copies agree on their values, the failure can be *corrected*.

Fault tolerance techniques can be either implemented in hardware or in software. Thus, we distinguish between *hardware techniques* and *software techniques*. The characteristics of hardware techniques and software techniques are diverging: software techniques tend to reexecute critical computations which causes mainly a time overhead at run time. Hardware techniques tend to replicate components which increases the circuit's area but — since the components operate in parallel — has only little effect on the system performance.

In practice, an engineer faces the problem to optimize the fault tolerance by choosing system components and deciding which hardware and software techniques to apply to the chosen components. Additionally, cost functions of the system, e.g., circuit area or performance, with conflicting goals have to be optimized. This problem can be seen as a multi-objective optimization problem.

We propose a greedy algorithm which iteratively assesses and improves the fault tolerance of a given processor-based system. The processor-based system consists of a set of hardware components and is designed to execute a particular program. We abstract from the exact implementation of the hardware but assume that the hardware is fault tolerant except for the considered hardware components. The hardware of a processor-based system consists of a controller and a datapath. We assume that the controller is fault tolerant and thus we focus on applying fault tolerance techniques to harden the datapath. Our algorithm decides which software instructions and which hardware components have to be hardened choosing from a set of existing fault tolerance techniques. The assessment of the fault tolerance uses a model checker which analyzes the machine code of the software implementation and the gate-level representation of the hardware design. We evaluate the algorithm in a simple case study using a *Traffic Collision Avoidance System* (TCAS) [5] which is publicly available in the Software-artifact Infrastructure Repository (SIR) [6].

II. Preliminaries

We consider a specific class of embedded systems which we call *processor-based systems*. A processor-based system consists of a processor which executes the machine code of a program. Focusing on the functional parts of the execution unit, a processor-based system is built from a set $\mathsf{C} = \{c_1, c_2, \dots, c_n\}$ of hardware components. Each component $c \in \mathsf{C}$ is associated with a type $\mathsf{t}(c) \in \mathsf{Ops}$, where Ops is the set of operations supported by the processor-based system.

The processor-based system may have redundant components, i.e., for a hardware component $c \in \mathsf{C}$ there are one or more alternative instances $\mathsf{alt}(c) = \{c^{(1)}, c^{(2)}, \dots, c^{(k)}\}$ which implement the same functionality in form of different circuits. The individual alternatives differ in two measures: (1) the *fault tolerance* and (2) the *costs*. The fault tolerance $\mathsf{ft}_{\mathtt{HW}}(c^{(j)}) \in [0, 1]$ measures the ability of the instance to function correctly while affected by transient faults and is given as a real value in the interval $[0, 1]$. A high value indicates that the instance is less vulnerable to transient faults, whereas a low value indicates that transient faults more likely affect the correct functioning of the component. The costs are defined in form of one or more cost functions $\mathsf{cost}_{\mathtt{HW}}(c^{(j)}) \in \mathbb{R}$. We use superscripts, e.g., $\mathsf{cost}^{\mathtt{time}}_{\mathtt{HW}}$ or $\mathsf{cost}^{\mathtt{space}}_{\mathtt{HW}}$, to distinguish between different cost functions. Each cost function measures the complexity of instances by quantifying a specific characteristic of the corresponding circuits. For instance, a cost function $\mathsf{cost}^{\mathtt{space}}_{\mathtt{HW}}$ may count the number of gates of a circuit.

A program defines a control-flow graph over instructions with a fixed set of input registers and a fixed set of output registers. We focus on *transformational programs* and assume that the program is initially fixed, i.e., each processor-based system

This work was supported by the German Research Foundation (DFG, grant no. FE 797/6-1). Copyright: 978-3-9815370-0-0/DATE13/©2013 EDAA

is designed to execute a specific transformational program which is stored in the system memory. A transformational program computes a final result at the end of a terminating computation. Each transformational program consists of a set of instructions. We deal only with a subset Inst of all instruction of the program. An instruction $i \in \mathsf{Inst}$ has at most two operands and produces one result. Thus, instructions which manipulate the control-flow of the program, e.g., branching instructions, are not included in Inst. Moreover, we exclude instructions with side-effects from Inst. Otherwise, adding redundancy, e.g., by duplicating instructions, may change the behavior of the program. Each instruction $i \in \mathsf{Inst}$ has a type $\mathsf{t}(i) \in \mathsf{Ops}$ and can be executed by any instance of a hardware component with the same type. For instance, an instruction of a multiplier type can be executed by any multiplier of a processor-based system. A *binding* defines which instruction is executed on which instance of a hardware component, i.e., the binding is a complete function mapping from Inst to $\bigcup_{c \in \mathsf{C}} \mathsf{alt}(c)$. Lastly, we associate each transformational program with a *fault tolerance* $\mathsf{ft}_{\mathtt{SW}}$ which quantifies the ability of the program to function correctly.

This paper focuses on assessing and improving the fault tolerance of a processor-based system considering the hardware components C of the system and the executed program P. We restrict our perspective to single errors assuming that only one transient fault affects the system at a time [2]. We use $\mathsf{cost}_{\mathtt{space}}(\mathsf{P}, \Phi)$ and $\mathsf{cost}_{\mathtt{time}}(\mathsf{P}, \Phi)$ to describe the costs of the system. If the parameter Φ is not explicitly stated, i.e., $\mathsf{cost}_{\mathtt{space}}(\mathsf{P})$ and $\mathsf{cost}_{\mathtt{time}}(\mathsf{P})$, a non-deterministic binding is used. The exact definitions of the costs depend on the application. However, we assume that an upper bound on the time costs $\hat{b}_{\mathtt{time}}$ and the space costs $\hat{b}_{\mathtt{space}}$ is given which the system must not exceed. The *fault tolerance optimization problem* is then stated as follows: find a binding Φ and a program P' functionally equivalent to P such that the fault tolerance $\mathsf{ft}(\mathsf{P}', \Phi)$ of the system becomes maximal without exceeding the given cost bounds, i.e., $\mathsf{cost}_{\mathtt{time}}(\mathsf{P}', \Phi) \leq \hat{b}_{\mathtt{time}}$ and $\mathsf{cost}_{\mathtt{space}}(\mathsf{P}', \Phi) \leq \hat{b}_{\mathtt{space}}$.

III. Fault Tolerance Optimization

In this section, we present the greedy algorithm which iteratively assesses and improves the fault tolerance of a processor-based system.

We assess the fault tolerance of a processor-based system in two steps: firstly, we compute the fault tolerance of each individual instruction $i \in \mathsf{Inst}$. Recall that the instructions do not manipulate the control-flow of the program and are free from side-effects. We say that an instruction is *robust* if the values of the output registers of the program do not diverge from their correct values when an instruction is affected by a transient fault. Otherwise, we say that the instruction is *non-robust*. Secondly, for all non-robust instructions we consider the fault tolerance of the hardware instance on which the instruction is executed.

The fault tolerance of the software is computed by utilizing a combination of *Satisfiability (SAT)-based equivalence checking* [7] and *hardware robustness checking* [8]. The program is instrumented with additional fault injection logic which models the possible effects of transient faults. The instrumented program is then formalized as a logic formula. The logic formula is satisfiable *if and only if* (iff) a transient fault affects an instruction and causes the value of at least one output register to diverge from its correct value. In particular, the logic formula expresses the semantics of the program twice: firstly, the semantics of the program is encoded without any changes. Secondly, the semantics of the instrumented program is encoded. The fault injection logic is used to enable and disable the semantic effects of individual instructions, i.e., the result of an instruction is replaced by an open variable.

We use a *Satisfiability Modulo Theories* (SMT) solver to check for the satisfiability of the logic formula. The solver searches for satisfying assignments where the semantic effect of exactly one instruction is disabled. A satisfying assignment corresponds to a witness which shows that the open variable affects an output register. If the formula is unsatisfiable, then no replacement for the open variable propagates to the program's output registers.

Suppose that instructions $\mathsf{Inst} = \mathcal{T} \cup \mathcal{S}$ are partitioned into *robust instructions* $\mathcal{T}$ and *non-robust instructions* $\mathcal{S}$, respectively. The fault tolerance $\mathsf{ft}_{\mathtt{SW}}$ of a program P is then quantified as

$$\mathsf{ft}_{\mathtt{SW}}(\mathsf{P}) = \frac{|\mathcal{T}|}{|\mathsf{Inst}|} = \frac{|\mathsf{Inst} \setminus \mathcal{S}|}{|\mathsf{Inst}|}. \quad (1)$$

Robust instructions are not vulnerable to transient faults regardless on which instance of a hardware component the instructions are executed. For each non-robust instruction, we additionally consider the hardware instance on which the instruction is executed. Suppose Φ is a binding which maps instructions to hardware instances with the same type. The function

$$r(i, \Phi) = \begin{cases} \mathsf{ft}_{\mathtt{HW}}(\Phi(i)) & \text{if } i \in \mathcal{S} \\ 1 & \text{otherwise,} \end{cases} \quad (2)$$

computes the fault tolerance of a given instruction $i \in \mathcal{S} \subseteq \mathsf{Inst}$ considering Φ.

The fault tolerance of the processor-based system for the given binding Φ

$$\mathsf{ft}(\mathsf{P}, \Phi) = \frac{\sum\limits_{i \in \mathsf{Inst}} r(i, \Phi)}{|\mathsf{Inst}|} \quad (3)$$

is the sum of the fault tolerances of all instructions normalized by the number of instructions.

Fig. 1 shows a flow diagram of the greedy algorithm. The input of the algorithm is a program P, the set C of hardware components, an upper bound $\hat{b}_{\mathtt{space}}$ on the space costs, an upper bound $\hat{b}_{\mathtt{time}}$ on the time costs, and additional parameters $\lambda_1, \lambda_2, \ldots, \lambda_q$ which are used to configure heuristics. Depending on the heuristic in use, the meaning of the parameters λ_k changes. In each iteration, the algorithm produces a new program P'_l which is functionally equivalent to P and a binding Φ_l such that the system fault tolerance has been improved without exceeding the given bounds. Since all instructions are free from side-effects, the conducted code transformation which adds redundancy to the program is by construction guaranteed to produce a functionally equivalent program.

1) Heuristic Software Hardening: In the first step, the algorithm applies software fault tolerance techniques to the program P. Existing software fault tolerance techniques include [9], [10], [11], [12], [13] which instrument the machine code of the program to either detect or correct transient faults. Our greedy algorithm uses a heuristic approach to decide which technique is applied to which instruction. The heuristic produces k differently hardened programs $\mathsf{P}'_1, \mathsf{P}'_2, \ldots, \mathsf{P}'_k$ when applied to P. Each program P'_l is functionally equivalent to P. If a cost function of a hardened program P'_l exceeds the corresponding cost bound, the program is rejected. For instance, suppose $\mathsf{cost}_{\mathtt{time}}(\mathsf{P}'_l) \geq \hat{b}_{\mathtt{time}}$ then P'_l is rejected. If none of the hardened programs remains, then the algorithm terminates early with the results of the previous iteration. Early termination is not shown in Fig. 1.

2) Assessing the Software Fault Tolerance: In the second step, we use bounded model checking to compute the fault tolerance for each of the programs produced in the first step. Recursive functions and loops are unrolled when needed [7]. We extended a SAT-based software model checker,

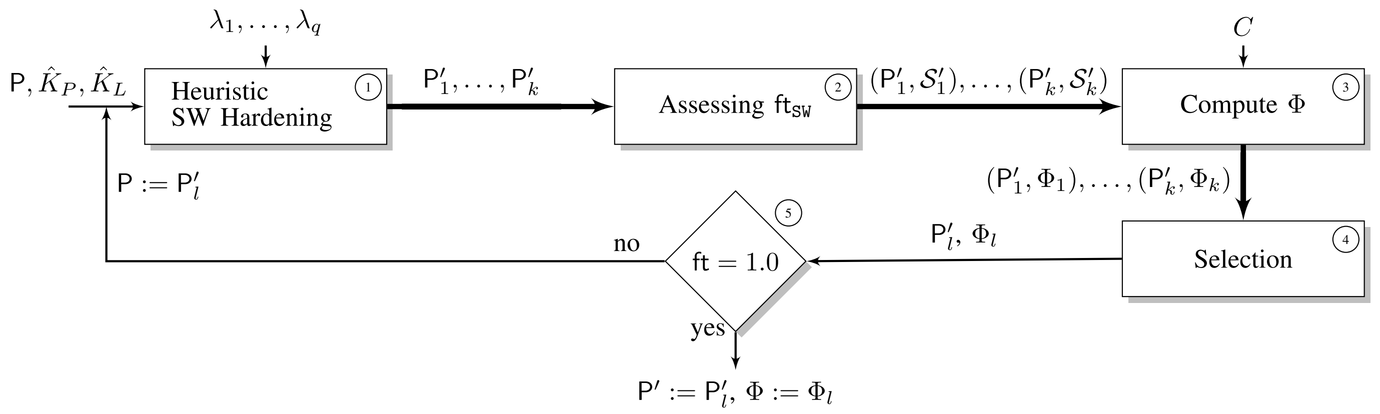
$\lambda_1, \dots, \lambda_q$
$\mathsf{P}, \hat{K}_P, \hat{K}_L$
Heuristic SW Hardening
1
$\mathsf{P}'_1, \dots, \mathsf{P}'_k$
Assessing $\mathsf{ft}_{\mathtt{SW}}$
2
$(\mathsf{P}'_1, \mathcal{S}'_1), \dots, (\mathsf{P}'_k, \mathcal{S}'_k)$
C
Compute Φ
3
$(\mathsf{P}'_1, \Phi_1), \dots, (\mathsf{P}'_k, \Phi_k)$
Selection
4
P'_l, Φ_l
$\mathsf{ft} = 1.0$
5
no
yes
$\mathsf{P} := \mathsf{P}'_l$
$\mathsf{P}' := \mathsf{P}'_l, \Phi := \Phi_l$

Fig. 1. Flow diagram of the Greedy Algorithm

called FAuST [14], which operates on *Low Level Virtual Machine* (LLVM) [15], i.e., a RISC-like intermediate representation. The model checker computes the set $\mathcal{S}_l$ of non-robust instructions for each program P_l.

3) Computation of Bindings: In the third step, the instructions are bound to hardware instances which execute them. Non-robust instructions have to be executed on robust hardware instances in order to maximize the fault tolerance of the processor-based system. For each program P'_l, a binding Φ_l is computed which maximizes the system fault tolerance without exceeding the given cost bounds.

The best binding is computed by formalizing and solving an optimization problem. The optimization problem is expressed as a logic formula with *Pseudo-Boolean* (PB) constraints and solved by calling an SMT solver incrementally. The optimal solution corresponds to a binding Φ_l which maximizes the fault tolerance without exceeding the given cost bounds.

The logic formula becomes unsatisfiable if all possible bindings exceed the given cost bounds. The corresponding program is then rejected. Again, if none of the programs remains, the algorithm terminates early with the results of the previous iteration.

4) Selection of Program and Binding: In the fourth step, the algorithm selects one of the pairs (P'_l, Φ_l) with the highest fault tolerance from the sequence $(\mathsf{P}'_1, \Phi_1), \dots, (\mathsf{P}'_k, \Phi_k)$ computed in the third step, i.e., $\mathsf{ft}(\mathsf{P}'_l, \Phi_l) \geq \mathsf{ft}(\mathsf{P}'_j, \Phi_j)$ for all $1 \leq j \leq k$.

5) Checking for Termination: In the fifth step, the algorithm terminates with the pair (P'_l, Φ_l) if a fault tolerance of 100% is achieved and otherwise proceeds in another iteration using the program P'_l as input.

IV. Case Study

A. Experimental Setting

In this section, the proposed greedy algorithm is evaluated in a case study. We consider a processor-based system consisting of a simple CPU which executes the transformational program TCAS [6]. TCAS is a publicly available program for collision avoidance of airplanes. We use the Clang 3.1 compiler to transform the program into LLVM-IR. We consider a fictional CPU which provides hardware components to execute each LLVM instruction type used in the program. The circuit of each hardware component is combinational. Hence, the time costs $\mathsf{cost}^{\mathtt{time}}_{\mathtt{HW}}$ are constantly 1 for all predefined hardware instances. The space costs $\mathsf{cost}^{\mathtt{space}}_{\mathtt{HW}}$, however, count the number of gates in the circuit. For each hardware component, we additionally provide robust circuits by utilizing *Triple Modular Redundancy* (TMR). TCAS uses 7 different instruction types which results in 14 hardware instances in total.

For the evaluation, we implemented one software fault tolerance technique, called *Error Detection by Duplicated Instruction* (EDDI) [10]. EDDI is illustrated in Fig. 2: the left-hand-side shows a single instruction which adds two registers and stores the result in r_0. On the right-hand-side, the same instruction is shown (line 0) after EDDI has been applied. EDDI duplicates the instruction using another register r'_0 to store the result (line 1). An additional comparison (line 2) checks whether the values of the two registers r_0 and r'_0 are equal and otherwise signals an error to the environment (line 3). We have extended the model checker FAuST: the special function `sig_error_if` can now be used to check whether a fault has been detected.

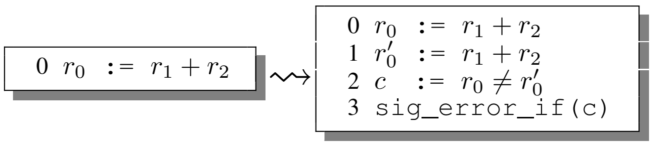

Fig. 2. Hardening an instruction using EDDI

The space costs of the system are defined as the accumulated number of gates of the hardware instances after binding:

$$\mathsf{cost}_{\mathtt{space}}(\mathsf{P}, \Phi) = \sum_{c \in H} \mathsf{cost}^{\mathtt{space}}_{\mathtt{HW}}(c),\ H = \bigcup_{i \in \mathsf{Inst}} \Phi(i)$$

The time costs of the system are measured by computing the length of the longest path in the program. Since all hardware instances are combinational circuits, the result of each instruction is available after 1 time step:

$$\mathsf{cost}_{\mathtt{time}}(\mathsf{P}, \Phi) = \mathsf{longestPath}(\mathsf{P})$$

The greedy algorithm uses a heuristic to improve the fault tolerance of the transformational program (first step in Fig. 1): we create a priority list of the instruction types ranking them by the space costs of their associated hardware instances. EDDI is then systematically applied to all instructions of the instruction type with the highest priority in the priority list. As a result a new program P' functionally equivalent to P is created and the instruction type with highest priority is removed from the priority list. If the created program P' exceeds one of the cost functions, the program is reject and the heuristic proceeds with the next priority from the priority list. Eventually, either a program is created which does not exceed any cost function or the priority list becomes empty. Then, the greedy algorithm proceeds with the second step. In the first case the input of the second step is the newly created program P' and in the later case the input is the program P.

Exp	$\hat{b}_{\texttt{space}}$	$\hat{b}_{\texttt{time}}$	$\mathsf{cost}_{\texttt{space}}$	$\mathsf{cost}_{\texttt{time}}$	ft	$\vert\mathsf{Inst}\vert$	t	It
SW	1559	∞	1559	487	100%	74	20.51s	1
HW	∞	247	4746	247	100%	0	15.91s	1
1	4000	247+50	3999	253	100%	1	23.44s	1
2	3500	247+50	3482	259	100%	2	32.76s	2
3	3000	247+50	2998	259	100%	2	34.99s	2
4	2500	247+50	2440	283	100%	6	202.80s	4
5	2000	247+50	1983	295	100%	10	90.58s	6
6	1800	247+50	1747	295	98%	13	82.13s	7
7	1800	247+100	1792	313	100%	15	86.64s	7
8	1600	247+50	1590	295	92%	13	22.98s	6
9	1600	247+100	1587	319	94%	21	24.59s	8
10	1600	247+150	1582	385	96%	41	24.21s	9
11	1600	247+200	1588	415	100%	50	31.02s	10

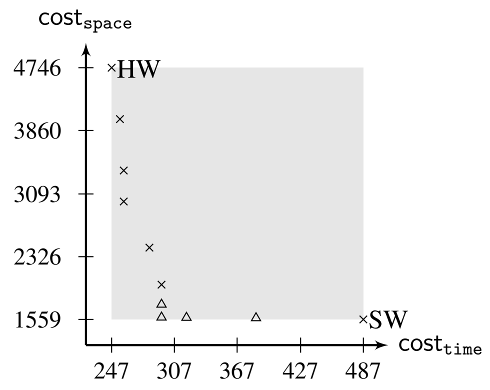

Fig. 3. Experimental results for TCAS

B. Experimental Results

All experiments are conducted on a Linux machine with an Intel Core i7 CPU (2.4GHz, 8GB RAM). The SMT solver Z3 [16] (version 3.2) was used to check for satisfying assignments of logic formulae. In all experiments, the memory consumption was less than 150MB.

Fig. 3 lists the experimental results. The table on the left side shows 13 experiments. Additionally, the solution space has been visualized on the right side. The table is built as follows: the first column identifies the experiment by an unique id. For the first two experiments `HW` and `SW` we allow for unlimited space and time costs, respectively, which is denoted by the symbol ∞. The second and third column list the cost constraints. The next two columns list the computed costs of the final result produced by the greedy algorithm for the processor-based system. The sixth column shows the computed fault tolerance and the number of instructions hardened by using EDDI. Our greedy algorithm creates a program with 100% fault tolerance if the cost constraints are not too tight. The last two columns list the required run time in seconds and the number of iterations.

The experiments `SW` and `HW` present two special cases. In the `SW` experiment, the fault tolerance was improved using software fault tolerance techniques exclusively, i.e., EDDI was applied to all instructions. The instructions are bound to the hardware instances with the lowest costs and lowest fault tolerances. In the `HW` experiment, the fault tolerance was improved using hardware fault tolerance techniques exclusively, i.e., EDDI was not applied but the instructions are bound to the hardware instances with the highest fault tolerances. The fault tolerance of the system is in both cases 100%.

In the remaining experiments 1 to 11, we systematically evaluated the greedy algorithm using different cost constraints. For the space costs 4000, 3500, 3000, 2000, 1800 and 1600 were chosen as upper bounds. The longest path of TCAS counts 247 instructions. Thus, for the time costs, 247, $247+50$, $247+100$, $247+150$, $247+200$ were selected as upper bounds. If the algorithm terminated with a fault tolerance less than 100%, we run another experiment and incremented the time costs by 50. For example, our algorithm could not find a solution for the time cost constraint $247 + 50$ and thus we relaxed the time cost constraint in Experiment 7.

Fig. 3 on the right side shows a scatter plot of the time costs and the space costs of the experiments. On the horizontal and vertical axis the time and space costs are plotted, respectively. A cross marks a solution which has a fault tolerance of 100%. A triangle marks a solution with a fault tolerance lower than 100%. The fault tolerance is only lower than 100% if one of the cost constraints is too tight.

Our algorithm effectively determines solutions with balanced costs within short run times. In contrast, exclusively hardening in software or hardware leads to high costs. Exclusively hardening in software requires 97% additional run time costs and hardening in hardware requires 204% additional space costs, respectively. In Experiment 7, our algorithm computes a solution with 100% fault tolerance considering only 14% additional space costs and only 26% additional time costs.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Lanwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] S. Baeg, J. Bae, S. Lee, C. S. Lim, S. H. Jeon, and H. Nam, "Soft error issues with scaling technologies," in *Asian Test Symposium*, 2012, pp. 68–68.

[3] D. K. Pradhan, *Fault-Tolerant Computer System Design*. Prentice Hall PTR, 2006.

[4] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria," in *International Conference on Software Engineering*, 1994, pp. 191–200.

[6] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[7] D. Kröning, "Software verification," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, pp. 505–532.

[8] G. Fey, A. Sülflow, S. Frehse, and R. Drechsler, "Effective robustness analysis using bounded model checking techniques," *IEEE Transactions on CAD*, vol. 30, no. 8, pp. 1239–1252, 2011.

[9] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 33–42.

[10] N. Oh, P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in superscalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[11] ——, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[12] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *IEEE International On-Line Testing Symposium*, 2003, pp. 137–143.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[14] H. Riener and G. Fey, "FAuST: A framework for formal verification, automated debugging, and software test generation," in *International SPIN Workshop on Model Checking of Software*, 2012, pp. 234–240.

[15] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–88.

[16] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.