Transient Fault Detection in State-Automata

Bernhard Fechner, Andre Osterloh

Department of Computer Science

FernUniversität in Hagen

{Bernhard.Fechner, Andre.Osterloh}@fernuni-hagen.de

Abstract

State automata are implemented in numerous ways and technologies – from simple traffic light controls to high-performance microprocessors comprising thousands of different states. Highly-integrated microprocessors get more and more susceptible to transient faults induced by radiation, extreme clocking, temperature and decreasing voltage supplies. A transient fault in form of a single event-upset (SEUs) can change the current state of an automaton to another valid state, thus causing a control-flow error. From control-flow based simulations of a microprogrammable automaton we determine the number of effective, overwritten and latent faults. Faults can be detected by counting the number of transitions to the ending state and the comparison with a precomputed value being part of the microcode and the number of counted cycles. Faults cannot be detected if the original state is transferred to another valid state, reaching the ending state in the same number of transitions. We further determine the number of faults which can be detected by using this simple scheme and propose to encode these states in a way that a bit-flip will result in a state with a different distance from the ending state without any additional space consumption for the code.

1. Introduction

With shrinking feature sizes, increasing clock frequencies and decreasing voltages, transient faults are imminent in modern consumer electronics. Their probability is 5 to 100 times higher than for permanent faults [2]. Apart from radiation, they can be caused from power fluctuations, loosely coupled units, timing-faults, meta-stable states and environmental influences (radiation, temperature, humidity, and force). Single Event Upsets (SEUs) are able to change the state of memory elements. Modern microprocessors consist of a very complex control logic and memory on the microarchitectural level. We distinguish two implementation types for state automata. A microprogrammed (μprogram) or microprogrammable control and finite state automata to control e.g. floating point units etc. A typical implementation in modern microprocessors is the microcode (μcode) storage. Figure 1 shows a microprogrammed control including possible locations of a fault from the fault model. For the selection of microinstructions a start address is used. It is determined either by the opcode of the related machine instruction or field ‘next address’. The selected control word will be directed to an execution unit. The field ‘next address’ can be encoded by using One-Hot, Binary, Gray[6] or any other bijective encoding chosen by the microcode designer. To implement conditional jumps within the microprogram, a microprogram counter is used. E.g. if a condition is true the contents of the field ‘next address’ are chosen as next address, else the incremented microprogram counter. ASIC implementations often realize the μcode storage as ROM. Therefore the only way a SEU can compromise the execution of a microcode program is to change the contents of the latch holding the next address/ the microprogram counter.

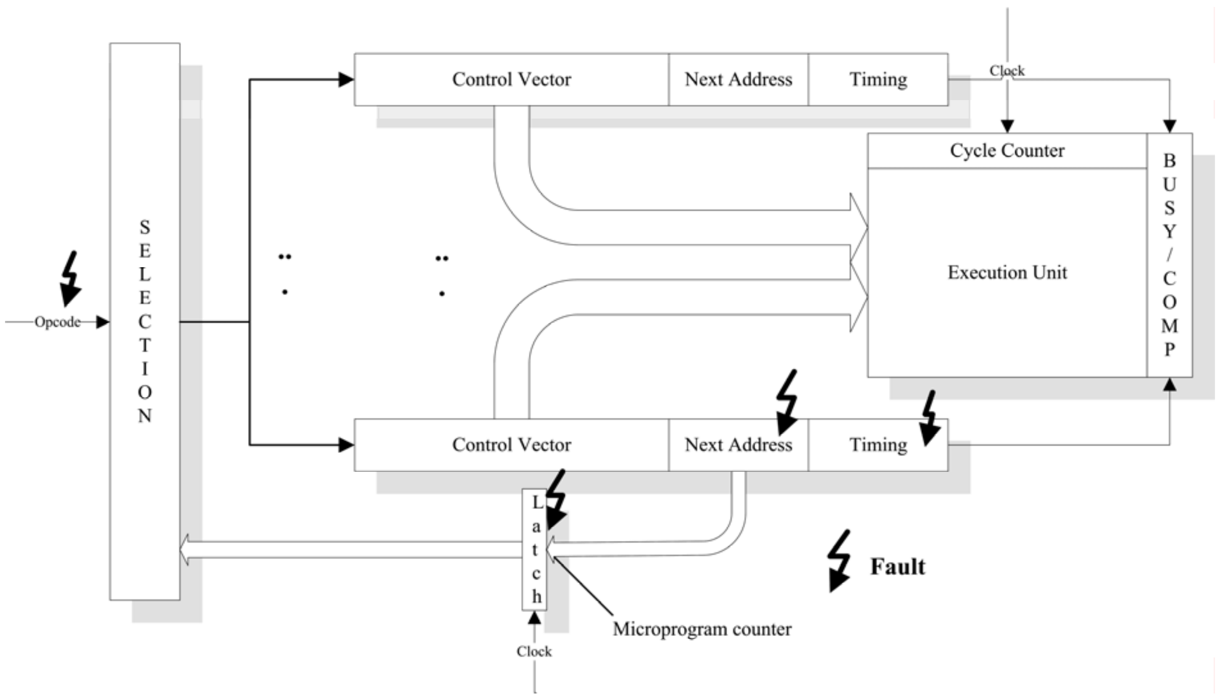

Figure 1: A microprogrammable control

For performance reasons, the ROM is transferred to a faster RAM-based storage. One example is the reversible µcode-ROM update since Intel Pentium 3 systems. In SRAM-based FPGA implementations or any microprogrammable control the whole µcode storage is susceptible to SEUs. If a SEU irritates the opcode, the field 'next address' or the microprogram counter, another microcode or an undefined entry (no control vector associated to that entry) will be selected. Faults in the control flow can be detected by counting the number of transitions to the last state and by comparing a precomputed value being part of the microcode and the number of counted cycles. For details on the discussed scheme, see [8]. The faults we indent to recognize are control-flow faults. We assume one fault at a time in a state since multiple bit faults are extremely seldom. For a brief introduction into related work, please see [8].

We distinguish three error manifestations:

- Overwritten: an error which is overwritten by a correct value before the faulty value was read. A bit-flip fault occurs twice at the same bit position in the same microinstruction.
- Effective: an error is effective, if it manifests within the specified observation interval. An effective error is neither overwritten nor latent.
- Latent: a fault does not get effective in the observation interval. This can be e.g. a fault in a microinstruction or microprogram which is not referred in the observation interval.

The paper is organized as follows. In Section 2, we present preliminary definitions. Different state encodings and the probability of a state transition by a single event upset regarding encodings like Gray, Binary and One-Hot and the results of a fault-coverage analysis are presented in Section 3. Section 4 concludes the paper.

2. Basic Definitions

In the following we will examine probabilistic finite state automata. A probabilistic finite state automaton is a triple T=(V, E, p) where (V, E) is a directed graph with n:=|V| nodes (also called states), r:=|E| edges and a function $\mathrm{p}: \mathrm{E} \rightarrow [0,1]$ such that for all $\mathrm{v} \in \mathrm{V}$ we have $\sum_{(v,x)\in \mathrm{E}} \mathrm{p}(v, x) \in \{0, 1\}$. We call a state $\mathrm{s} \in \mathrm{V}$ a *final state* iff the outdegree of s∈(V, E) is zero and denote the set of final states by F. For all states s∈V\F we demand $\sum_{(s,x)\in \mathrm{E}} \mathrm{p}(s, x) = 1$. We call a state s∈V a *starting state* iff the indegree of s ∈ (V, E) is zero. We model the execution of a microprogram as a path p in T that starts in a starting state and ends in a final state. Hence we can identify the length of a path (here the length of a path is the number of edges in the path) with the execution time of a microprogram. Our idea is that we have information about the execution time of an error-free execution of a microprogram. This information will be loaded when a program starts (in a starting state) and it will be compared with the real execution time when a program reaches a final state. If we are able to detect an error due to the fact that the real execution time differs or due to the fact that the maximal execution time is exceeded we say that we are able to *detect the error by counting steps.*
To represent the simple structure of microprograms we restrict our attention to the case where (V, E) is a directed acyclic graph with exactly one starting and one final state. Furthermore the outdegree of each state have to be at most two. Since we have a probabilistic automaton we get a probability for a path in T. Let $w=e_1e_2\ldots e_l$ be a path in T, i.e. $e_i=(a_i,b_i)\in$ E, and $b_i=a_{i+1}$, the probability of w is $\mathrm{p}_w := \prod_{i=1}^{l} \mathrm{p}(e_i)$. States of our automaton are represented as words of length $\mathrm{m} := \lceil \log_2 \mathrm{n} \rceil$ over $\{0,1\}$. So edges are words of length 2m. We denote with $\mathrm{c}: \mathrm{V} \rightarrow \{0,1\}^{\mathrm{m}}$ an injective function that assigns states of T to their representation as words of length m over $\{0, 1\}$. We call c a *state encoding* of T or *coding* for short. We assume that one one-bit error occurs. In our model such an error only manipulates the successor of a state (this corresponds with the next address field), i.e. the target of an edge. Hence for an automaton with n states and r edges the total number of bits that can be manipulated by a one-bit error is $\mathrm{r}\lceil \log_2 \mathrm{n} \rceil$=rm. One bit is changed with a probability of 1/rm. We assume that we are able to *detect an error by counting steps* when the error changes a successor of a state to a nonexistent state[1]. In our model the probability p(e) of an edge e is not changed by an error.
We are interested in the probability that an effective (i.e. the execution of the program is manipulated by an error) one-bit error can be detected by counting steps as described in the previous paragraphs. In our paper we call this probability α.

We give a short list to recapitulate what we assume that can be detected by counting steps:

If a program reaches a final state: The execution time differs from the calculated time.
During the execution of a program: The calculated maximal execution time is exceeded.
During the execution of a program: A nonexistent state is reached.

We define β:=1- α.

2.1. An Example and Further Definitions

In Figure 2 an example of an automaton is given. The automaton has one starting state (node 1), and one final state (node 13). Possible path lengths from state 1 to state 13 are 6 and 7. Some of the edges in the figure have probabilities ($p_1,\ldots,p_4>0$). If no annotation is given the probability is assumed to be one. In our example two paths from 1 to 13 of length 6 exist. One of these paths has probability p_2p_3,

[1] We assume that the program/automaton infinitely cycles in an undefined state and hence the maximal execution time will be exceeded.

the other one has probability p_2p_4. Hence a path length of 6 occurs with probability p_2 and a path length of 7 occurs with probability $p_1=1-p_2$ (note that we have $p_1+p_2=1$ and $p_3+p_4=1$).

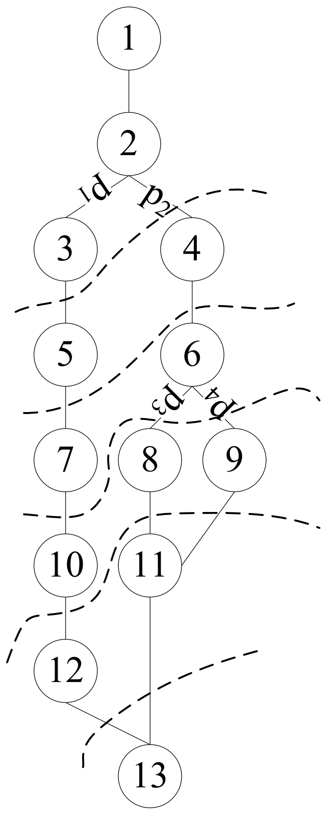

Figure 2: Example of a microprogram

The valid path lengths could be stored by setting bits in a sufficiently large bit vector. This bit vector can be loaded when the microprogram starts in state 1. We calculate α for the automaton T=(V, E, p) given in Figure 2. The states are encoded using the binary representation of their numbers filled up with zeros to get the correct length, i.e. node 1 is written as 0001 etc.
For clarity we show an example encoding for Figure 2 in Table 1.

Table 1: State encoding example

Node i	c(i)	Successor1	Succssor2
1	0001	**0010**	
2	0010	**0011**	**0010**
3	0011	**0101**	
4	0100	**0110**	
5	0101	**0111**	
6	0110	**1000**	**1001**
7	0111	**1010**	
8	1000	**1011**	
9	1001	**1011**	
10	1010	**1100**	
11	1011	**1101**	
12	1100	**1101**	
13	1101		

If an error changes an edge such that a new path p from 1 to 13 with a length of 6 or 7 is created we are *not able* to detect this by counting steps. So the probability β that we are not able to detect an error by counting steps depends on the probability of the existence of a path like p. To calculate the probability β we define for an automaton T=(V, E, p) and every node $v \in V$ the sets $D_s(T,v) := \{n \mid \text{in T a path of length n from the starting state to v exists}\}$ and $D_f(T, v) := \{n \mid \text{in T a path of length n from v to a final state exists}\}$. For our example we have $D_f(T, 1) = \{6, 7\}$, $D_f(T, 9) = D_f(T, 8) = D_f(T, 10) = \{2\}$, or $D_f(T, 1)=D_s(T, 13)=\{6, 7\}$. For two sets $A, B \subseteq N$, we define $A+B:=\{x+y \mid x\in A, y\in B\}$. We further define $p^{T,f}_{v,x}$ as the probability that in T a final state can be reached from v in x steps. Analogously we define $p^{T,s}_{v,x}$. As said above a one-bit error changes an edge in the graph (V, E). For example, using the coding from Table 1, the edge from node 6 to node 9 may be changed to an edge from node 6 to node 1, 8, 11, or 13. Let us assume that an error has changed the edge from 6 to 9 to an edge from 6 to 1. We denote with $T_{(6,9)\to(6,1)}$ (or T' for short) the new automaton. More formally, $T_{(x,y)\to(x,y')}:=(V, E', p')$ where $E'=(E\setminus\{(x,y)\})\cup\{(x,y')\}$, $p'(e):=p(e)$ for all $e\in E'\setminus\{(x,y')\}$, and $p'((x,y'))=p((x,y))$[2]. We have $((D_s(T, 6) + \{1\})+ D_f(T', 1))\cap D_f(T,1)=\varnothing$. Hence we are able to detect this error by counting steps. Now let us assume the error has changed the edge from 6 to 9 to an edge from 6 to 8. Again let T' be the resulting automaton. Since the cut of $(D_s(T, 6)+\{1\})+D_f(T', 8)$ with $D_f(T,1)$ is the set $\{6\}$ there is a possibility greater than zero that we do not detect this error by counting. Since $p^{T',f}_{8,2}=1$, $p^{T,s}_{6,3}=p_2$ and $p((6,9))=p_4$ the probability that we do not detect this error by counting (under the condition that the edge change happens) is $p^{T,s}_{6,3}\cdot p((6,9))\cdot p^{T',f}_{8,2}=p_2p_4$. In our model an edge change happens with probability $(rm)^{-1}$. Hence in our example the edge change from (6, 9) to (6, 8) happens with probability 1/56 and therefore the probability that we do not detect this error by counting is $p_2p_4/56$. We denote such a probability by $p_{(6,9)\to(6,8)}$. So we have $p_{(6,9)\to(6,1)}=0$ and $p_{(6,9)\to(6,8)}= p_2p_4/56$. To calculate β we have to calculate $\sum_{\{\text{all edge changes}\}} p_{\{\text{edge change}\}}$. For our example we get $\beta=(7+2p_2p_3+p_1p_2)/56$. If we assume that $p_1=p_2=p_3=p_4=1/2$ then $\beta=31/224$ and $\alpha=193/224$ (86.2%).

In Table 2 we list all edge changes for our example where $p_{\{\text{edge change}\}}>0$.

Table 2: Edge changes with $p_{\{\text{edge change}\}}>0$.

Edge change	Probability $56\cdot p_{\{\text{edge change}\}}$	Edge change	Probability $56\cdot p_{\{\text{edge change}\}}$
<0001,0010>→<0001,0011>	1	<0110,1001>→<0110,1000>	$p_2p_3+p_2p_4$
<0010,0011>→<0010,0010>	$p_1p_2p_3+p_1p_2p_4$	<0111,1010>→<0111,1000>	p_1
<0010,0100>→<0010,0101>	p_2	<0111,1010>→<0111,1011>	p_1
<0011,0101>→<0011,0111>	p_1	<1000,1011>→<1000,1001>	p_2p_3
<0011,0101>→<0011,0100>	$p_1p_3+p_1p_4$	<1000,1011>→<1000,1010>	p_2p_3
<0100,0110>→<0100,0111>	p_2	<1001,1011>→<1001,1010>	p_2p_4
<0101,0111>→<0101,0110>	$p_1p_3+p_1p_4$	<1010,1100>→<1010,1101>	p_1
<0110,1000>→<0110,1010>	p_2p_3	<1011,1101>→<1011,1100>	$p_2p_3+p_2p_4$
<0110,1000>→<0110,1001>	$p_2p_3+p_2p_4$		

2.2. The Choice of the State Encoding c

[2] We assume that (x,y') is not in T. If (x,y') is in T, then T' is a multigraph. We omit the formal definition for this case.

The state encoding c of an automaton is important, because it determines – to some extend - the reliability of the automaton. To give reasons for this assertion we come back to our example. There we used the number of a state to encode it and got α=193/224. If we are able to find a c such that for all pairs of edges (x,v), (x,v') where h(c(v),c(v'))=1 (here h is the hamming distance) the property $((D_s(T, v) + \{1\})+D_f(T', v')) \cap D_f(T,1))= \varnothing$ is fulfilled (here T, T' are automaton and T' is the result of the edge change from (x,v) to (x,v') in automaton T, i.e. $T'=T_{(x,v)\text{->}(x,v')}$), then we get α=1, i.e. we are able to detect **all** effective errors by counting steps. We omit the proof of this fact here. For our example we show four state encodings c_1,c_2,c_3,c_4 with this property in Table 4.

Table 4: Examples with α=1

Node i	$c_1(i)$	$c_2(i)$	$c_3(i)$	$c_4(i)$
1	1110	0011	0001	0011
2	1000	0101	0111	0101
3	0001	1100	1110	1100
4	1101	0000	0010	0000
5	0111	1010	1000	1010
6	1011	0110	0100	0110
7	1100	0001	0011	0001
8	0000	1101	1111	1000
9	0101	1000	1010	1101
10	0011	1110	1100	1110
11	1111	0010	0000	0010
12	1001	0100	0110	0100
13	0100	1001	1011	1001

2.3. Notes on the Complexity of the Problem to find a Good c

The sets $D_s(T, v)$ or $D_f(T, v)$, $v \in V$, defined in Section 2.2 can be calculated in linear time using a slightly modified depth first search. For the calculation of $D_f(T', v)$ we have to be careful since T' is not necessarily acyclic and hence $D_f(T', v)$ is not finite. The good news is that we do not need to calculate $D_f(T',v)$ completely. For our purpose it is sufficient to find a cut with a finite $D_f(T,v)$. So, for a given coding c, we are able to calculate α in polynomial time. Hence the problem to find a state encoding c for a given automaton such that α is maximal (or β is minimal) is in **NP**. We suppose that this problem is **NP**-complete. We further suppose that it is also **NP**-complete to decide whether a state encoding with α=1 for an automaton exists.

2.4. Experimental Results

To experimentally determine the fault-coverage of the standard state encodings used in conjunction with microcode timing [8] we conducted fault-injection experiments. Table shows the main parameters for the fault-injection.

Table 3: Fault-injection Parameters

Parameter	Value
Fault type	Transient
Fault rate	10^{-5} [2]
Executed μOps	10^{6}
Distribution	Exponential

The parameters were used as inputs for a control-flow simulator of a microcontrolled automaton. The simulator supports different state encodings such as binary, Gray, One-Hot and the proposed state encoding. microprograms can be generated or loaded. An architecture file describing important microcode-ROM parameters such as width, size, control-flow parameters (μPC position and width, number and position of control bits, etc.) must be loaded prior to be able to execute microprograms. A concrete architecture file and microprograms from the microprogram simulator JMic, developed at the Technical University of Munich and an architecture file resembling a RISC-style control automaton was developed. Figure 3 shows the fault coverage of Microcode Timing. Only generated microprograms in dependence from two important parameters (1) the number of paths with unequal length and (2) the number of branches were examined. The maximal number of microinstructions within a microprogram was assumed to be 16. Note that different meanings for the y-axis are used. The first meaning is the fault coverage in % (detected), the second is the number of overwritten, latent and effective faults. The x-axis shows a combination from unequal path lengths within a microprogram (between 0 and 0.99) and the branch probability, varied between 0.5 and 0.1 over time. Note that time is a non-linear factor since faults manifest randomly over time. For a high probability of equal path lengths and branches, the (negative) peaks in fault coverage result. The states within the state automata are binary encoded states.

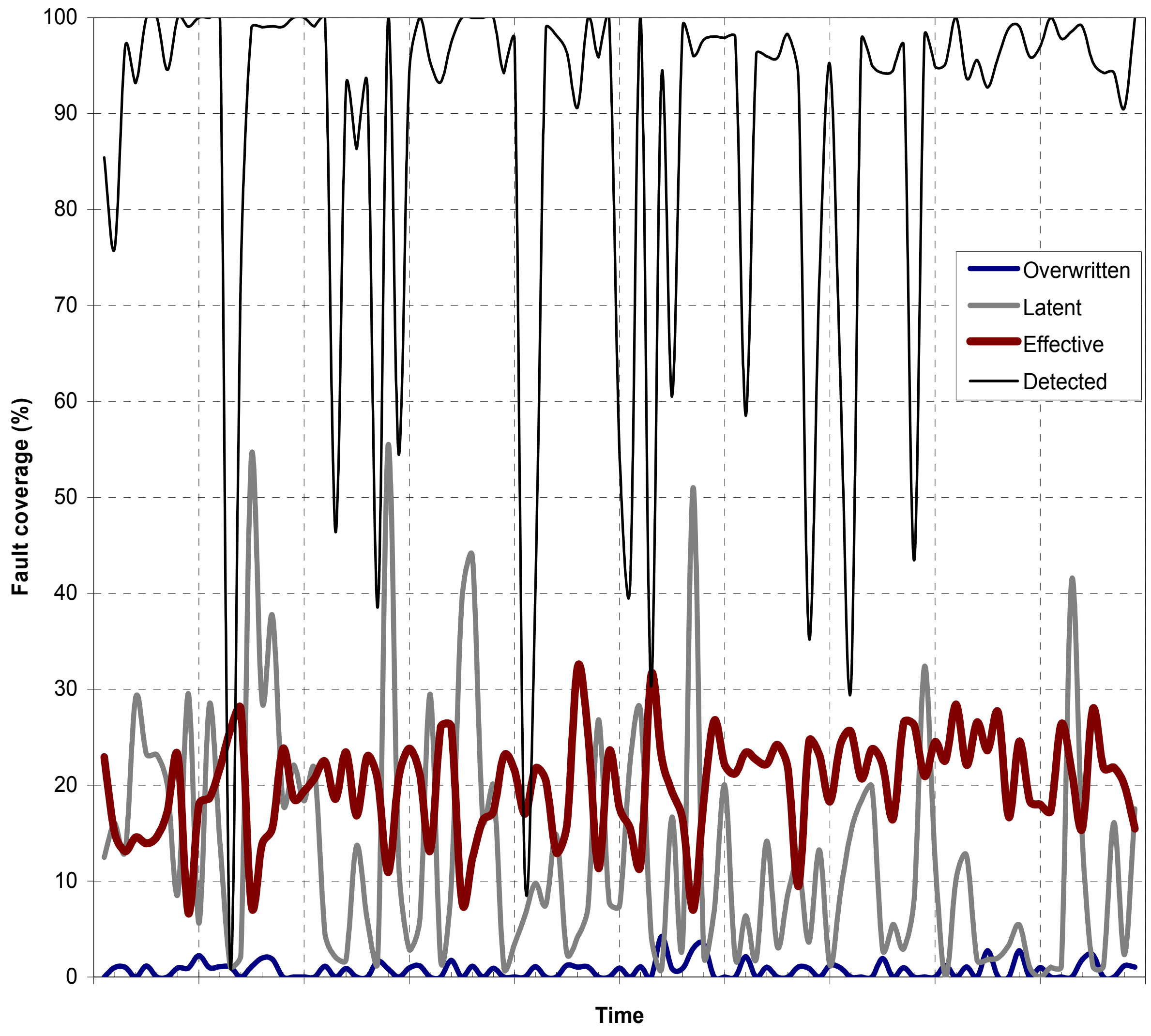

Figure 3: Fault coverage for Microcode Timing

3.Future Work and Conclusion

The usage of state machines is very popular. Since they can consist entirely out of flip-flops, they are must be protected against Single Event Upsets. Due to their broad usage and vulnerability against SEUs, fault-tolerance mechanisms must be implemented. Microcode timing offers the possibility to detect a fault before it will get effective, manifesting in the architectural state, furthermore third party state machines can be secured without changing the state machine. The execution of a single microinstruction or a whole microprogram is possible, since timing information can be individually computed for both cases. The percentage of overwritten, latent and effective faults was determined as 0.71%, 12.9% and 20.1% of all injected faults. The average fault coverage was determined to be 88%. Our future work will include the fault-coverage computation of the proposed encoding and a reduction of the high fault coverage variance on a high probability of different path lengths/branches.

References

[1] R.W. Wieler, Z. Zhang, R.D. McLeod, *Simulating static and dynamic faults in BIST structures with a FPGA based emulator*. In Proc. of IEEE International Workshop of Field-Programmable Logic and Application, pp. 240-250, 1994.

[2] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, L. Alvisi. *Modeling the effect of technology trends on soft-error rate of combinational logic*. Int'l. Conference of Dependable Systems and Networks, June 2002.

[3] Thomas H. Cormen, Charles Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms.* MIT Press, 2nd edition 2001, ISBN 0262531968

[4] R. Katz, R. Barto, and H. Tiggeler: *Sequential Circuit Design for Spaceborne and Critical Electronics*, Mil/Aero Applications of Programmable Logic Devices (MAPLD) International Conference, 2000.

[5] R. Katz,, J. Wang, J. McCollum, and B. Cronquist: *The Impact of Software and CAE Tools on SEU in Field Programmable Gate Arrays*, IEEE Transactions on Nuclear Science, December, 1999.

[6] F. Gray, *Pulse Code Communication*, U. S. Patent 2 632 058, March 17, 1953.

[7] B. Fechner, J. Keller, A. Wohlfeld. Web Server Protection by Customized Instruction Set Encoding. In *Proc. 11th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, Rhodes Island, April 2006.

[8] B. Fechner. Microcode with Embedded Timing Constraints. In *Proc. ARCS '06 Workshop on Dependability and Fault Tolerance*, Frankfurt, March 2006, pp. 45-51, GI 2006.