High-Level Test Generation for Processing Elements in Many-Core Systems

Adeboye Stephen Oyeniran, Raimund Ubar, Siavoosh Payandeh Azad, Jaan Raik
Tallinn University of Technology, Estonia

Abstract—**The advent of many-core system-on-chips (SoC) will involve new scalable hardware/software mechanisms that can efficiently utilize the abundance of interconnected processing elements found in these SoCs. These trends will have a great impact on the strategies for testing the systems and improving their reliability by exploiting system's re-configurability to achieve graceful degradation of system's performance. We propose a strategy of Software-Based Self-Test (SBST) to be used for testing of processing elements in many-core systems with the goal to increase fault coverage and structuring the test routines in a way which makes test-data delivery in many-core systems more efficient. A new high-level fault model is introduced, which covers a broad class of gate-level Stuck-at-Faults (SAF), conditional SAF, and bridging faults of any multiplicity in processor control paths. Two algorithms for high-level simulation-based test generation for the control path and a bit-wise pseudo-exhaustive test approach for data path are proposed. No implementation details are needed for test data generation. A novel method for proving the redundancy of high-level functional faults is presented, which allows for precise evaluation of fault coverage.**

Keywords: ***processor core testing, high-level control faults, test generation, high-level fault coverage, fault redundancy***

I. INTRODUCTION

Technology scaling trends and the advent of many-core chips suggest that communication, not computation, will dominate delay, area and power budgets in future computing systems. The shift to communication-centric focus on many-core architectures will involve new scalable hardware/software mechanisms that can efficiently utilize the abundance of interconnected processing elements found in these new architectures. These trends will have a great impact on the strategies for testing the systems and improving their reliability by exploiting system's re-configurability as a mechanism for providing graceful degradation of system's performance. In modern many-core system-on-chips, obtaining an overall overview of the system's health is becoming a more pressing issue for resource management and application re-mapping.

Maintaining a global view of the system health requires a testing and monitoring strategy that covers system's processing elements, memory and interconnection infrastructure along with a mechanism for diagnostic information propagation to system's kernel. Using the diagnostic information obtained from different sources, system's kernel can perform more optimal reconfiguration and application mapping decisions. This in turn, will positively contribute to maintaining a graceful degradation of system performance during its lifetime.

There are many challenges for the architecture design of many-core processors, one of which has the most serious concern is manufacturing yield because an IC's profitability depends heavily on it [1]. With the ever-increasing circuit density, obtaining high fabrication yield solely through improving the manufacturing process is increasingly difficult and will become unaffordable in the near future. A more practical solution is to provide defect tolerance capabilities on-chip by incorporating redundant circuits. Previous attempts in this domain mainly focused on introducing microarchitecture-level redundancy [2, 3]. This is appropriate for multicore processors with a restricted number of cores in order to keep the hardware overhead small. The situation is different in the case of many-core processors when the number of cores increases to a point when a single core becomes inexpensive compared to the entire processor. In this case, it is not necessary to tolerate defective cores at the microarchitecture-level, and it will be more appropriate to employ core-level redundancy to reduce the complexity associated with microarchitecture-level redundancy.

More pronounced aging effects (wear-out), process variability, more frequent early-life failures, and incomplete testing or verification due to time-to-market pressure in new fabrication technologies impose also reliability challenges on forthcoming systems [4].

A promising solution to these reliability challenges is concurrent on-line self-test of computing cores and self-reconfiguration of system on chips [5-7]. Core self-test can be performed concurrently with applications executing normally. Concurrent on-line test (COLT) exploits the massive structural redundancy of multi- and many-core architectures by shutting down some subset of cores within the SoC for testing while the remaining cores run user applications as normal. This allows the system to achieve its reliability requirements and maintain an extremely high level of availability with minimum application intrusion. A prerequisite of that is the availability of core self-test routines which should provide high fault coverage at minimum testing time cost.

978-1-5386-3344-1/17/$31.00 ©2017 IEEE

Software-Based Self-Testing (SBST) [8-10] is an emerging new paradigm in the testing domain, which relies on the exploitation of existing available resources resident in the system. The SBST approach is based on software programs that are designed to test the functionality of the processor cores. The major cost of SBST is the time overhead incurred by the execution of the test routines. The hardware overhead is either non-existent, or negligible, and no Instruction Set Architecture (ISA) extensions are required. An important aspect of on-line core testing in SoCs is the scheduling strategy of the test process. One approach is to periodically initiate testing on all system cores simultaneously [11, 12] causing the entire system to be offline, thereby interrupting the execution of application programs. Another approach is to initiate testing on individual cores that have been observed to be idle for some time [13]. In this case, the testing process is minimally intrusive, but the time required to complete the test for all cores is longer. Testing may also be selective, targeting cores that have experienced prolonged stressing due to high utilization.

Recent research results in periodic organization of online testing of many-core processors are presented in [4] to facilitate autonomous detection and omission of faulty cores which makes graceful degradation of the many-core architecture possible. In [14], test-data delivery optimization algorithms are developed for SoC designs with hundreds of cores, where a network-on-chip (NoC) is used as the interconnection fabric.

In this paper, we focus on the quality of the core self-test in terms of increasing the fault coverage, minimizing test length and producing well-structured test routines to ease the diagnosis and test-data delivery around the SoC. The memory and interconnect testing remain beyond the scope of the paper. Also, we will not target the problem of organization and scheduling of test sessions in SoC as a whole.

The rest of the paper is organized as follows. In section 2, we present the state-of-the-art of core testing. In section 3, a method of high-level fault modeling is proposed, and in section 4, the problem of mapping high-level faults into the gate-level faults is investigated. Section 5 presents two new methods for test generation of the control parts of MP cores, and in section 6, a method for proving the redundancy of high-level functional faults is proposed. Section 7 discusses a method for high-level testing of the data path. In section 8, we present the structure of the test routines delivered to the cores of SoC, section 9 presents experimental data, and section 10 concludes the paper

II. STATE OF THE ART

For the last decade, there has been an extensive research on SBST of processors [8,15-18]. The quality of SBST is mainly affected by test data used in test programs. One of the ways to obtain test data is executing an Automated Test Pattern Generator (ATPG). In [16] it was shown that the processor can be divided into Modules under Test (MUT) to ease the task of ATPG. On the other hand, the difficulties of the method arise from the need of guiding ATPGs by functional constraints to produce functionally feasible test patterns. An alternative way is to use random test patterns for MUTs [17]. In [18], shifting of SBST generation from gate-level to Register-Transfer Level (RTL) was suggested. The drawback of this method is that it has not been shown how to achieve high fault coverage of low-level structural faults. Hybrid SBST methods were proposed for combining deterministic structural SBST with verification-based self-test codes [8, 18-20]. In addition to Hybrid SBST [20, 21], there are methods that achieve comparable results and improve scalability when generating SBST program using only RTL [19, 22].

The drawbacks of the known methods are: the fault coverage is traditionally measured only with respect to SAF, no broader fault classes have been considered, and no attempts have been made to evaluate the test quality regarding covering of multiple faults with avoidance of fault masking.

To cope with the complexity of gate or RT level representations of microprocessors (MP), we consider the problem of SBST program generation with focus on modeling functional faults at the behavioral-level using Instruction Set Architecture (ISA). At the same time, for the purpose of evaluating the quality of tests generated at high-level, we target a broader class of faults than SAF. We consider the conditional SAF and bridging faults as well.

Our previous work has been targeting SBST field with the methodology of using High-Level Decision Diagrams (HLDD) for diagnostic modeling of MP at the behavior level [23-25]. For test program generation, a hierarchical approach was used where the control functions of MP were tested exhaustively (by conformity test), but the data operands for testing the data path were generated by gate-level ATPG (scanning test). The paper [25] was devoted to the generation of test groups for detecting multiple faults and avoiding fault masking. However, in [25] only the multiple faults in the READ/WRITE logic were considered.

In [24] it was shown that the high-level functional fault model for the control part of MP, based on HLDDs, can be mapped well on the related low-level faults of a joint class of the stuck-at faults (SAF), conditional SAF and bridging faults. Moreover, it was proven that the conformity test with 100% high-level fault coverage would be able to detect also any multiple gate-level fault from the mentioned joint single fault class.

In [26], a method for high-level functional fault simulation for microprocessors was proposed. Based on this method it became possible to measure the quality of test programs also with respect to high-level fault coverage. It was shown that the simplified deterministic high-level test generation method developed in [24] for testing the control faults was not able to achieve 100% high-level fault coverage. We have established two reasons for that: (1) the redundant

high-level faults were not identified, and (2) the proposed deterministic method was not able to handle so-called "hard-to-test high-level faults".

In this paper, we extend the previous results [24, 26] by proposing a new test generation method for testing control faults in processor cores of digital systems, which is able to better handle "hard-to-test high-level faults". We also propose a method for proving possible redundancy of not detected high-level faults.

III. HIGH-LEVEL FAULT MODEL FOR PROCESSOR CORES

In this paper, we focus on testing of the ALU module of processor cores. Consider a typical set of instructions in Table 1. The operations are represented by operation codes and the related formulas f_i for calculation of the output values of the ALU. The high-level structure of an ALU is depicted in Fig 1. The control variable c can have values from the domain {0,1, …, 15}. Denote by I_i the instruction (with opcode c_i) which performs the function f_i in ALU.

Table 1. *Instruction subset for an ALU of a microprocessor*

Mnemonic	f_i	*Opcode c*	Mnemonic	f_i	*opcode*
MOV	f_0	0000	SHL	f_8	1000
ADD	f_1	0001	SHR	f_9	1001
SUB	f_2	0010	ASR	f_{10}	1010
CMP	f_3	0011	INC	f_{11}	1011
AND	f_4	0100	DEC	f_{12}	1100
OR	f_5	0101	RLC	f_{13}	1101
XOR	f_6	0110	RRC	f_{14}	1110
NOT	f_7	0111	NOP	f_{15}	1111

Represent the instruction set in Table 1 by the high-level structural circuit and High-Level Decision Diagram (HLDD) [23-25] in fig 1. The HLDD has a single decision node labeled by the control variable c and 16 terminal nodes labeled by the functions fi implemented in the ALU data path and selected by instruction fi respectively. The node c represents the whole control part of the ALU. In general case, if the system is described by more control variables (representing control fields of the instruction word, register addresses, flags, conditions etc.), the internal structure of the HLDD will be more complex as well [25].

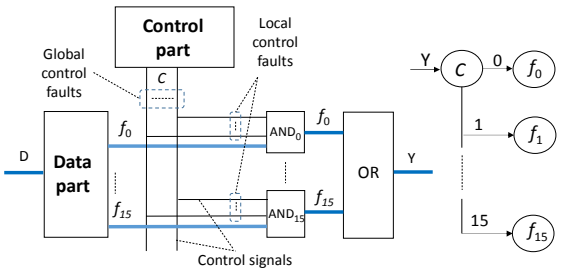

Fig.1. *Behavior level structure of ALU and its HLDD*

The value of the graph variable Y is calculated by traversing the HLDD from the root node to the terminal nodes. In the current example, the value of c decides the direction of traversing. If $c = i$, we will have $Y = f_i$

According to such a mapping of the system functions into the HLDD model, we can classify two types of HLDD-based high-level functional fault models: *control faults* (the faults related to the non-terminal nodes), and *data faults* (the faults related to the terminal nodes).

Definition 1. As the fault universe related to non-terminal nodes we allow any corruption in the behavior of the nodes be defined as follows [23]:

(1) the output edge of the node is broken (the control signal SAF $c_i \equiv 0$, or at the bit level SAF $c_{i,k} \equiv 0$ for controlling function in the bit k);

(2) the output edge of a node is always activated (SAF $c_i \equiv 1$, or SAF $c_{i,k} \equiv 1$);

(3) instead of the activated edge, another edge or a set of edges are at the same time erroneously activated

The last most general fault can be notated as a conditional SAF ($c_j \equiv 1/c_i$), or as ($c_{j,k} \equiv 1/c_i$), This fault type may be caused by bridging fault, other line coupling faults, or it may be explained by more complex physical defects of the control line c_j (or $c_{j,k}$). □

Denote by m the internal node under test in the HLDD, and by $M^T(m)$ the set of terminal nodes which can be reached from the node m. Let $f(m^T)$ denote the expression labeling the terminal node $m^T \in M^T(m)$. In [24] the following constraints (as part of the fault model) were introduced for testing the control part of MP:

$$\forall m^T \in M^T(m): [f(m^T) \neq \Omega)], \quad (1)$$

$$\forall m_i, m_j \in M^T(m),\ i \neq j: \forall k\ [f_k(m_i) < (f_k(m_i) * f_k(m_j))] \quad (2)$$

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [27]. Here, ZERO denotes a binary vector (00…0), and, similarly, ONE stands for (11…1). The index k refers to the bit number of the data words.

Let us focus here only on the case: Ω = ZERO, * stands for OR. Since $f_k(m_i) < (f_k(m_i) * f_k(m_j))$ is valid always if $f_k(m_i) < f_k(m_j)$ is valid, and since in the bit-based analysis the constraint (1) can be satisfied indirectly from (2), we can simplify the constraints (1) and (2) as follows:

$$\forall m_i, m_j \in M^T(m),\ i \neq j: \forall k\ [f_k(m_i) < f_k(m_j)] \quad (3)$$

From Definition 1, the high-level fault model *for non-terminal nodes* results, which leads in a natural way to exhaustive testing of control modes of MP. In other words, the fault model of a non-terminal node leads to exercising of the set of all possible values of the node variable, accompanied with the constraints (3) for data operands to be satisfied during the exhaustive test of the node.

The number of functional faults for the non-terminal node m can be calculated as $N(m) = n_m(n_m - 1)k$ where n_m – is the number of output edges of the node m, and k – is the length of the data word. In our example, we have $N(c) = 16*15*k = 240*k$.

Definition 2. As the high-level fault universe for terminal nodes f_j., we use the functional fault model defined as the sets of test data $D(f_j)$ needed for testing the functions f_j □

The size of the data path high-level fault model is measured as the number of faults $N(f) = \Sigma_j |D(f_j)|$ where $|D(f_j)|$

is the number of test data for testing the sub-circuit of ALU, responsible for the ALU operation f_j.

From the one-to-one correspondence between the functional faults in the Data Path and the tests targeting these faults, the following results.

Corollary 1. The length of the full ALU data path test $T(f)$, in terms of the number of instructions involved as test objectives, is equal to the number of functional faults in the ALU data path $T(f) = N(f)$ □

Proof. The proof results from the principle of how we test the data path.

Note, the number of high-level functional faults in the data path is not predefined before test generation, rather it is the byproduct of the test generation procedure. The quality of the ALU data path test is measured by low-level simulation to calculate the low-level fault coverage.

The situation will be different for the ALU control part where the high-level functional fault coverage may have even broader meaning and importance than the traditional SAF coverage. As we will see in the next Section, by high-level fault coverage it becomes possible to evaluate the low-level test quality for a broader class of faults than SAF, including multiple faults.

IV. Relationships between High- and Low-Level Fault Models

Consider now the following statements about the quality of test programs synthesized according to the fault models described in the previous Section for control and data parts of the ALU. For simplicity, consider in the following discussion the simplified case of HLDD in Fig.1. Denote $f_k(m_i)$ in (3) for brevity as $f_{i,k}$.

Introduce the following notations. Let F – be the set of functions at the terminal nodes reachable from the node c under test, $d(f_i)$ – a group of data operands as arguments for the function f_j, and $D(f_i)$ – a set of data groups $d(f_i)$. Denote by $T(c_i)$ a test which consists in repeating the instruction I_i for all groups of data operands in $D(f_i)$.

Theorem 1. The control test $T(c_i)$ will detect all gate level conditional SAF faults $\boldsymbol{c_{j,k} \equiv 1/c_i}$ for all $j \neq i$, and SAF $\boldsymbol{f_{i,k} \equiv 1}$, iff for each pair (i,j) and each bit k, there is at least one $d(f_i) \in D(f_i)$ which satisfies the set of constraints.

$$\forall f_j \in F,\ _{i \neq j}: \forall k\ (f_{i,k} < f_{j,k}) \qquad (4)$$

Proof. Consider the ALU in Fig.2 divided into control and data paths. Let us focus on testing of the ALU control part, using the instruction I_i which produces the opcode $c = i$, and assigns the values $c_i = 1$ ($c_{i,k} = 1$ for all k) for all related control signals, and activates the operation f_i: $Y = f_i$. This is the test for checking if the output edge $c = i$ of the node c is not broken in the HLDD in Fig. 1 (no fault of type $c_{i,k} \equiv 0$), according to the fault model in Definition 1. The edge is working correctly if the non-zero correct value of f_i propagates from the output of data path to the output Y of the control path (see Fig.2). However, according to the fault model in Definition 1, there may be present another conditional SAF fault $c_{j,k} \equiv 1/c_i$ on other edges of the HLDD which may mask the faults $c_{i,k} \equiv 0$.

Assume now that there is a fault $c_{j,k} \equiv 1/c_i$, and the data operands are selected so that $f_{j,k} = 1$. If the constraint (4) for the bit k is satisfied, i.e. if $f_{i,k} < f_{j,k}$, the fault $c_{j,k} \equiv 1/c_i$, will be detected. Moreover, from the constraints $f_{k,i} < f_{k,j}$, it results that the only way to satisfy this constraint is to assign $f_{i,k} = 0$, and $f_{j,k} = 1$, which means that the faults $\boldsymbol{f_{i,k} \equiv 1}$ will be as well detected. □

To satisfy the constraint (4), a set $D(c_i)$ of several data operands may be needed. Hence, the test $T(c_i)$ for the instruction f_i will consist of repeating the instruction $|D(c_i)|$ times with all data operands in $D(c_i)$.

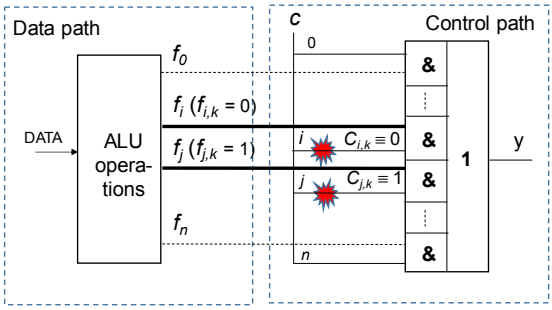

Fig.2. *Behavior level structure of ALU and its HLDD*

Definition 3. Denote the full ALU control test as $T(c)$ which consists of all tests $T(c_i)$ for all functions $f_i \in F$. The length of the test $T(c)$ in terms of the number of instructions to be tested is $N(c) = \Sigma_i |D(c_i)|$

Corollary 2. From Theorem 1, it results that by synthesizing the test $T(c)$ all conditional SAF faults $c_{i,k} \equiv 1/c_j$ for all $j \neq i$, and all SAF $f_{i,k} \equiv 1$, in the control part, will be detected. □

From Theorem 1, it results that the condition (4) is not sufficient for testing the SAF faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$ that is needed for testing the correctness of the output edge $c = i$ of the node c in the HLDD in Fig. 1, which was the basis of synthesizing the test data for $T(c_i)$. The same is valid in accordance to Corollary 2 for all other $j \neq i$.

However, this deficiency of the test $T(c_i)$ is not critical, as it can be concluded from the following statement.

Corollary 3. SAF faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$ in the control part of ALU will be detected by the ALU data path test $T(f_i)$ for the instruction I_i as a byproduct.

Proof. The proof is straightforward. The test $T(f_i)$ consists in repeating the same instruction f_i with different data from $D(f_i)$. But any data in $D(f_i)$ which produces the values $c_{i,k} = 1$ creates the situation where both values, $f_{i,k} = 1$ and $c_{i,k} = 1$ are supporting mutually the propagation of faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$, to the output Y. □

Definition 4. Define the full ALU data path test $T(f)$ as the set $\{T(f_i)\}$ of all tests $T(f_i)$ for the ALU functions $f_i \in F$, respectively, and define the full ALU test as a sum $T_{ALU} = T(c) + T(f)$.

From Corollary 1 and Definition 3 the following results:

Corollary 4. The length of the full ALU test T_{ALU} is $N_{ALU} = N(c) + N(f)$.

In [24] it was shown that the test sequences which satisfy the constraints (1) and (2) will cover a broad class of single faults including Stuck-at Faults (SAF), conditional SAF and bridging faults. It is easy to see that the same is valid also for the constraints (4) used in Theorem 1, which are easier to simulate.

Note that to create final test program, each instruction $I_i \in T(c) \cup T(f)$ with related data operands $op \in D(f_i) \cup D(f_i)$ implies a sequence of instructions which consists of the initialization phase of loading the data operands into the proper registers, testing phase of applying the instruction I_i, and final phase of storing the test result.

V. Generating Data for Control Path Test

From above, it follows that the fault model defined by the set of constraints in (4) can be interpreted as the definition of the universe of high-level faults. A direct impact of this interpretation is the possibility of evaluating the high-level functional fault coverage as the percentage of constraints in (4) for the given test. The fact that the faults $f_{i,k} \equiv 0$ and $c_{i,k} \equiv 0$ in the control path will not be taken in the fault universe of the control faults can be overseen because, according to Corollary 3, these faults will be covered anyway as the byproduct by the data path test $T(f)$.

In the following, we present an algorithm for generating the set T(c) of test data for testing the control path.

Algorithm 1: RANDOM test data generation for ALU

Input: Instruction set of the processor
Output: Sets of test operands OP_i for each instruction, and fault table D
Notations: n – number of instructions (functions F_j), op – test operand, OP – current set of selected random test operands, $f_j(op)$ – the result of the instruction I_j for the operand(s) op, D – fault table, D_{ij} – w-bit entry in D (w – length of the data Word).

1 Initialize $OP = \varnothing$
2 Generate a set of R random operands
3 **for** $i = 1, \ldots, n$
  ***generation of operands for instruction I_i
4 Initialize $OP_i = \varnothing$,
5 **for** $j = 1, \ldots, n$ ($j \neq i$)
    ***operands for solving constraints $f_{i,k} < f_{j,k}$
6 Initialize $D_{ij} = 0$
7 **for** all $op \in R$ **while** $D_{ij} \neq 0$
      ***adding new operands for covering D_{ij}
8 $D_{ij}(op) = f_j(op) \wedge (f_i(op) \oplus f_j(op))$
      *** calculating fault coverage for op
9 **if** $(D_{ij}(op) \vee D_{ij}) \oplus D_{ij} \neq 0$ **then**
        *** check for the coverage increment
10 **begin**
11 $D_{ij} = D_{ij} \vee D_{ij}(op)$
        *** update of the coverage vector
12 Include op into OP_i
        *** new operand is selected
13 **end**
14 **endfor** op
15 **endfor** j
16 **endfor** i

The result of Algorithm 1 will be a set of operands OP_i for each instruction I_j, and the fault table $D = \| D^k_{ij} \|$ where $D^k_{ij} = 1$ means that the functional fault described by the constraint $f_{i,k} < f_{j,k}$ is covered at least by one operand $op \in OP_i$, otherwise $D^k_{ij} = 0$. The percentage of 1s in D is the high-level functional fault coverage of the test for control path.

The Algorithm 1 is called RANDOM since in each step of 7, the first random operand $op \in R$ will be chosen which produces an increase in the fault coverage, no matter how big it is. To reduce the test length for testing the control path we implemented another algorithm called GREEDY.

Algorithm 2: GREEDY test data generation for ALU

GREEDY algorithm differs from RANDOM in running the step 7 before selecting the operand the whole search space R of random operands to calculate the fault coverage increase for all $op \in R$, and only then selects the best one which produces the maximum increase in fault coverage. Then the next operand is selected in a similar way. The step 7 ends when the goal of $D^k_{ij} = 1$ is reached, or no more operands can be selected to satisfy all constraints $f_{i,k} < f_{j,k}$ □

The constraints $f_{i,k} < f_{j,k}$ may not be solved for two reasons: either the related functional fault is redundant, or the search space R is not big enough.

The proof of redundancies of high-level faults introduced in this paper is easy compared to the proof of fault redundancies at gate-level.

VI. Redundancy Proof of High-Level Faults

Consider Table 2, which illustrates the high-level fault coverage D, which was generated by the algorithms in the previous Section for a subset of instructions in Table 1.

In Table 2, the 0s may refer to possible redundancies for the functional faults related to the constraints $f_{i,k} < f_{j,k}$ where i and j correspond to the rows and columns, respectively. All 0s for a D_{ij} refer to the high probability of redundancy of the related fault, i.e. that the constraint $f_i < f_j$ is not possible to satisfy. In most cases of ALU operations, it is very easy to demonstrate this type of redundancy. For example, if i refers to AND operation and j refers to OR, it is straightforward that $(a \vee b) < (a \wedge b)$ can never happen. In Table 2, all 0-s refer to the redundant high-level faults.

Table 2. *Example of a High-Level Fault table*

	f_1 (MOV)	f_2 (ADD)	f_3 (SUB)	f_4 (CMP)	f_5 (AND)
f_1 (MOV)		111111	111111	111111	000000
f_2 (ADD)	11111		111110	111111	111111
f_3 (SUB)	11111	111110		111111	111111
f_4 (CMP)	11111	111111	111111		000000
f_5 (AND)	11111	111111	111111	111111	

In cases when there is 0 in D_{ij} which refers to the case of no solution for $f_{i,k} < f_{j,k}$, only in a single bit k, or few bits, we can suggest for the proof a method which can be called as "*partial truth table method*". The idea of the method stands

in showing the equivalence of partial truth tables (or impossibility to solve the constraint) for the functions involved, where as few as possible responsible bits should be selected for the need of proof. Let us consider few examples for the redundancy proof possibilities for the set of ALU functions in Fig. 1.

In Table 3, examples are shown for 4 partial truth tables for the functions SUB, ADD, OR, AND and for the bit k, where the columns 00, 01, 10, 11 represent the values of the data variables in the bit k. For SUB and ADD, the equivalence of the behavior in the given bit is demonstrated, and in the case of OR and AND, the missing of solution for (4) is highlighted.

Table 3. *Examples of redundancy proofs with 1-bit truth tables*

No	$f_{i,k} < f_{j,k}$	D_{ij}	$f_{i,k}/f_{j,k}$	00	01	10	11
1	SUB < ADD	1…11**0**	SUB	0	1	1	0
			ADD	0	1	1	0
2	OR < ADD	1…11**0**	OR	0	1	1	1
			ADD	0	0	0	0

It is easy to show the equivalence of operations ASR and SHR for all bits, except the most significant bit (MSB). Hence, for all the bits k except for MSB, we can prove that the value $D^k_{ij} = 0$ refers to the redundant faults.

In some cases of proof the partial truth method will not work, because the results of operations may depend substantially on all bits of the word like for increment and decrement. When this happens, specific corner cases should be found for the proof. For example, to proof the equivalence of increment and decrement operations in the least significant bit, the operand 1…110 should be used, where both instructions INC and DEC produce the same result "all 1s".

VII. Generating Operands for Data Path Test

For testing the data path, we can use the pseudo-exhaustive testing approach instead of exploiting traditional gate-level ATPG [28]. Using pseudo-exhaustive data makes the test generation procedure not depending on the implementation details of the processor cores under test.

The method of pseudo-exhaustive test generation lays on the idea of testing the operations in all bits independently of other bits.

Since the logic operations are substantially independent in all bits, we can apply true exhaustive approach for testing, using only 4 exhaustive patterns {(0,0), (0,1), (1,0), (1,1)} per bit. For unary operations like shifts or moves, only two patterns are sufficient. Examples of the pseudo-exhaustive test data for addition and subtraction operations are shown in Tables 4 and 5. Here the cases of ripple carry for addition and ripple borrow for subtraction are used. In general case, e.g. at carry-ahead-addition, the method of pseudo-exhaustive approach will be more complex, and more test data will be needed.

In Tables 4 and 5, all bits of all ALU operations f_i, i = 0,1,2,…., are tested exhaustively. For ADD and SUB operations, 8 data pairs are needed to cover all combinations of 3 inputs (two operands and carry/borrow bit) of each bit of the adder (sub-tractor). We start to generate patterns from the least significant bits, calculate the carry c_i for the next bit and fit for the next bit the values of operand bits a_i and b_i with the calculated carry c_i, so that all exhaustive combinations for this bit section were achieved. In such a way created patterns for the 2nd bit and 1st bit can be "copy-pasted" for the next two-bit sections to right.

Table 4. *Pseudo-exhaustive test data for addition operation*

No	…	4-bit $a_4\ b_4$ c_4	3-bit $a_3\ b_3$ c_3	2-bit $a_2\ b_2$ c_2	1-bit $a_1\ b_1$ c_1	0-bit $a_0\ b_0$ c_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	0 1 0	0 1 0	0 1 0	0 1 0	0 0 1
3	…	1 0 0	1 0 0	1 0 0	1 0 0	0 1 0
4	…	1 1 0	0 0 1	1 1 0	0 0 1	0 1 1
5	…	0 0 1	1 1 0	0 0 1	1 1 0	1 0 0
6	…	0 1 1	0 1 1	0 1 1	0 1 1	1 0 1
7	…	1 0 1	1 0 1	1 0 1	1 0 1	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

Table 5 *Pseudo-exhaustive test data for subtraction operation*

No	…	4-bit $a_4\ b_4$ c_4	3-bit $a_3\ b_3$ c_3	2-bit $a_2\ b_2$ c_2	1-bit $a_1\ b_1$ c_1	0-bit $a_0\ b_0$ c_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	1 1 0	0 1 1	1 1 0	0 1 1	0 0 1
3	…	0 0 1	1 0 0	0 0 1	1 0 0	0 1 0
4	…	1 0 0	1 1 0	1 0 0	1 1 0	0 1 1
5	…	0 1 1	0 0 1	0 1 1	0 0 1	1 0 0
6	…	1 0 1	1 0 1	1 0 1	1 0 1	1 0 1
7	…	0 1 0	0 1 0	0 1 0	0 1 0	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

VIII. Composition of Test Routines for Cores

In accordance to the described method of generating test data for the testing the processor, we can divide the test patterns into two parts: (1) *conformity test patterns* which target the control faults, and (2) *scanning test patterns* which target the data faults of the core. Each test pattern $T_{i,j} = (I_i, d_{i,j,1}, d_{i,j,2})$ consists of an instruction pattern I_i, = ($opcode_i$, $A_{i,j,1}$, $A_{i,j,2}$) and two data operands d_{j1}, d_{j2} with addresses $A_{i,j,1}$, $A_{i,j,2}$, respectively. Denote for each instruction I_i the numbers of data operand pairs by c_i and s_i, for the conformity and scanning test parts, respectively. Since each instruction under test should be executed for all related conformity and scanning test patterns, we can represent the all test information as a set of $n + 1$ arrays (n is the number of instructions under test): the array I of n instruction patterns, and n data arrays with $n_i = c_i + s_i$, data patterns (operand pairs) each (see Fig.3).

From above, a test program structure results, which will consist of n loops where n is the number of tested instructions. The body of the i-th loop will consist of (1) the initialization sequence of instructions for loading the data d_{j1}, d_{j2} into the registers involved in the instruction I_i, (2) executing the instruction I_i, under test, and (3) observation sequence.

The whole test can be compressed by representing it as a program template consisting of two embedded loops, and n + 1 data arrays as program parameters: an array of n instructions, and n data arrays with $n_i = c_i + s_i$, data patterns (operand pairs) each (see Fig.4).

The first loop consists of test program cyclic accesses using indirect addressing mode to the array of instructions under test. In the body of the second internal loop of the test program, the data operands will be cyclically prepared for use by the current instruction under test. Observation and analyzing of test results can be implemented by software signature analyzer.

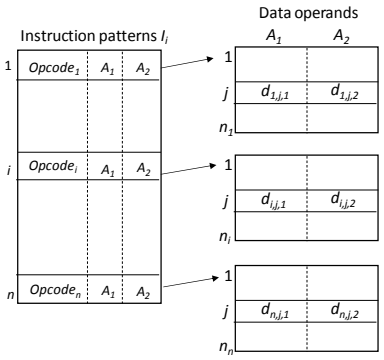

Fig.3. *Structure of the test information for self-test of cores*

For all instructions I_i, $i = 1,2, \ldots n$
  For all data operands $(d_{i,j,1}, d_{i,j,2}), j = 1,2, \ldots n_i$
    Read $d_{i,j,1}$
    Read $d_{i,j,2}$
    Execute the instruction $I_i, = (opcode_i, d_{i,j,1}, d_{i,j,2})$
    Write the test result in signature analyzer
  End for data
End for instructions

Fig.4. *Test program template for self-test of cores*

The originality of the test strategy stands in on-line test generation method based on modifying on the fly the stored test program template which uses hierarchically organized test data.

The presented test program with embedded two loops and the hierarchical structure of the test information in form of instruction and data operand arrays allow splitting the test information to be used in the template in Fig. 4 in arbitrary ways into different segments (by specifying different values of n). This provides high flexibility for delivering test data for cores under test in the system and gives in such a way better possibilities for optimization of scheduling strategies of the test process at given constraints of currently running applications.

IX. Experimental Results

The motivations of high-level test generation are threefold: generating test programs without knowing the details of circuits' implementations, speeding up the generation procedure, and increasing the quality of tests due to the possibility of covering multiple low-level faults without their counting. We carried out experiments with ALU sub-circuit of the VLIW processor [29].

The experimental results for the two proposed algorithms RANDOM and GREEDY for generating control part test, are depicted in Table 6, and Table 7. The results are compared with the reference method in [26]. The results in Table 6 correspond to the search space of 1500 random patterns. The not-100% SAF coverage is explained by the not tested faults $f_{i,k} \equiv 0$, and $c_{i,k} \equiv 0$, which were not the target of the control test (Corollary 3), and which will be covered by the data path test.

Table 6. *Comparison of the control test algorithms*

Method	Test length, # instructions	Fault coverage		Time, s
		HL-FF	SAF	
RANDOM	204	100 %	99.34 %	2.00
GREEDY	139	100 %	99.34%	7.85
Method [26]	48	56.0 %	85.8 %	Manually generated
Gate-Level Deterministic	68	57.2%	100%	

Table 7. *Numbers of test operands for testing of the control path*

Op-code	# patterns			Op-code	# patterns		
	RANDOM	GREEDY	PS. D		RANDOM	GREEDY	PS. D
MOV	5	5	2	SHL	11	10	2
ADD	9	7	9	SHR	12	9	2
SUB	10	6	9	ASR	14	9	2
CMP	20	13	2	INC	16	6	3
AND	8	6	4	DEC	15	13	3
OR	20	15	4	RLC	20	13	2
XOR	9	6	4	RRC	20	13	2
NOT	9	4	2	NOP	6	4	2
	Total				204	139	54

The relationships between the fault coverage and test length are illustrated by the curves in Fig.5, and the dependence of test generation time on the size of search space (number of random test candidates) is illustrated in Fig.6.

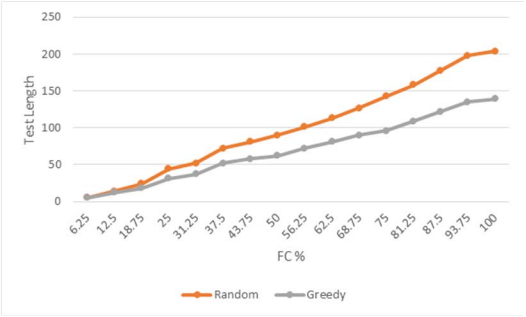

Fig.5. Dependence of the high-level control fault coverage on test length

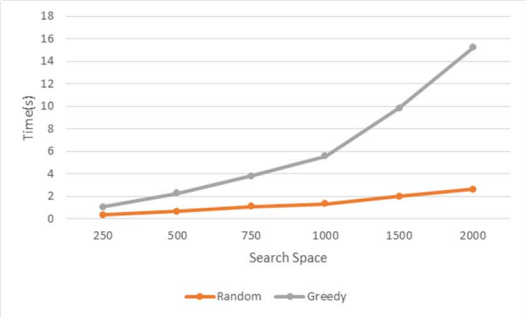

Fig. 6. *Test generation time and the size of search space*

The data path test with pseudo-exhaustive patterns had a length of 95 patterns and produced 99.1% gate-level fault coverage. Both control and data path tests together had a total length of 234 instructions and achieved 100% fault coverage for both high-level and gate-level faults. As a comparison, the referenced method [26] was able to achieve only 56.0 % high-level fault coverage. We also compared with gate-level deterministic approach which gives 100% single SAF coverage. However, because of very low high-level fault coverage this test gives no information about the quality of multiple fault detection.

X. CONCLUSIONS

In this paper, we propose a novel test program generation method which produces high fault coverage. The originality of the proposed method stands in on-line test generation based on modifying on the fly the stored test program template which uses hierarchically organized test data.

The well-structured test program provides high freedom of splitting it into segments, and therefore also high flexibility for delivering test data for cores under test in them system. This is a good basis for optimization of scheduling strategies of organizing test processes at given constraints of currently running applications.

High test quality is achieved by using a new functional model for high-level control faults in processor cores. The method does not need information about implementation details of the cores and uses only the instruction set as input data. We proposed a novel method to detect and prove the redundancy of high-level functional faults, which allows evaluation of the high-level fault coverage more precisely thanks to the possibility of removing redundant faults from the fault list. As the result, it was possible to prove the 100% high-level fault coverage for the control path test.

In [24] and [26], it was shown for the case of testing the ALU instructions that the bigger is the high-level fault coverage of the ALU control test, the bigger will be the confidence of covering multiple low-level faults in the control part of the processor. In accordance with that statement, we can now claim that the 100% high-level control fault coverage, achieved by the proposed test generation method, is equivalent to 100% coverage of low-level multiple faults from the broad class of SAF, conditional SAF, and bridging faults, with avoidance of mutual fault masking in ALU control.

The future work will be in extending the proposed method for the broader instruction sets, and for covering the faults in pipeline stages and addressing logic of processors.

Acknowledgment: The work has been supported by EU FP7 STREP project BASTION, HORIZON 2020 RIA project IMMORTAL, and by European Structural Funds. We thank Mario Schölzel from U Potsdam for providing us with VHDL description of the VLIW processor for carrying out the experiments.

REFERENCES

1. L. Zhang, Y. Han, Q. Xu, X. Li. Defect Tolerance in Homogeneous Manycore Processors Using Core-Level Redundancy with Unified Topology. Design, Automation and Test in Europe, 2008. DATE '08.
2. S. Premkishore, S. W. Keckler, C. R. Moore, D. Burger. Exploiting microarchitectural redundancy for defect tolerance. Proc. ICCD, pp. 481–488, 2003.
3. E. Schuchman, T. N. Vijaykumar. Rescue: a microarchitecture for testability and defect tolerance. Proc. ISCA, pp. 160–171, 2005.
4. A. Kamran, Z. Navabi. Self-Healing Many-Core Architecture: Analysis and Evaluation. VLSI Design Volume 2016, Article ID 9767139, http://dx.doi.org/10.1155/2016/9767139
5. M. A. Skitsas, C. A. Nicopoulos, M. K. Michael. Exploration of System Availability During Software-Based Self-Testing in Many-core Systems under Test Latency Constraints. 2014 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014.
6. J. D. Lee, R. N. Mahapatra, P. S. Bhojwani. A Distributed Concurrent On-Line Test Scheduling Protocol for Many-Core NoC-Based Systems. IEEE Int. Conf. on Computer Design – ICCD, 2009.
7. Y. Li, S. Mitra. VAST: Virtualization-Assisted Concurrent Autonomous Self-Test. Proc. International Test Conference (ITC), 2008, pp. 1-10.
8. S. Gurumurthy, S. Vasudevan, and J. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. ITC '06. Oct 2006, pp. 1–9.
9. D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi. IEEE Trans. on Systematic software-based self-test for pipelined processors. Vol. 16, no. 11, pp. 1441 –1453, Nov. 2008.
10. M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Reorda. Microprocessor software-based self-testing. IEEE Design Test of Computers. Vol. 27, no. 3, pp. 4–19, May 2010.
11. A. Apostolakis, D. Gizopoulos, M. Psarakis, A. Paschalis. Software-based self-testing of symmetric shared-memory multiprocessors. IEEE Trans. on Computers, vol. 58, no. 12, pp. 1682 –1694, Dec. 2009.
12. N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, A. Gonzalez. Self-test optimization in multithreaded multicore architectures. ITC 2010, Nov. 2010, pp. 1 –10.
13. Y. Li, O. Mutlu, S. Mitra. Operating system scheduling for efficient online self-test in robust system. Proc. 2009 Int. Conf. on Computer-Aided Design, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 201–208.
14. M. Agrawal, M. Richter, K. Chakrabarty. Test-Delivery Optimization in Manycore SOCs. IEEE Trans. on computer-aided design of integrated circuits and systems, vol. 33, no. 7, 2014.
15. D.Gizopulos, A.Paschalis, Y.Zorian. Embedded Processor-Based Self-Test. Kluwer Acad. Publishers, 2004, 216 p.
16. L.Chen, et al. A scalable software based self-test methodology for programmable processors," in Proc. of DAC, 2003, pp. 548 - 553.
17. L. Chen and S. Dey. SW-based self-test methodology for processor cores, IEEE Trans. on CAD of IC and systems, vol. 20, no. 3, March 2001, pp. 369 - 380.
18. N.Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software based self-testing of embedded processors," in IEEE Trans. on Comp., vol.54, no.4, 2005.
19. Y.Zhang, H.Li, and X.Li. Automatic test program generation using executing-trace-based constraint extraction for embedded processors," in IEEE Trans. on VLSI Systems, vol.21, no.7, 2013.
20. N. Kranitis, et al "Hybrid-sbst methodology for efficient testing of processor cores," in IEEE Design and Test of Computers, vol. 25, no. 1, 2008, pp. 64-75.
21. C.-H. C. Tai-Hua Lu and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," IEEE Trans. on VLSI Systems, vol. 19, no. 3, March 2011, pp. 516 - 520.
22. C. H.-P. Wen, et al. Simulation-based functional test generation for embedded processors. IEEE Trans. on Comp., Vol.55, No. 11, 2006.
23. A.Jasnetski et al. SW-based Self-Test Generation for Microprocessors with HLDDs. Proc. of the Estonian Academy of Sciences, 2014, 63, 1, 48-61.
24. A.Jasnetski et. al. High-Level Modeling and Testing of Multiple Control Faults in Digital Systems. Proc. of DDECS. Košice, Slovakia, April 20-22, 2016, 6p.
25. R.Ubar, M. Schölzel, S.A. Oyeniran, H.T. Vierhaus. Multiple Fault Testing in Systems-on-Chip with High-Level Decision Diagrams. 10th IEEE International Design & Test Symposium IDT'15, Dead Sea, Jordan,.December 14-16, 2015.
26. A.S.Oyeniran et al. A New Measure for Calculating Multiple Fault Coverage of Microprocessor Self-Test. BEC, Tallinn, Oct 3-5, 2016.
27. S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors, IEEE Trans. On Computers, C-29, No.6, pp.429-441, June 1980.
28. M.L.Bushnell, V.D.Agrawal. Essentials of Electronic testing. Kluwer Acad. Publishers, 2013.
29. M.Schölzel. Self-Testing and Self-Repairing Embedded Processors: Techniques for Statically Scheduled Superscalar Architectures. Habilitation Thesis. Brandenburg University of Technology Cottbus-Seftenberg, 2015.