Maintaining Scalability of Test Generation using Multi-core Shared Memory Systems

Stavros Hadjitheophanous, *Member, IEEE,* and Stelios N. Neophytou, *Senior Member, IEEE,* and Maria K. Michael, *Member, IEEE,*

***Abstract*—Taking advantage of multi-core architectures can provide significant improvement for many design automation problems. However, the parallelization procedure introduces challenges such as workload duplication, limited search space exploration and race contention among different threads. In this work we propose a parallel framework for ATPG using shared memory multi-core systems that supports test generation for both single-detect and multiple-detect fault models. The framework follows a two-epoch approach, each focusing on a different category of faults, during which a test seed generation is followed by compatibility merging. Various optimization techniques are incorporated in each epoch, designed to achieve higher speed-up for the overall test generation procedure without impacting much the test set size. A cluster-based approach is also presented extending the proposed framework to consider multiple-detect fault models without affecting its efficiency. The obtained experimental results demonstrate increased speed-up rates compared to the state-of-the-art multi-core based tools while, at the same time, the test inflation problem is restrained. For the multiple-detect extension, these properties are maintained despite the increased workload and the additional constraint of retaining the number of detections for each fault while merging.**

***Index Terms*—ATPG; Parallel Test Generation; Test Compaction; n-detect; Multi-core systems;**

I. Introduction

TECHNOLOGY shrinking in the integrated circuit manufacturing process allowed the implementation of multiple processing units (cores) on a single chip as well as large amounts of on-chip memory. These developments offer extensive processing power that can be used in various computationally intensive problems including popular electronic design automation processes. However, the distributed fashion of this processing power guides towards the development of parallel methodologies that scale well as the number of cores per chip are expected to increase beyond a few dozens to hundreds.

Automatic Test Pattern Generation (ATPG), a well-known NP-hard problem, becomes more demanding as devices under test are becoming larger and more complicated and as emerging defects require new fault models of higher complexity. While previously proposed procedures are very effective, see [1], [2], among many others, they are inherently non-parallel and thus, cannot rely on automatic parallelization using sophisticated compilers. Proper problem decomposition, workload distribution and final test set re-composition are essential to guarantee the quality of the results while maintaining fault coverage and other test set characteristics such as test size. Since, typically, each computing core does not consider the entire search space, parallel approaches tend to choose local optimal solutions resulting in test set increase [3], known as the test inflation problem.

Parallel ATPG has been studied before the on-chip multi-core era, by either applying bit level parallelism or distributing ATPG components among multiple processing units, not physically on the same chip [3], [4] . These approaches were designed to avoid/minimize communication overhead and were constrained by the machine's word size. In current on-chip multi-core architectures with shared memory, on-chip communication is much faster, significantly reducing the cost of inter-core communication. Furthermore, high level of memory coherency is guaranteed and the number of available cores keeps increasing. These new developments and trends motivate towards the investigation of parallel ATPG approaches capable of achieving speed-up scalability as the number of on-chip cores increases, while overcoming new challenges such as shared memory contention, as well as efficient workload distribution of parallel threads.

ATPG parallelization for on-chip multi-core environments exploit a variety and, often mixture, of parallelism dimensions such as fault parallelism, structural (circuit) parallelism, and algorithmic (including search-space) parallelism. Moreover, the goal of utilizing parallelism often varies. For example, [5] exploits algorithmic parallelism via SAT solver parallelism for maximizing fault coverage with limited speed-up with respect to the corresponding serial process. Similarly, [6] applies bit-level parallelism to generate multiple test patterns concurrently that meet different quality metrics to achieve higher physical-aware coverage. Static fault parallelism is explicitly considered in [7] using a master-slave architecture to reduce inter-process communication which achieves sub-linear speed-up up to 8 cores but suffers from increased test set sizes (test inflation). Furthermore, recent applications of test generation algorithms in security and reliability of integrated circuits employ parallel approaches. For example, the recent work of [8] proposes a side-channel-ware parallel test generation approach which

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 739551 (KIOS CoE) and from of the Republic of Cyprus through the Directorate General for European Programmes, Coordination and Development.

Stavros Hadjitheophanous is with the Electrical and Computer Engineering Dept., KIOS Research and Innovation Center of Excellence, University of Cyprus, Nicosia, Cyprus (e-mail: stavros.hadjitheophanous@ucy.ac.cy).

Stelios N. Neophytou is with the Electrical and Computer Engineering Dept., University of Nicosia, Nicosia Cyprus (e-mail: neophytou.s@unic.ac.cy).

Maria K. Michael is with the Electrical and Computer Engineering Dept., University of Cyprus and KIOS Research and Innovation Center of Excellence, Nicosia, Cyprus (e-mail: mmichael@ucy.ac.cy).

aims to statistically increase hardware Trojan sensitivity.

Parallelization speed-up rates and test set inflation for shared memory architectural models are investigated in [9]–[13]. Shared memory is utilized in [9] as a low latency communication mean with high capacity to leverage synchronization and communication of the process with the goal of minimizing test inflation. Fault coverage is maintained while pattern count is reduced but at the expense of linearly scalable speed-up. The work in [10] proposes a low communication circular pipeline parallel ATPG procedure which emulates the deterministic execution of a serial ATPG in order to be able to reproduce the same test set every time the parallel algorithm is executed. This also leads to limitations in speed-up scalability. The series of works in [11]–[13] target both parallelization speed-up and test inflation minimization strategies, incorporated in state-of-the-art commercial tools. In particular, [11] achieves speed-up by applying dynamic fault partitioning and depth-first-search based compaction. [12] extends [11] for distributed multi-core hybrid architectures, while [13] incorporates a copy-on-write technique for private data protection in order to reduce memory locking when the same part of the memory is used concurrently by more than one cores. Similarly to the above approaches, the work proposed here is targeted towards achieving high degree of speed-up, as the number of available cores increases, and at the same time limiting test set inflation. A more elaborated comparison with the results of the works of [9]–[13] can be found in Subsection V-A.

Parallel approaches have also been proposed targeting Graphic Processing Units (GPUs) based architectures. In contrast to the fault simulation problem where the GPU model can be effective due to its concurrent nature which can directly adopt the single instruction multiple data (SIMD) approach of GPUs [14], [15]. ATPG parallel threads often require inter-thread communication in order to achieve high speed-up rates and avoid test inflation. This communication can be effectively facilitated by shared memory, which is however very limited in GPU-based architectures. Existing approaches for ATPG using GPUs/GPGPUs suffer from limited speed-ups or high test inflation rates [16]–[18]. For example, the recent method in [18] reports overall speed-ups (with respect to a serial ATPG approach) in the order of 0.71x – 40.7x when using a GPU with 2880 processing cores.

In this work we propose a parallel ATPG methodology, for shared memory multi-core systems, geared towards high speed-up and test inflation containment. The methodology takes advantage of fast and low cost shared memory communication, inherent in the underlying architecture, in order to coordinate the main steps of the ATPG to avoid redundant work. The approach dynamically allocates workload, while minimizing memory contention caused by multiple cores (threads) accessing shared data. A test generation flow is proposed in which hard-to-detect faults are targeted first, followed by a parallel fault simulation-based merging process to maximize fault coverage. This process employs a series of newly proposed parallelization heuristics to explicitly avoid simultaneous consideration of the same faults by two or more cores, in order to minimize extra work and thread idle time. Any remaining undetected faults are targeted during a following phase, in a similar manner.

The proposed parallel approach is applicable to fault models of linear size with respect to the circuit size, where the faults can be enumerated (if needed), and is demonstrated in this work using the well known single stuck-at fault model. Furthermore, we extend the approach to n-detect models which require n different tests per fault in order to increase the defect coverage of a test set [19]–[21], at the expense of an increase in the test set size. In this case, the scalability and/or the quality of the proposed partitioning-based parallel approach for single-detect fault models may be impacted as the fault list partitioning does not result in mutually exclusive sub-lists. To address this problem, we extend the proposed parallel ATPG methodology to a clustered-based approach for n-detect test generation. Specifically, the generation of the different detections for the same fault is systematically assigned to different processing cores and test merging for different faults is performed in a restricted manner in order to avoid merging multiple detections for the same fault. Moreover, each generated test for the same fault is optimized to be highly different, as it is well known that this can increase the defect coverage of the overall test set [22]–[24]. To the best of our knowledge, this is the first parallel ATPG approach to explicitly target high quality n-detect test set generation. The obtained experimental results demonstrate the effectiveness of the proposed approach in maintaining scalability of the ATPG process and provide comparisons with relevant recent work.

The rest of the paper is organized as follows. Subsection II presents a high level description of the proposed parallel ATPG while Section III focuses on particular parallel optimizations used to reduce the test inflation problem and favor speed-up. Section IV describes the new challenges for n-detect test sets and proposes a cluster based approach for parallel ATPG. Section V presents and discusses the experimental results and Section VI concludes the paper.

II. Proposed High-Level ATPG Framework

Typically, a parallelization procedure consists of three basic steps: (i) decomposition (domain or functional), (ii) parallel execution, and (iii) final result assembly. Step (ii) can result in a significant compromise of the quality of the obtained results and, at the same time, not offer the expected speed-up. An efficient parallel algorithm should effectively overcome challenges such as memory contention and imbalanced workload distribution. The proposed ATPG method appropriately designs all three steps to ensure that these challenges are treated efficiently. Specifically, two conceptual approaches are adopted: (i) problem partitioning to avoid executing the same work concurrently in different cores and (ii) fine-grained granularity of each step to provide a dynamic distribution of work. This section presents the test generation flow of the proposed methodology which is based on these two concepts; various parallel optimization heuristic based on these concepts are discussed on Section III.

The proposed methodology relies on an initial test-per-fault step, for a limited number of faults, to obtain an initial seed test set over which the algorithm evolves. The many degrees

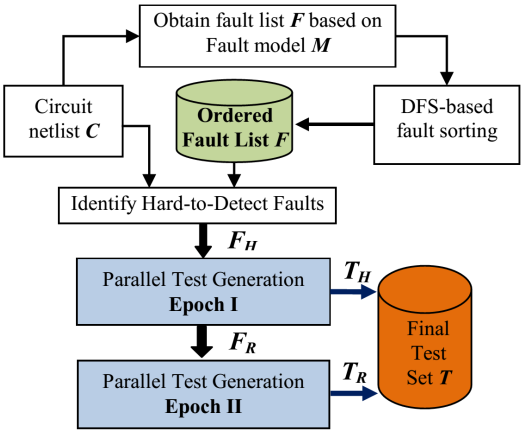


Fig. 1: High level flow of the main Test Generation (TG) processes.

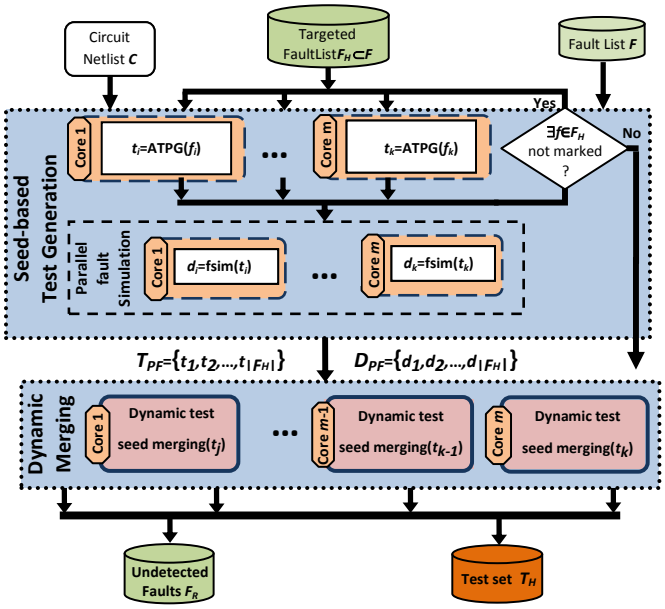


Fig. 2: A test epoch targeting hard-to-detect faults (Epoch I). Same steps are repeated in Epoch II, with input fault list F_R and resulting test set T_R.

of freedom allowed in a test seed by our single fault ATPG process, provides the desired granularity that allows mutually exclusive distribution of work in the different cores. However, this distribution may result in a large amount of unnecessary work when not taking advantage of fault dropping. Fault dropping plays a critical role anyway, as it can significantly affect test set size. In parallel test generation, inefficient dropping of faults can also restrict speed-up, although the main process for identifying faults to be dropped (fault simulation) can be implemented very efficiently in parallel environments [14], [15]. A fair trade-off between high granularity and fault dropping consideration is to develop a methodology based on distinct test epochs, one targeting hard-to-detect faults and a following one targeting the remaining undetected faults.

Fig.1 presents the high level description of the proposed methodology. Firstly, the circuit netlist is analyzed to obtain a collapsed fault list $\boldsymbol{F}$ for the underlying fault model $\boldsymbol{M}$. Consequently, the fault list is sorted in a depth-first-search (DFS) order (based on their location in the netlist) in an attempt to implicitly group faults with structural similarities in $\boldsymbol{F}$. This fault locality property of the input fault list benefits fault dropping after $\boldsymbol{F}$ is partitioned to the available cores. The next step identifies hard-to-detect faults to be targeted by the first test epoch (Epoch I) of the methodology. We use random test pattern generation, which is a simple, quick and acceptable way to classify faults; however, other more sophisticated methods can be incorporated such as [25], [26]. Hard-to-detect faults are identified using a multiple detection approach where 10% (set by experimental exploration) of the faults in $\boldsymbol{F}$ with the fewer detections are considered as hard and used as the input fault list of test Epoch I ($\boldsymbol{F}_H$). Epoch I performs explicit test generation for each fault in $\boldsymbol{F}_H$ also considering faults in $\boldsymbol{F}-\boldsymbol{F}_H$ during fault simulation to identify faults detected coincidentally ($\boldsymbol{F}_C$). A test compaction step follows to produce a set of test patterns (T_H) detecting all faults in $F_H \cup F_C$. A second test epoch (Epoch II), similar to the first one, is invoked to target the remaining faults, i.e. $\boldsymbol{F}_R = \boldsymbol{F} - (\boldsymbol{F}_H \cup \boldsymbol{F}_C)$ producing a set of tests $\boldsymbol{T}_R$ such that $\boldsymbol{T} = \boldsymbol{T}_H \cup \boldsymbol{T}_R$ detects all faults in $\boldsymbol{F}$. Detailed description of the individual steps taken during a test epoch is provided in Subsection III-A.

III. Parallelization Methodology And Optimizations

This section describes in detail how the test generation process is partitioned and discusses the decisions taken to address the main parallelization challenges. Subsection III-A describes the major steps undertaken during a test epoch, discussing dynamic fault partitioning and core synchronization, while Subsection III-B describes a number of optimizations proposed to overcome parallelization issues.

A. Test-Epoch Parallelization

Fig.2 presents a flowchart illustrating the basic steps of the parallel methodology followed during a test epoch, namely *seed-based Test Generation (TG)* and *dynamic test merging*. An epoch explicitly targets only a small subset of the fault list $\boldsymbol{F}$ ($\boldsymbol{F}_H$ for Epoch I and $\boldsymbol{F}_R$ for Epoch II), on a fault-by-fault basis. Note that $\boldsymbol{F}_C = \boldsymbol{F} - (\boldsymbol{F}_H \cup \boldsymbol{F}_R)$ typically constitutes the overwhelming majority of the faults which are easily/randomly detectable.

During the first step (seed-based TG in Fig.2), each available core performs test seed generation (TG with maximal don't care bits) for the next undetected fault $\boldsymbol{f}_i$ in the list using a PODEM-based process optimized to identify tests with a large number of unspecified bits (proven to be beneficial for a plethora of application e.g., test set compaction [24], low power testing [27]). The order of the selection of the next fault(s) is not important here, as the partitioning is designed

to work in an independent manner and produce standalone results. The system shared memory holds the updated fault list (faults not yet targeted) and, therefore, duplication of work is avoided as each core works on a distinct fault. For each test seed $\boldsymbol{t}_i$ generated, parallel fault simulation is performed and all faults detected (including those in $\boldsymbol{F}$ -$\boldsymbol{F}_H$) are stored in a list $\boldsymbol{d}_i$. Faults in $\boldsymbol{d}_i$ are not immediately dropped as this information is used during the following step. This first step terminates when all faults in $\boldsymbol{F}_H$ have been targeted. $\boldsymbol{T}_{PF}$ contains the test seeds and $\boldsymbol{D}_{PF}$ contains the corresponding fault simulation results which are both kept in the shared memory (Fig.2 between the two steps).

The next step is invoked (dynamic test merging in Fig.2) in order to merge compatible test seeds and reduce the size of $\boldsymbol{T}_{PF}$. Each core selects the test seed from $\boldsymbol{T}_{PF}$ with the larger detection list d_i and marks it (core's primary seed) so that other cores cannot select it. A detailed description of this selection is given in Subsection III-B. This merging step is dynamic *due to the efficient communication of the merged tests through the shared memory*. Thus, in each iteration, the number of candidate tests seeds for merging is reduced at a fast rate.

Algorithm 1 outlines the merging process undertaken by each core while the shared memory accommodates information about faults detected and test seeds discarded. The input to this merging process is the test seed generated in the previous step (T_{PF}) and their corresponding faults detected (D_{PF}) as well as the fault list F_H. This process is similar for the two Epochs of the methodology hence, without loss of generality, here we describe the method for Epoch I. Each core considers a distinct subset of the test seed set T_{PF} in order avoid utilizing

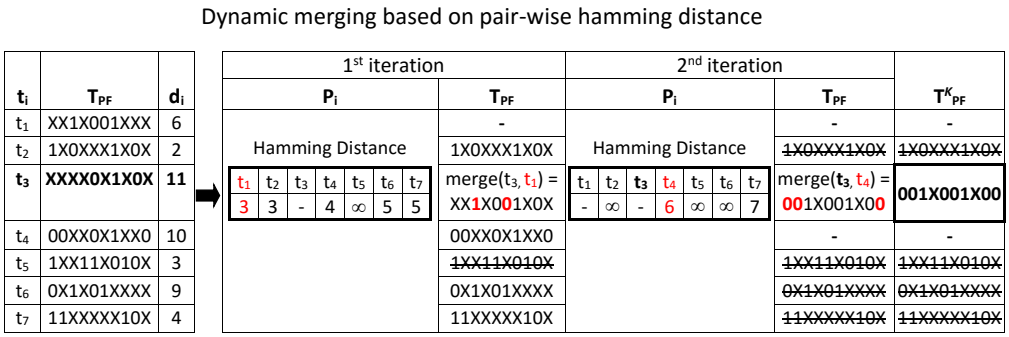
Algorithm 1 *Dynamic Merging for Core k*

Inputs: test seeds T_{PF}, faults detected per seed D_{PF}, shared fault list F, fault sublist F_H
Output: test set $T^k{}_{PF}$

```
01. T^k_PF = ∅
02. For each seed t_i ∈ T_PF(k:k+1) : d_i = max{D_PF(k:k+1)}
03.   F_H = F_H\d_i
04.   T_PF = T_PF - t_i
05.   For h = k,k+1,...,m-1,0,1,...,k-1
06.     For each t_j ≠ t_i ∈ T_PF(h:h+1)
07.       P_i ← hamming(t_i,t_j)
08.     For each t_j ≠ t_i ∈ T_PF(h:h+1) : (t_i,t_j)=min{P_i}
09.       If (min{P_i} = ∞)
10.         break;
11.       t_i ← merge(t_i,t_j)
12.       T_PF = T_PF- t_j
13.       F_H = F_H\d_j
14.   For each unspecified bit b_z of t_i
15.     t_i^0 = t_i : b_z = 0
16.     d_i^0 = fsim(t_i^0,F-F_H)
17.     t_i^1 = t_i : b_z = 1
18.     d_i^1 = fsim(t_i^1,F-F_H)
19.     If (d_i^0≠∅ AND |d_i^0| ≥ |d_i^1|)
20.       t_i = t_i^0
21.       F_C = F_C ∪ d_i^0
22.     If (d_i^1≠∅ AND |d_i^0| < |d_i^1|)
23.       t_i = t_i^1
24.       F_C = F_C ∪ d_i^1
25.   T^k_PF = T^k_PF+t_i
26. F = F\F_C
```

the same seed for merging in more than one cores. Core k (out of the m available) considers only $\frac{|T_{PF}|}{m}$ seeds for primary selection in the range $k \times \frac{|T_{PF}|}{m}, \ldots, (k+1) \times \frac{|T_{PF}|}{m} - 1$, denoted here as $T_{PF}(k : k+1)$. Once a seed t_i in this range is selected (line 02) as a primary seed to be merged, all the detected faults are immediately dropped from the globally maintained fault list F_H (line 03) and the seed is removed from the given seed test T_{PF} (line 04). Then, the hamming distance between each of the remaining seeds in the considered range (i.e., $T_{PF}(k : k+1)$) and t_i is calculated and saved in list P_i (lines 06-07). Observe that, in subsequent iterations, this range changes for the secondary seeds (line 05) so that all seeds in T_{PF} are considered for merging with the primary t_i. Then, the seed with the minimum hamming distance is selected (line 08) and merged (line 11) with t_i. The merged seed is removed from T_{PF} (line 12) and its detected faults are dropped from further consideration (line 13). When no more seeds in the range can be merged with t_i, the algorithm continues to the next range of $\frac{|T_{PF}|}{m}$ seeds (lines 09-10). Lines 14 to 24 are invoked when all the secondary seeds are considered and, hence, no more merging is possible, in order to identify detections of faults not in F_H (i.e., in $F - F_H$). Lines 14 to 24 are skipped in Epoch II since all the remaining faults are placed in the given sublist F_R. First, a bit still with unspecified value in t_i is randomly selected (line 14), fixed to 0 (line 15) and simulated for faults not in F_H (line 16). Then, in lines 17 and 18 bit fixing and fault simulation are repeated for value 1. Based on which bit fixing detects more faults, t_i is updated accordingly (lines 19-20 for 0, lines 22-23 for 1) and a list of faults F_C accumulates all the coincidentally detected faults (line 21 for 0 and 24 for 1). All these faults will be dropped from the global fault list F at the end of this process (line 26). Finally, the obtained test t_i is inserted in a test set for core k (line 25) that contains tests to be placed in the output test set of the Epoch i.e., $T_H = \bigcup_{k=0:m-1} T^k_{PF}$.

Fig. 3 presents the dynamic test merging procedure with an example. First, t_3 is selected since it detects the most faults (11) among the other seeds in T_{PF} (left top table in Fig. 3). Next, the hamming distances between t_3 and all other seeds in T_{PF} are calculated as the sum of the bit-wise distances per bit pair indicated in Column 3 of the bottom tables. For example, the hamming distance between t_3 and t_5 is $1+0+0+1+\infty+0+\infty+1+0+0 = \infty$ indicating that no merging between them is possible. The hamming distances between t_3 and all other seeds in T_{PF} are saved in P_i (shown under P_i column of 1^{st} iteration in top table). These values indicate that the best seed to be merged with t_3 is either t_1 or t_2 with the former selected. Merging of t_3 and t_1 is shown in the T_{PF} column under 1^{st} iteration (changed bits are shown in red) and follows the rules on bottom tables of Fig.3. During the 2^{nd} iteration, the hamming distances are recalculated in P_i. Observe that t_2 and t_6 are now incompatible with t_3 after its merging with t_1. Seed t_4 is selected to be merged with the current seed. The resulting seed has no compatibility with the remaining seeds and hence, no further merging is possible. During Epoch I, bit fixing together with fault simulation follows the process shown here to detect coincidental faults. When all unspecified



t_i	T_{PF}	d_i	1st iteration P_i	1st iteration T_{PF}	2nd iteration P_i	2nd iteration T_{PF}	T^K_{PF}
t_1	XX1X001XXX	6		-		-	-
t_2	1X0XXX1X0X	2		1X0XXX1X0X		~~1X0XXX1X0X~~	~~1X0XXX1X0X~~
t_3	**XXXX0X1X0X**	**11**	t_1 t_2 t_3 t_4 t_5 t_6 t_7 / 3 3 - 4 ∞ 5 5	merge(t_3, t_1) = XX1X001X0X	t_1 t_2 t_3 t_4 t_5 t_6 t_7 / - ∞ - 6 ∞ ∞ 7	merge(t_3, t_4) = 001X001X00	**001X001X00**
t_4	00XX0X1XX0	10		00XX0X1XX0		-	-
t_5	1XX11X010X	3		~~1XX11X010X~~		~~1XX11X010X~~	~~1XX11X010X~~
t_6	0X1X01XXXX	9		0X1X01XXXX		~~0X1X01XXXX~~	~~0X1X01XXXX~~
t_7	11XXXXX10X	4		11XXXXX10X		~~11XXXXX10X~~	~~11XXXXX10X~~

Bit-wise hamming distance calculation and merging rules

i[th] bit of t_i	i[th] bit of t_j	hamming distance	i[th] bit of merge(t_i,t_j)
0	0	+0	0
0	1	+∞	conflict
0	X	+1	0
1	0	+∞	conflict
1	1	+0	1

i[th] bit of t_i	i[th] bit of t_j	hamming distance	i[th] bit of merge(t_i,t_j)
1	X	+1	1
X	0	+1	0
X	1	+1	1
X	X	+0	X

Fig. 3: Dynamic Merging Example.

bits have been fixed and fault simulated, the test is advanced to the output test set T^k_{PF} and all the corresponding faults detected are dropped from the global fault list.

B. Parallelization Optimizations

Detection-Based Primary Test Selection. In the merging step of Algorithm 1, test selection is very important for the efficient evolution of merging since it sets the constraints for the consequent merging iterations, and fault simulation. Practice in ATPG suggests that early fault dropping plays a more important role than having fewer constraints (more unspecified bits) in the test seed. For this reason, the *primary test* $\boldsymbol{t_i}$ during dynamic merging (merging seed) is selected based on its number of detected faults in $\boldsymbol{d_i}$. Recall that the fault simulation process performed at the end of the first step of the test epoch (Fig.2) does not drop faults; instead, it is used as a metric for this selection during the second step. Tests to be merged (with the primary test) are then selected based on their Hamming distance to the primary test. In the (often common) case where more than one tests have the same Hamming distance to the primary test, their fault detection metric (number of coincidentally detected faults) is used to decide which test will be merged. This optimization greatly assists in *dynamic workload balancing* and *minimization of unnecessary work* since early fault dropping reduces the faults for which explicit test generation is needed.

Balanced Workload Distribution. Distribution of workload to the available cores can significantly impact the speed-up of a parallel methodology. Test generation and fault simulation processes have unpredictable execution times due to the nature of the problems and fault dropping. Core idle time is minimized by dynamically selecting: (i) the next fault to be targeted in seed-based test generation in each epoch (Fig.2), (ii) the next test to be used as primary in test merging (Fig.2), and (iii) the tests to be merged with the primary test seed (Algorithm 1). Since the fault list and the test seeds are stored in shared memory, they are easily accessible by all cores, *and can provide a punctual way of determining how the workload will be selected at each step and by each optimization mechanism of the approach.*

Scalable Parallel Fault Simulation. Fault simulation is used in many cases in the proposed methodology and, thus, its performance significantly affects the overall performance. Specifically, fault simulation is used on two separate occasions in each Epoch:

(i) To find the number of detected faults per test seed without fault dropping. This information is given as input to the merging procedure in order to avoid unnecessary simulations after each merging.

(ii) At the end of the merging step in order to detect as many coincidental faults as possible and, hence, minimize the test set size.

In case (i) the fault simulation is performed after test seeds have been generated for all faults. Since generation in the various cores is executed independently, the cores finish this step at different times, resulting in idle cores. These cores can be utilized for fault simulation in a parallel fashion. The challenge here is that the number of idle cores is changing (increasing) as more cores finish and, hence, the simulation should utilize them as well. To take full advantage of the idle cores we have fully incorporated the highly scalable parallel fault simulation of [28]. This fault simulator has been shown to provide linear speed-up as the number of cores increases and can be dynamically adjusted to the number of available cores. In case (ii) the fault simulation should proceed within one core since, after the merging step, the obtained test is simulated to identify co-incidental fault detections (see Fig. 2). Recall, that in this step the test seeds are dynamically acquired by cores

from the shared fault list and merged with other seeds until no more merging is possible. Hence, fault simulation cannot run in parallel to get maximum benefit. Nevertheless, the fault simulations exploit bit-parallel simulation principles presented in [28] where many faults are simulated with a single circuit traversal. This results in a considerable speed-up of this step.

Test Set Private Consideration. The search for the best candidate tests to be merged involves high interaction of each core with the shared memory. Specifically, selecting the primary test, as well as computing the pair-wise compatibilities with the remaining tests in $\boldsymbol{T}_{PF}$, inherently involves memory contention since all cores are searching $\boldsymbol{T}_{PF}$. This issue is addressed by dynamically partitioning $\boldsymbol{T}_{PF}$ in $\boldsymbol{m}$ private subsets ($\boldsymbol{m}$ being the number of available cores), one for each core. Each core can only select tests from its own private subset of $\boldsymbol{T}_{PF}$ (and the corresponding $\boldsymbol{D}_{PF}$) which can be safely moved to its own private cache. This *implicitly minimizes concurrent memory access requests from different cores* that can result in inefficient memory utilization due to memory contention. Moreover, it implicitly *minimizes duplication of work* as each core considers a distinct part in $\boldsymbol{T}_{PF}$. When a core finishes with the merging process within its private part of $\boldsymbol{T}_{PF}$, it is allowed to work on the entire set in order to *ensure workload balancing*. At this point, concurrent memory accesses can occur, however, their impact is minimal as the bulk of the merging process has already occurred during the private consideration, and, hence, the size of $\boldsymbol{T}_{PF}$ is by this point significantly reduced.

Test Provisional Marking. During compatibility merging, the list $\boldsymbol{P}_i$ which holds pair-wise compatibilities between tests, requires updating after each merging. This updating is highly demanding in processing resources as it is of cubic complexity in the worst case. To avoid this issue the proposed methodology calculates and ranks compatibilities only once for each test $\boldsymbol{t}_i$. If a test $\boldsymbol{t}_j$ is selected to be merged with $\boldsymbol{t}_i$, it is provisionally marked in $\boldsymbol{T}_{PF}$ so that it is not merged in another core, *explicitly avoiding imposing unnecessary constraints in another thread that performs merging*. If compatibility between $\boldsymbol{t}_i$ and a test $\boldsymbol{t}_j$ in $\boldsymbol{P}_i$ is invalidated by a previous merging, merging between $\boldsymbol{t}_i$ and $\boldsymbol{t}_j$ is not completed and the provisional marking is cleared. Otherwise, provisional marking indicates permanent discarding of $\boldsymbol{t}_j$ from $\boldsymbol{T}_{PF}$.

Shared Memory Contention Avoidance. Access to the shared memory must be efficient and well-targeted in order to avoid memory contentions. The proposed method accesses the share resources thoughtfully using the following ways: (i) During *Seed-based Test Generation* and *fault simulation* phases shared memory access is avoided since cores work independently on a test seed basis. However, during *Dynamic merging* phase shared memory access can affect the efficiency and the quality of the test generation method. Appropriate bookkeeping with shared fault list F ensures that no test is simulated twice for the same fault and that faults will be dropped immediately after they are detected. (ii) Due to the dynamic nature of the proposed method, the best candidate test seeds would be attractive by many cores. During the merging phase, cores are initially working independently on their own private space for seeds assigned to them ($\boldsymbol{T}_{PF}$) and updating of the shared memory is only done at the end (Test Set Private Consideration). (iii) During the *Dynamic merging* phase, pair-wise compatibilities between tests seeds are calculated once per test seed for $\boldsymbol{T}_{PF}$ (stored in $\boldsymbol{P}_i$). In this case, shared memory access is not necessary since all $\boldsymbol{T}_{PF}$ belong to the private memory space of the cores. Shared memory is only accessed towards the end of processing $\boldsymbol{P}_i$, when very few tests are left to be considered.

IV. n-detect Parallelization Methodology

The proposed methodology can be easily extended to consider any linear fault model. In this section, we extend the parallel framework to generate test sets with multiple detections for each fault, known as n-detect test sets.

The main challenge here is to ensure the n-detect property. The challenge applies both to seed generation and seed merging processes. Test seed generation should guarantee the generation of n different seeds per fault in the given fault list (F_H in Epoch I and F_R in Epoch II). Furthermore, in order to ensure high quality of the resulting test set, test seeds should have significant difference since this was shown to detect more defects [16], [21]–[23]. In addition, the merging process should be constrained in order to guarantee that the seeds generated for the same fault are not merged in the same final test, and, hence, reduce the number of detections for that fault. In Subsection IV-A we propose a test generation technique that satisfies these requirements and can be incorporated in the framework presented in Section II. Subsection IV-B describes a technique for partitioning test seeds that ensures the n-detect property and maintains speed-up increase as the number of cores increases. The steps of the 2-epochs methodology proposed in Sections II and III remain the same.

A. Multiple Test Seed Generation

The PODEM based test generation of Section III-A is extended to produce multiple (n) test seeds per considered fault. The method rolls back in the decision tree of the algorithm altering taken decisions to ensure n incompatible test seeds are generated. Procedure 1 shows the four-step process invoked for each fault.

Hence, following the test generation of the first test seed for a fault (step (i)), the process looks back at the various decisions made during this initial seed generation (step (ii)). A decision here refers to alternative circuit path segments that the algorithm selects in order to generate the seeds. For example, in the circuit of Fig.4 test generation for fault f_x Sa0 is presented. In order to enforce value 1 to f_x (activate fault),

Procedure 1 *Multiple Test Seed Generation*

i. Generate one test seed using a PODEM-based process (as in single detect TG).
ii. Roll back on the decision tree; alter decisions to produce more distinct seeds.
iii. Discard any seed compatible with other seeds.
iv. If necessary fix unspecified bits to obtain n incompatible test seeds.

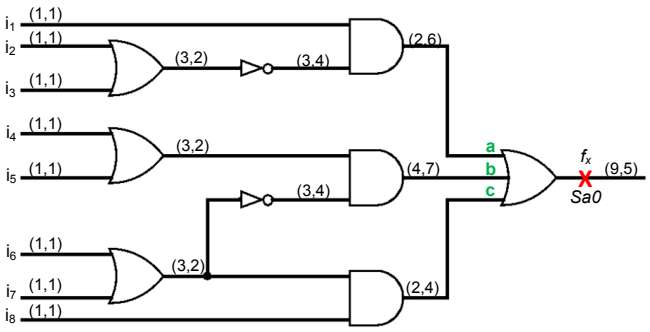


Fig. 4: Decision changes for multiple test seed generation. Numbers inside parentheses denote 0 and 1 controlability values, respectively.

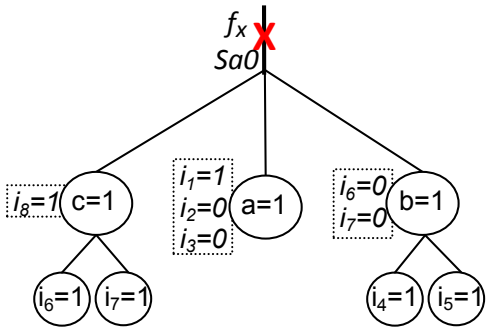


Fig. 5: Fault activation decision tree for f_x Sa0 in Fig.4.

we need at least a 1 at any of the OR gate inputs (i.e, a, b, c). For the first seed, the algorithm assigns value 1 at input c and justifies it with a backward traversal on the circuit. For the second seed, the algorithm takes the alternative decision assigning 1 at line a instead of c. Similar decisions are taken for the propagation of the fault effect to a primary output. The closer the decision to be altered is to the fault site, the higher the difference will be between the seeds. In step (ii) a decision tree is progressively constructed to record all the taken decisions and their alternatives. Fig.5 shows the decision tree for the activation of the Sa0 fault at line f_x. The different paths of the tree indicate the different decision combinations for the primary test seed and for the alternative (n) seeds for the specific fault under investigation.

In step (iii) each generated seed is *checked for compatibility* with previously generated seeds (compatible seeds have no conflicting bits). For each pair of compatible seeds only one is kept, that with the fewer number of specified bits allowing more room for merging. Discarding compatible seeds ensures the n-detect property of the final test set throughout the merging process of the methodology. Specifically, it prevents the generation of the same final test two or more times reducing the number of detections for some faults below n.

Step (iv) of Procedure 1 is invoked only when all the decision combinations for a fault have been exhausted and only when fewer than n seeds have been generated. In this step, the seeds with the fewer specified bits are modified by *specifying bits to conflicting values*, to derive two or more incompatible seeds. The output of this process is n different sets of seeds $T_{PF_1}, T_{PF_2}, ...T_{PF_n}$ each containing one seed generated per fault together with the corresponding number of faults detected by each seed saved in $D_{PF_1}, D_{PF_2}, ...D_{PF_n}$. These sets consist the input to the following part of the methodology.

We explain this modified seed generation with a comprehensive example summarized in Fig. 6. Consider again the circuit of Fig.4 and assume seed generation for fault f_x stuck-at-0. The values inside parenthesis denote the controllability values for 0 and 1, respectively [29]. The generation algorithm selects the line with the smaller controllability metric for logic value 1 i.e., line c to get value 1. In order to justify $c = 1$, the algorithm performs another decision i.e., $i_6 = 1$ while directly implying that $i_8 = 1$. In the decision tree of Fig.5, all direct implications for each decision are shown in a dashed outline next to the decision node. Hence, the seed XXXXX1X1 is generated by taking the leftmost path of the tree in Fig.5 (step (i) of Procedure 1). This step is shown in row 1 of Fig.6. Next, the decision closest to the fault is altered and the input with the next smaller controllability is selected, i.e., $a = 1$. This decision's direct implications ($i_1 = 1, i_2 = i_3 = 0$) generates seed 100XXXXX (step (ii) of Procedure 1) which, however, is discarded since it is compatible with the previous one (step (iii) of Procedure 1) shown in row 2 of Fig.6. In the same way, the next decision is taken and a new seed XXX1X001 (seed #2) is generated which is not discarded as it contains a conflicting bit with seed #1 (row 3 in Fig.6). The process goes on until n different seeds are generated or until all decision combinations have been examined. For the specific example and for $n = 5$, steps (i) - (iii) have produced only 2 different seeds and, hence, step (iv) is necessary. After selecting the seed with the fewer specified bits, seed #1 is replaced by two other seeds one setting its 7^{th} bit (chosen randomly) to value 0 and one setting the same bit to value 1 (indicated with green and red in column 5 in Fig.6). This process is repeated two more times to generate two more tests (columns 7 and 9 in Fig.6) i.e., until 5 different seeds are generated. Step (iv) of Procedure 1 guarantees to produce distinguished seeds, since from previous steps no compatible seeds are allowed to reach step (iv) and the bit fixing process ensures conflicting bits in the obtained seeds. Each of this bit will be placed in a different seed set T_{PF_i}.

The proposed multiple-seed generation process drastically simplifies the merging process, since it needs to consider much fewer constraints among seeds. The procedure is also efficient since the n different seeds are generated in an incremental test generation process which is much faster than n independent seed generations. This is the reason why the multiple seeds generation for the same fault is not chosen to be executed in parallel in the proposed method.

B. Clustered Dynamic Seed Merging

When n seeds are generated per modeled fault, the methodology proceeds to the dynamic merging phase (as in Fig.2). Since the n-detect property is preserved by the seed generation process (at least one conflicting bit between seeds of the

steps (i), (ii) and (iii) of Procedure 1

seed #	decisions taken	implications	t_i
1	c=1, i_6=1	i_8=1	XXXXX1X1
~~2~~	a=1	i_1=1, i_2=0, i_3=0	~~100XXXXX~~
2	b=1, i_4=1	i_6=0, i_7=0	XXX1X00X
~~3~~	c=1, i_7=1	i_8=1	~~XXXXXX11~~
~~3~~	b=1, i_5=1	i_6=0, i_7=0	~~XXXX100X~~
3	none	none	-
4	none	none	-
5	none	none	-

step (iv) of Procedure 1

# Sp bits	t_i'	# Sp bits	t_i'	# Sp bits	t_i'
2	XXXXX101	3	XXX1X101	4	XXX1X101
-	-	-	-	-	
3	XXX1X00X	3	XXX1X00X	3	XXX1X000
-	-	-	-	-	
-	-	-	-	-	
-	XXXXX111	3	XXX1X111	4	XXX1X111
-	-	-	XXX0X101	4	XXX0X101
-	-	-	-	-	XXX1X001

Fig. 6: Example of Procedure 1 n-detect test seed generation. n=5, number of PIs=8

same fault), the main challenge in this step is to ensure that the merging will not affect the speed-up scalability of the methodology. Leaving the merging process identical to the one presented in Algorithm 1, results in a significant reduction in the obtained speed-up by 10%-25%. These observations have been made by employing corresponding experimentation which are not presented here due to space limitations.

To mitigate the speed-up reduction, the parallel framework has been extended to a cluster-based approach where the available cores are partitioned into n clusters. Cluster i explicitly targets dynamic merging for a subset of seeds (i.e., T_{PF_i}) obtained from the seed generation of Subsection IV-A. Fig.7 presents the main components of the cluster-based dynamic merging procedure. Inside each cluster, the dynamic merging procedure is identical to that described with Algorithm 1; yet the number of calculations is significantly smaller than the non-clustered merging as there are fewer seeds to pairwise compare with. While each cluster operates on its own T_{PF_i}, fault dropping is performed via the global fault list located in the shared memory. Hence, any fault detection due to seed merging or identified during the subsequent fault simulation (lines 13, 21 and 24 of Algorithm 1) updates the global fault list, reducing the number of desired detections by 1 per obtained test. When this number becomes 0 the corresponding fault is completely dropped from further consideration. This n-detect aware global fault dropping makes sure that merging will not continue to consider seeds corresponding to dropped faults.

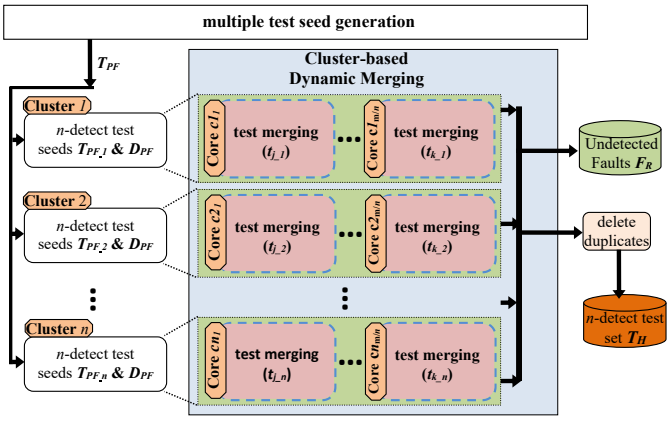


Fig. 7: Cluster-based Dynamic Merging.

The clustered-based dynamic merging produces n independent test sets with complete coverage of the given faults. These n sets are combined into a unified test set by eliminating duplicate tests to provide the final n-detect test set as shown in the right part of Fig.7. This elimination does not affect the n-detect property of the obtained set since, by construction the methodology prevents the generation of the same test two or more times explicitly targeting the same fault. In other words, if two tests are identical (each coming from a separate cluster) they have been generated by merging seeds corresponding to different faults. This occurs since seeds for the same faults are generated to contain at least one conflicting bit.

V. Experimental Results

The proposed method was implemented using C++ language and run on a 20-cores (x2 threads) Intel Xeon CPU E5-2670v2 with 98GBs of RAM, running Linux. OpenMP parallel programming framework was used for parallelization. We present results for the larger full-scan versions of the circuits in the IWLS'05 benchmarks suite. The method can be applied to any linear fault model; here we present results for the stuck-at fault model.

A. Single Detection Parallel Test Generation

Table I presents the obtained speed-up and test set sizes (as increase %) of the proposed parallel ATPG method compared to a serial version of the algorithm. The speed-up measure allows for the evaluation of the scalability of the approach under different execution set-ups, as well as for a fair comparison with other works considering the same architecture but with different characteristics such as CPU clock and total memory. Results from experimental set-ups with 8, 12, 16, 20, and 40 cores are reported. 40 cores are obtained by enabling hyperthreading in the 20 physical core system used. After the circuit name and the number of inputs (Col. 1-2), the size of the circuit and number of faults in the collapsed fault list (Col. 3-4) are presented. Col. 5-6 report the number of aborted faults (indicating the achieved fault coverage) and the test set size obtained by a serial version of the proposed methodology, respectively. The number of aborted faults in the multi-core execution set-ups is always smaller than the one reported in Col. 5 (hence, fault coverage is at least as high) and is not reported here due to space limitations. Col. 7, 9, 11, 13, and 15 list the test set size increase as a percentage of the one

TABLE I: Speed-up and Test set Increase Results for the proposed method using 8, 12, 16, 20 and 40 cores.

				Serial		8 cores		12 cores		16 cores		20 cores		40 cores	
Circuit	**# PIs**	**# Nodes**	**# Faults**	**Aborted**	**|T|**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**
c1355	41	1355	1410	0	84	0	5.61	1.19	7.43	1.19	9.21	1.19	10.38	0.0	20.30
c1908	33	1908	2056	0	111	7.21	4.18	2.70	5.85	6.31	5.02	2.70	7.00	1.80	12.09
c2670	233	2670	2954	0	66	1.52	9.65	-3.03	13.17	-1.52	15.27	1.52	18.40	3.03	28.12
c3540	50	3540	3742	0	103	1.94	8.24	9.71	8.87	2.91	10.56	2.91	11.74	2.91	22.54
c5315	178	5315	6016	0	66	6.06	8.85	9.09	11.35	12.12	14.79	15.15	15.02	21.21	22.54
c6288	32	6288	7744	0	26	7.69	6.58	15.39	7.87	11.54	9.32	11.54	11.57	11.54	24.90
c7552	207	7552	8080	0	101	4.95	6.48	2.97	9.52	5.94	12.25	5.94	15.14	9.90	28.64
s9234.1	247	9234	6781	0	141	7.09	4.91	14.18	7.48	12.06	9.22	15.60	11.67	18.44	18.56
s13207	700	13207	10456	0	255	4.31	5.92	6.67	9.01	7.06	10.72	7.45	13.54	9.02	21.09
s15850.1	611	15850	12150	0	125	8.00	4.97	10.40	8.29	12.8	9.77	14.40	11.44	18.40	23.70
s38417	1664	38417	32320	0	117	1.71	7.61	4.27	10.38	7.69	14.09	9.40	16.50	15.38	24.06
s38584.1	1464	38584	38358	0	131	3.05	6.93	6.11	10.07	7.63	13.99	9.16	16.97	13.74	26.32
s35932	1763	35932	39094	0	20	5.00	6.22	15.00	9.37	20.00	11.00	25.00	11.69	25.00	14.42
b14	277	21680	23716	0	751	1.33	7.34	1.20	11.20	0.80	14.71	1.07	17.50	0.67	30.71
b15	485	20186	23498	192	461	0	7.86	0.87	11.40	0.87	14.93	1.30	19.20	2.17	28.64
b17	1449	61044	75498	0	826	0.36	7.32	0.85	11.33	1.57	15.04	2.78	18.54	4.96	30.63
b18	3307	179967	223352	8	1030	0.78	7.52	1.46	10.57	2.04	13.92	2.33	15.34	4.17	29.22
b19	6666	479800	534144	991	3245	0.80	7.16	6.01	10.05	5.76	12.18	6.29	14.03	6.72	24.69
b20	522	31258	34528	0	519	2.70	7.78	3.28	11.75	2.31	15.04	5.97	17.91	8.67	31.99
b21	522	31157	34331	0	561	2.67	7.66	1.78	11.45	3.03	14.76	1.43	19.48	1.78	31.23
b22	735	39385	48812	0	561	-1.07	7.58	1.426	11.59	0.36	14.34	3.21	17.99	4.99	34.56
ac97_ctrl	2283	39485	39226	0	62	3.23	5.98	6.45	7.67	11.29	10.26	12.90	11.98	19.35	21.03
ucb_funct	1874	40479	42214	0	113	7.96	6.18	13.27	9.18	14.16	11.12	15.04	14.46	18.58	22.98
tv80	373	24357	24810	0	554	0.61	8.08	0.61	11.71	0.20	15.19	1.01	17.06	1.44	26.77
systemcaes	930	30015	29256	0	143	0.70	6.77	6.29	9.06	5.59	12.43	8.39	13.99	10.49	22.73
mem_ctrl	1198	37904	39882	0	485	1.86	6.91	3.09	10.52	0.82	13.91	2.89	17.25	6.80	32.40
ethernet	10640	223959	221628	5	1421	1.27	7.05	1.97	9.70	2.18	12.50	3.03	14.33	4.64	28.91
					Average	**3.03**	**6.94**	**5.30**	**9.85**	**5.80**	**12.43**	**7.02**	**14.82**	**9.10**	**25.16**
			Average Memory Increase (x)			**2.11**		**2.59**		**3.02**		**3.27**		**3.77**	

TABLE II: Speed-up, Test Set Size and Memory Increase Comparison with the works in [10], [11] and [13].

	Comparison with [10] - 12 cores						**Comparison with [11] and [13] - 16 cores**								
	Speed-up (x)		**Test set increase (%)**		**MemoryIncrease (x)**		**Speed-up (x)**			**Test set increase (%)**			**MemoryIncrease (x)**		
Circuit	**[10]**	**prop.**	**[10]**	**prop.**	**[10]**	**prop.**	**[11]**	**[13]**	**prop.**	**[11]**	**[13]**	**prop.**	**[11]**	**[13]**	**prop.**
D1	8.20	-	-1.50	-	-	-	-	-	-	-	-	-	-	-	-
D2	7.70	-	19.00	-	-	-	-	-	-	-	-	-	-	-	-
D3	7.50	-	16.00	-	-	-	-	-	-	-	-	-	-	-	-
D4	7.30	-	1.40	-	-	-	-	-	-	-	-	-	-	-	-
D5	-	-	-	-	-	-	7.38	9.99	-	10.64	0.41	-	4.33	3.02	-
D6	-	-	-	-	-	-	9.37	10.00	-	2.54	12.17	-	4.89	3.22	-
D7	-	-	-	-	-	-	8.88	9.28	-	1.37	2.14	-	3.94	2.78	-
s38417	7.50	9.99	40.00	4.27	-	2.18	-	-	14.09	-	-	7.69	-	-	2.98
s38584.1	7.30	9.89	12.00	6.11	-	2.03	-	-	13.99	-	-	7.63	-	-	2.35
s35932	7.40	8.64	10.00	15.00	-	2.75	-	-	9.34	-	-	20.00	-	-	3.15
b15	7.80	11.40	18.00	0.87	-	2.92	-	-	14.93	-	-	0.87	-	-	3.01
b17	8.30	11.33	5.00	0.85	-	3.09	-	-	15.04	-	-	1.57	-	-	3.27
b18	-	10.57	-	1.46	-	3.80	-	-	13.92	-	-	2.04	-	-	4.09
b19	-	10.05	-	6.01	-	3.11	-	-	12.18	-	-	5.76	-	-	3.78
ethernet	-	9.70	-	1.97	-	3.59	-	-	12.50	-	-	2.18	-	-	4.19
Average	**7.67**	**10.20**	**13.32**	**4.58**	**-**	**2.93**	**8.54**	**9.76**	**13.25**	**4.85**	**4.91**	**5.97**	**4.39**	**3.01**	**3.35**

TABLE III: Test set size, test set increase % and speed-up for *5*-detect parallel ATPG method using 1, 20, 30, 40 cores

	5-detect $\|T\|$	**$\|T\|$ increase (%)**			**Speed-up (x)**		
Circuit	**serial**	**20 cores**	**30 cores**	**40 cores**	**20 cores**	**30 cores**	**40 cores**
s9234	696	6.75	9.34	12.07	12.96	20.46	28.21
c6288	133	7.52	9.02	11.28	16.72	24.31	36.44
c7552	500	3.60	6.20	9.40	17.27	25.82	36.56
s13207	1279	3.67	6.49	8.99	14.96	22.32	31.72
s15851	624	2.88	6.25	10.74	17.44	23.91	31.54
tv_80	2782	4.06	6.00	8.38	18.69	28.36	37.94
b15	2316	5.35	6.13	8.51	19.00	28.23	38.72
b14	3734	1.47	3.91	5.17	19.57	28.55	37.58
systemcaes	691	1.45	3.91	4.63	18.45	28.77	36.52
s38417	580	8.28	9.83	10.69	18.19	25.92	32.26
b21	2775	5.12	7.64	8.58	17.86	23.93	28.90
b20	2616	3.71	5.73	7.38	17.81	24.33	30.17
s35932	98	4.08	7.14	13.27	17.38	24.95	37.91
s38584	654	11.62	11.93	13.91	16.60	19.87	33.83
ac97_controler	319	2.51	3.76	4.84	19.18	29.15	36.99
ucb_funct	544	7.90	11.95	14.34	17.99	28.81	39.61
mem_ctrl	2387	2.09	8.38	10.64	18.58	29.45	36.45
b22	2833	4.24	6.57	8.72	17.53	26.17	33.54
b17	4098	7.49	9.86	11.35	19.45	28.60	34.76
ethernet	7052	6.83	8.61	9.25	18.74	29.30	38.76
b18	5130	1.34	2.71	3.33	18.69	27.84	38.75
b19	16051	1.30	2.98	3.50	19.03	28.95	36.87
	Average	**5.03**	**7.43**	**9.77**	**17.72**	**26.06**	**34.92**

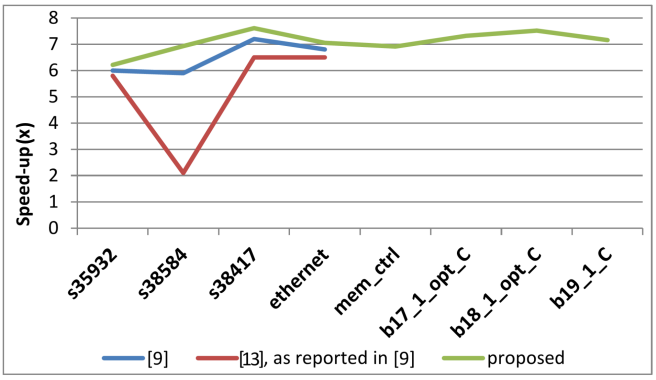


Fig. 8: Speed-up comparison with [9] and [13] for an 8-core set-up.

obtained by the serial execution and Col. 8, 10, 12, 14, and 16 report the speed-up achieved, when 8, 12, 16, 20, and 40 cores are employed.

The obtained results demonstrate almost linear speed-up increase while at the same time the test set size increase is very limited for most of the circuits. In the worst-case test set increase is no more than ~20% whereas in the average case it is only 9.10%, for 40 cores. Circuit s35932 is an exception (with 25% increase) due to the very small test set size of the serial version with 20 tests, which becomes 25 tests for 40 cores. The proposed method also exhibits small memory increase, an objective often targeted by parallel methods. The last row of Table I summarizes the average memory increase among all circuits. For the 20-cores run, the required memory is only 3.27x more than the serial version, while the 40 cores runs increase the memory by 3.77x on the average. These numbers indicate that the memory increase does not grow proportionally with the number of cores; rather, it grows with a decreasing rate as the number of cores increase. This is attributed to the dynamic manner in which the proposed algorithm performs fault dropping, alleviating the following steps from unnecessary calculations.

The rest of this section compares the proposed methodology with the most relevant and recent parallel approaches considering the shared memory multi-core architecture model, such as [9]–[11], [13]. Where available, results are compared directly for the common benchmarks. Moreover, results on additional benchmarks for each technique are listed and average trends for each methodology are analyzed. For the proposed methodology, the larger (in terms of # nodes) circuits (b18, b19 and ethernet) are listed, on top of the common ones. The number of execution cores in each case was determined by the results reported in these works. Col. 2-7 of Table II provides a comparison with [10] for the set-up of 12 cores. Similarly, Col. 8-16 of Table II compare the proposed methodology with the approaches of [11] and [13], for the 16-cores setup reported in these works. A "–" indicates no available results for the corresponding work. Average trends are reported in the last row of Table II.

The proposed methodology reports higher speed-up on all circuits with an average 10.2x compared to the average speed-up of 7.67x reported in [10]. At the same time the proposed methodology reports lower test set increase than [10] in all circuits except s35932. For the largest common circuits reported (b15 and b17) the proposed methodology reports a negligible test set increase (less than 1%) and considerable speed-up (>11x) while [10] reports speed-up around 8x and much higher test set increase (18% and 15%, respectively). Col. 7 reports the memory increase factor for the proposed method, although this is not reported in [10], for completeness. For the case of the works in [11] and [13], the comparison is performed solely on average trends, as none of the industrial circuits D1-D7 are available to us. Clearly, the proposed methodology exhibits higher average speed-up than both approaches (13.25x vs 8.54x and 9.76x) with a small additional overhead on average test set size increase in the order of ~1% (5.97% vs 4.85% and 4.91%). The memory increase factor of the proposed method is on the average smaller than that of [11] and slightly higher than the one of [13] which explicitly targets memory utilization.

An 8-core system is considered in [9] and compared with an implementation of [13]. Fig.8 compares the speed-up of the proposed methodology with that of [9] and [13], as reported in [9]. The proposed methodology achieves on average ~7x speed-up, outperforming both existing methods for the 4 common circuits. This is achieved mainly due to the optimizations (discussed in Section III), which minimize redundant work by immediate updating of the fault status. Comparison regarding the test set size reduction is not possible with [9] as absolute values for test set sizes as well as corresponding fault coverage are not provided.

B. Parallel n-detect Test Generation

A comprehensive picture of the linear performance of the proposed n-detect test generation extension for n=5 is presented in Fig. 9 and Fig. 10. The scalability of the technique is illustrated in Fig. 9 compared to a serial execution of the algorithm. The achieved speed-up is reported in y-axis for different number of cores utilized (shown in x-axis). For all circuits examined, the proposed methodology scales linearly as the number of cores used for its computation increases. Moreover, observe that for larger circuits the obtained speed-up is higher due to the increased workload that allows smaller percentage of core idle times. As with the single-detect case the results imply that the scalability of the proposed method will continue to scale well as the number of cores is increased. In fact, while the speed-up trends are similar to that of the single-detect in absolute values, the speed-up is higher in n-detect and in some cases close to the theoretical maximum (number of utilized cores). For example, two of the largest circuits considered b18 and ethernet achieve ~19x speed-up for 20 cores, ~28x speed-up for 30 cores and ~39x speed-up when 40 cores are utilized. The corresponding numbers for single-detect are ~15x, ~22x and ~29x. This is mainly attributed to the cluster-based approach that dramatically reduces the number of pairwise condition checks during merging. The checks are restricted only inside the cluster where the seed to be merged was assigned. This possibility cannot be exploited

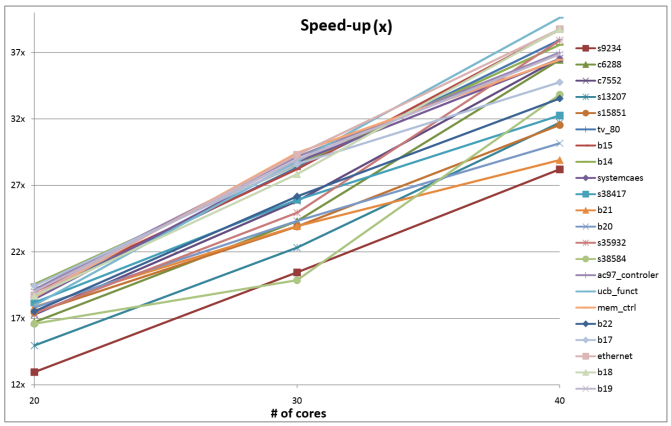


Fig. 9: Scalability of the proposed parallel n-detect test generation with respect to a serial execution, for n=5.

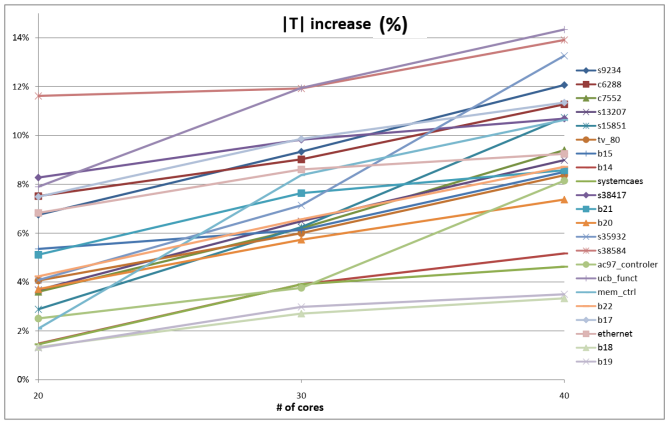


Fig. 10: 5-detect test set size ($|T|$) augmentation % compared to serial execution of the proposed method.

in a serial approach in a straightforward manner, without significant effect either at the performance or the final test set size.

Similarly to the results presented in Subsection V-A, the parallelization procedure affects the final test set size. Moreover, for the n-detect approach, due to the clustered-based merging, the solution is obtained from a reduced search space. Hence, it is expected, in many cases, to provide a sub-optimal solution that returns test detecting fewer faults. When fewer faults are detected per test, it is inevitable that the final test set will be larger (test set inflation). Fig. 10 shows the test set size increase as the number of cores used is increased for the examined benchmark circuits. The increase is reported as a percentage of the test set size obtained by a serial execution of the same algorithm. Unexpectedly, the test inflation for the n-detect method is not significant. In particular, for the 5-detect test generation the test inflation is on average 5% for 20 cores, 7.5% for 30 cores, and 9.5% for 40 cores whereas the corresponding numbers for the single-detect case are 7%, 8.5% and 9%.

Table III provides the exact data for the test set size and speed-up achieved by the proposed parallel n-detect test generation method for n=5 when 20, 30, 40 cores are employed. All test sets provide 100% 5-detect fault detection efficiency, i.e., all modeled faults are detected at least 5 times. The test set sizes are presented as % increase over the serial version of the same algorithm (shown in column 2). 5-detect fault coverage is achieved due to the *multiple test seed generation* procedure which enforces the generation of 5 different tests seeds. Columns 3, 4 and 5 show the % test set size increase for 20, 30 and 40 cores, respectively. Columns 6, 7 and 8 report the achieved speed-up when 20, 30 and 40 cores are utilized. The obtained results demonstrate that the linear speed-up increase as well as the small test set size increase achieved in the single-detect method are maintained. The achieved speed-up when 20, 30 and 40 cores are employed is on the average higher in the n-detect method. Specifically, it is 17.72x, 26.06x, and 34.92x for 20, 30 and 40 cores compared to 14.82x, 19.96x and 25.16x, respectively, for the single-detect method.

VI. Conclusions

We propose a parallel test pattern methodology for shared memory multi-core environments. A number of newly proposed heuristics attempt to avoid assigning the same workload to multiple cores, while the distribution of work in the available resources aims to minimize the core idle time. The methodology is also extended to generate multiple-detect (n-detect) test sets, previously proven to provide higher defect coverage. A new technique for the efficient generation of test seeds followed by a clustered-based dynamic merging procedure have been presented. Experimental results demonstrate high speed-up rates that keep increasing as the number of the available cores increases. Test set size increase is limited and comparable to other state-of-the-art parallel approaches. For the n-detect method, the results retain all the good properties of the basic methodology in a more beneficial extent.

References

[1] K. Scheibler, D. Erb, and B. Becker, "Improving test pattern generation in presence of unknown values beyond restricted symbolic logic," in *Test Symposium (ETS), 2015 20th IEEE European*. IEEE, 2015, pp. 1–6.

[2] S. Eggersglub, K. Schmitz, R. Krenz-Baath, and R. Drechsler, "Optimization-based multiple target test generation for highly compacted test sets," in *Test Symposium (ETS), 2014 19th IEEE European*. IEEE, 2014, pp. 1–6.

[3] S. Patil and P. Banerjee, "Fault partitioning issues in an integrated parallel test generation/fault simulation environment," in *Test Conference, 1989. Proceedings. Meeting the Tests of Time., International*. IEEE, 1989, pp. 718–726.

[4] J. M. Wolf, L. M. Kaufman, R. H. Klenke, J. H. Aylor, and R. Waxman, "An analysis of fault partitioned parallel test generation," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 15, no. 5, pp. 517–534, 1996.

[5] A. Czutro, I. Polian, M. Lewis, P. Engelke, S. M. Reddy, and B. Becker, "Tiguan: Thread-parallel integrated test pattern generator utilizing satisfiability analysis," in *VLSI Design, 2009 22nd International Conference on*. IEEE, 2009, pp. 227–232.

[6] K.-Y. Liao, C.-Y. Chang, and J. C.-M. Li, "A parallel test pattern generation algorithm to meet multiple quality objectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 11, pp. 1767–1772, 2011.

[7] K.-W. Yeh, M.-F. Wu, and J.-L. Huang, "A low communication overhead and load balanced parallel ATPG with improved static fault partition method," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2009, pp. 362–371.

[8] Y. Huang, S. Bhunia, and P. Mishra, "Scalable test generation for trojan detection using side channel analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 11, pp. 2746–2760, 2018.
[9] J. C. Ku, R. H. Huang, L. Y. Lin, and C. H. Wen, "Suppressing test inflation in shared-memory parallel automatic test pattern generation," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 664–669.
[10] K.-W. Yeh, J.-L. Huang, H.-J. Chao, and L.-T. Wang, "A circular pipeline processing based deterministic parallel test pattern generator," in *Test Conference (ITC), 2013 IEEE International*. IEEE, 2013, pp. 1–8.
[11] X. Cai, P. Wohl, J. A. Waicukauski, and P. Notiyath, "Highly efficient parallel ATPG based on shared memory," in *Test Conference (ITC), 2010 IEEE International*. IEEE, 2010, pp. 1–7.
[12] X. Cai and P. Wohl, "A distributed-multicore hybrid ATPG system," in *Test Conference (ITC), 2013 IEEE International*. IEEE, 2013, pp. 1–7.
[13] X. Cai, P. Wohl, and D. Martin, "Fault sharing in a copy-on-write based ATPG system," in *Test Conference (ITC), 2014 IEEE International*. IEEE, 2014, pp. 1–8.
[14] E. Schneider, S. Holst, M. A. Kochte, X. Wen, and H.-J. Wunderlich, "Gpu-accelerated small delay fault simulation," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1174–1179.
[15] M. Li and M. S. Hsiao, "3-d parallel fault simulation with gpgpu," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 10, pp. 1545–1555, 2011.
[16] K.-Y. Liao, S.-C. Hsu, and J. C.-M. Li, "Gpu-based n-detect transition fault ATPG," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 28.
[17] K.-Y. Liao, P.-J. Chen, A.-F. Lin, J. C.-M. Li, M. S. Hsiao, and L.-T. Wang, "Gpu-based timing-aware test generation for small delay defects," in *Test Symposium (ETS), 2014 19th IEEE European*. IEEE, 2014, pp. 1–2.
[18] M. Osama, L. Gaber, A. I. Hussein, and H. Mahmoud, "An efficient sat-based test generation algorithm with gpu accelerator," *Journal of Electronic Testing*, vol. 34, no. 5, pp. 511–527, 2018.
[19] E. J. McCluskey and C.-W. Tseng, "Stuck-fault tests vs. actual defects," in *Test Conference, 2000. Proceedings. International*. IEEE, 2000, pp. 336–342.
[20] C.-W. Tseng and E. J. McCluskey, "Multiple-output propagation transition fault test," in *Test Conference, 2001. Proceedings. International*. IEEE, 2001, pp. 358–366.
[21] J. Geuzebroek, E. J. Marinissen, A. Majhi, A. Glowatz, and F. Hapke, "Embedded multi-detect ATPG and its effect on the detection of unmodeled defects," in *Test Conference, 2007. ITC 2007. IEEE International*. IEEE, 2007, pp. 1–10.
[22] S. N. Neophytou and M. K. Michael, "Test pattern generation of relaxed *n*-detect test sets," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 410–423, 2012.
[23] Y.-T. Lin, O. Poku, N. K. Bhatti, R. S. Blanton, P. Nigh, P. Lloyd, and V. Iyengar, "Physically-aware *n*-detect test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 2, pp. 308–321, 2012.
[24] X. Kavousianos and K. Chakrabarty, "Generation of compact test sets with high defect coverage," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1130–1135.
[25] M. Venkatasubramanian, "Failure evasion: Statistically solving the np complete problem of testing difficult-to-detect faults," Ph.D. thesis, Auburn University, 2016.
[26] O. Golubeva, "Detection of hard-to-detect stuck-at faults and generation of their tests based on testability functions," in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. IEEE, 2018, pp. 1–5.
[27] P. Girard, N. Nicolici, and X. Wen, *Power-aware testing and test strategies for low power devices*. Springer Science & Business Media, 2010.
[28] S. Hadjitheophanous, S. N. Neophytou, and M. K. Michael, "Exploiting shared-memory to steer scalability of fault simulation using multicore systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[29] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.

Stavros Hadjitheophanous received his B.Sc. and Ph.D. degrees in Computer Engineering from the Electrical and Computer Engineering Department, University of Cyprus in 2009 and 2018 respectively. In 2010, he obtained the M.Sc. degree in Advance Computing Internet Technologies with Security from the University of Bristol, United Kingdom. Since 2010 Stavros is a researcher at KIOS Research and Innovation Center of Excellence. His main research interest include CAD algorithms for VLSI, multi-core algorithms for high quality testing, reliability, and embedded applications.

Stelios N. Neophytou is an Associate Professor at the Department of Engineering, University of Nicosia, Cyprus. He holds an Engineering Diploma from the Computer Engineering and Informatics Department of University of Patras, Greece (2003), and a PhD degree from the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus (2009). His research interests fall under Electronic Design Automation focusing on algorithm development for integrated circuits' design verification and post-manufacturing testing. Specifically, his work involves methodologies for quality enhancement of the Automatic Test Pattern Generation (ATPG) process related to detection of non-modeled defects, test size compaction and scaling in multiprocessing environments. He has also work related to test generation techniques for Built-In Self-Test architectures as well as graph-theoretic problems considering Binary Decision Diagrams. He is a senior member of IEEE.

Maria K. Michael is an Associate Professor at the Department of Electrical and Computer Engineering at the University of Cyprus. She is also a founding member and the Director of Education and Training at the KIOS Research and Innovation Center of Excellence, also at the University of Cyprus. Maria has a Ph.D. degree from the ECE Dept. of Southern Illinois University, Carbondale-USA. Her research expertise falls in the areas of test and reliability of digital circuits and chip-level architectures, with emphasis on embedded and general-purpose multi-core systems reliability and on-line testing, dynamic/intelligent parallel CAD algorithms for automatic testing and fault simulation, intelligent methods for design, test and fault tolerance, delay test and emerging fault models. Recent research interests expand to design and optimization of embedded systems and other chip-level architectures, dynamic self-detecting and self-healing architectures, and dependability and security in the hardware backbone of cyber-physical systems. She has published numerous papers in high-caliber refereed journals and international conferences and she serves on steering, organizing and program committees of several IEEE and ACM conferences in the areas of test and reliability. She is a co-recipient of a Best Paper Award of MSE'2009. She is a member of the IEEE and the ACM.