Integrated LFSR Reseeding, Test-Access Optimization, and Test Scheduling for Core-Based System-on-Chip

Zhanglei Wang, Krishnendu Chakrabarty, *Fellow, IEEE*, and Seongmoon Wang

***Abstract*—We present a system-on-chip (SOC) testing approach that integrates test data compression, test-access mechanism/test wrapper design, and test scheduling. An efficient linear feedback shift register (LFSR) reseeding technique is used as the compression engine. All cores on the SOC share a single on-chip LFSR. At any clock cycle, one or more cores can simultaneously receive data from the LFSR. Seeds for the LFSR are computed from the care bits for the test cubes for multiple cores. We also propose a scan-slice-based scheduling algorithm that attempts to maximize the number of care bits the LFSR can produce at each clock cycle, such that the overall test application time (TAT) is minimized. This scheduling method is static in nature because it requires predetermined test cubes. We also present a dynamic scheduling method that performs test compression during test generation. Experimental results for International Symposium on Circuits and Systems and International Workshop on Logic and Synthesis benchmark circuits, as well as industrial circuits, show that optimum TAT, which is determined by the largest core, can often be achieved by the static method. If structural information is available for the cores, the dynamic method is more flexible, particularly since the performance of the static compression method depends on the nature of the predetermined test cubes.**

***Index Terms*—ATPG, system-on-chip test, test compression, test scheduling.**

I. Introduction

RECENT growth in design complexity and the integration of embedded cores in system-on-chip (SOC) ICs have led to a significant increase in test data volume, test application time (TAT), and manufacturing test cost. Test data compression provides a promising solution to these problems [1]–[4]. Some state-of-the-art compression methods such as [4] use test generation techniques to generate patterns that are more suitable for compression. The performance of most compression techniques also depends on the number and lengths of scan chains. However, some SOC chips contain IP cores or black box cores that are not provided to the system integrator with detailed structural information [5]. Many SOCs also include hard cores that are delivered in the form of layouts such that the configurations of scan chains cannot be modified. Existing compression techniques for stand-alone ICs are, therefore, less efficient for such SOCs.

In addition to the problem of limited applicability of existing test compression techniques, restricted access to internal cores is another challenge in SOC testing [6]. To tackle this problem, test-access mechanism (TAM) and test wrappers have been proposed as key components of an SOC test architecture [7], as shown in Fig. 1. TAMs deliver precomputed test sequences to cores on the SOC, while test wrappers translate these test sequences into patterns that can be applied directly to the cores. The test wrapper and the TAM design directly impact the vector memory depth required on the automatic test equipment (ATE), testing time, and thereby affect test cost. Many techniques have been proposed for TAM/wrapper design under different constraints (e.g., testing time, test bus width, power dissipation, control overhead, routing, and layout) [8]–[16]. However, these techniques either do not consider test data compression, or they utilize relatively inefficient compression techniques [17].

In [18], test patterns for each core in an SOC are compressed separately using linear feedback shift register (LFSR) reseeding. Tester channels are time-multiplexed to transfer seed data to the LFSRs of each core. Patterns of each core are first split into blocks of fixed length. A seed is obtained by satisfying care bits from a variable number of blocks. When an LFSR is expanding a seed to a series of blocks, it need not receive data until all blocks encoded by this seed have been generated. Hence, seed streams for different cores can be time-multiplexed into one stream. The overall TAT is therefore reduced by testing cores simultaneously. The major drawback of [18] is that extra data and hardware are needed to enable the time-multiplexing mechanism. The use of fixed length blocks adversely affects the encoding efficiency. An optimum block length for one core is not necessarily optimum for other cores.

In [19], an XOR-network approach is used for test compression, and a compression driven TAM design heuristic is proposed. This heuristic is guided by a test time estimation function, which is obtained using curve fitting. It is not clearly reported in [19] how the estimation function can be derived,

Manuscript received August 2, 2008; revised January 6, 2009. Current version published July 17, 2009. The work of Z. Wang and K. Chakrabarty was supported in part by the National Science Foundation under Grant CCR-0204077. An earlier version of this paper appeared in Proc. IEEE/ACM Design, Automation and Test in Europe (DATE) Conference, pp. 201–206, 2007. This paper was recommended by Associate Editor A. Ivanov.

Z. Wang is with the Cisco Systems, Inc., San Jose, CA 95134 USA (e-mail: zhawang@cisco.com).

K. Chakrabarty is with the Electrical and Computer Engineering Department, Duke University, Durham, NC 27708-0291 USA (e-mail: krish@ee.duke.edu).

S. Wang is with the NEC Laboratories America, Inc., Princeton, NJ 08540 USA (e-mail: swang@nec-labs.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2021731

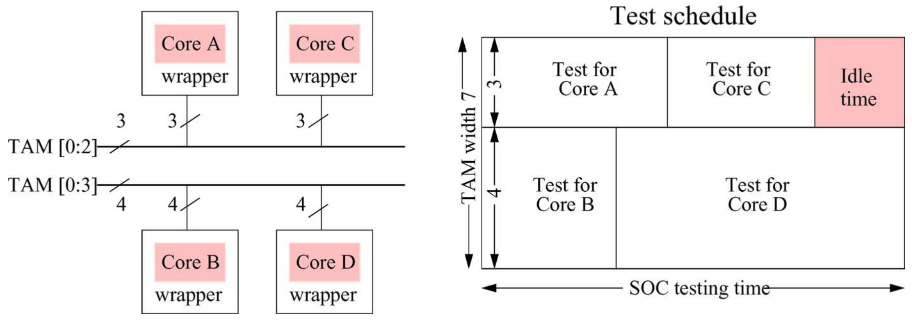


Fig. 1. Illustration of test wrapper, TAM, and test schedule [21].

and what impact this function has on the efficiency of the TAM design heuristic. Test scheduling is also not considered.

In this paper, we propose an SOC testing approach that integrates test data compression, TAM/test wrapper design, and test scheduling. We choose the LFSR reseeding technique proposed in [20] as the compression engine because of its high encoding efficiency. A single on-chip LFSR-based decompressor is used to feed all cores on the SOC. At a given clock cycle, each core is in one of the following modes: 1) Shift mode—data are shifted in from the LFSR, and output responses are shifted out; 2) Capture mode—output responses are captured into the scan cells; and 3) Inactive mode—the core is not scheduled for test at this clock cycle. Therefore, the LFSR is shared among the cores that are in the shift mode; other cores do not receive data from the LFSR. With appropriate TAM design and test scheduling, more cores can be tested in parallel, and the TAT for the entire SOC can be significantly reduced. Our experimental results show that in most cases, we can achieve a minimum TAT for the SOC, which is the same as the TAT of the largest core. The largest core is assigned a certain number of TAM lines, which depends on the size of the LFSR, such that its TAT cannot be further reduced.

The organization of the rest of this paper is as follows. Section II reviews relevant background material. Section III describes the proposed SOC testing approach. The associated static-scheduling algorithm is presented in detail in Section IV. Section V reports experimental results for static scheduling. Section VI presents an alternative optimization approach that combines dynamic test compression with the proposed test architecture. Simulation results for benchmark circuits are presented for this approach. Finally, Section VII concludes this paper.

II. Background

This section provides background material used for the rest of this paper.

A. Pareto-Optimal TAM Widths

As shown in Fig. 2, the TAT varies with the number of TAM lines (or TAM width) assigned to it as a "staircase" function, and decreases only at *Pareto-optimal* points, which are formally defined as follows: A solution to the wrapper design problem for Core i can be expressed as a two-tuple $(W_j, T_i(W_j))$, where

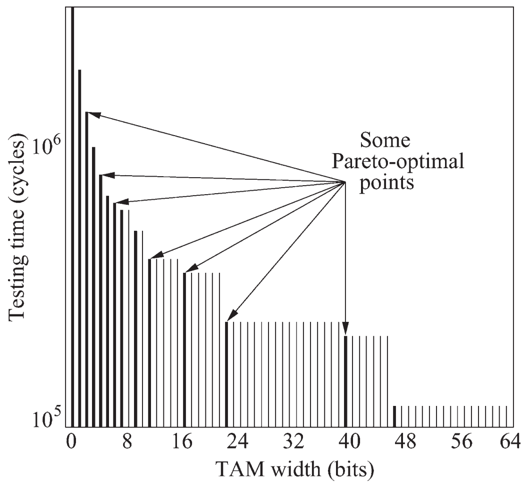


Fig. 2. Relationship between TAT and TAM width [21].

W_j is the TAM width supplied to the wrapper and $T_i(W_j)$ is the TAT of Core i with the given wrapper. A solution $(W_j, T_i(W_j))$ is *Pareto-optimal* if and only if there does not exist a solution $(W_k, T_i(W_k))$ such that $W_k \le W_j$ and $T_i(W_k) \le T_i(W_j)$, where at least one of the inequalities is strict. Intuitively, the steps at which the testing time decreases (as TAM width is increased) are the Pareto-optimal points. Only these Pareto-optimal TAM widths need to be considered when designing test wrappers. We use the *design_wrapper* algorithm from [21] to compute Pareto-optimal TAM widths for a given core.

For the rest of this paper, we use $W_{i,k}$ to denote the kth Pareto-optimal TAM width of Core i, $k = 1, 2, \ldots, N_i$, where N_i is the number of Pareto-optimal TAM widths of Core i. The TAT of Core i with TAM width $W_{i,k}$ is $T_i(W_{i,k})$. All Pareto-optimal TAM widths for Core i are sorted in an ascending order such that $\forall(k,l), 1 \le k, l \le N_i, l > k \Rightarrow W_{i,l} > W_{i,k}$.

B. TAT for a Core

Given a core, let $s_\mathrm{i}(s_\mathrm{o})$ be the length of its longest wrapper scan-in (scan-out) chain. The number of clock cycles required to apply p test patterns to this core is given by [21]

$$T = (1 + \max\{s_\mathrm{i}, s_\mathrm{o}\}) \cdot p + \min\{s_\mathrm{i}, s_\mathrm{o}\}. \tag{1}$$

Once a test pattern has been shifted into the core, in the next clock cycle, the core will capture the responses of the combinational parts to the scan cells. The "1+" part in (1)

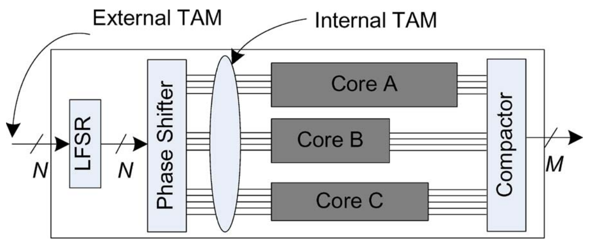


Fig. 3. Test architecture.

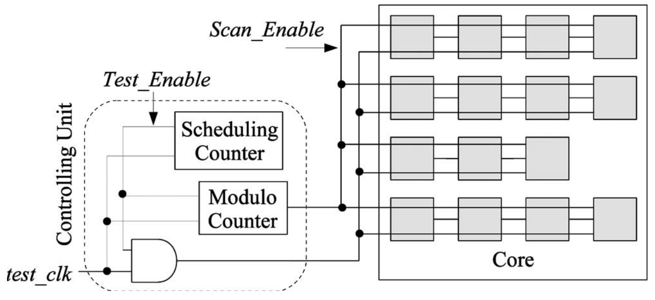


Fig. 4. Each core has a dedicated test control unit that provides the gated test clock and the scan_enable signals. Scheduling data for the core are stored in the scheduling counter.

corresponds to the clock cycles needed for response capture. While output responses of a pattern are shifted out, the next test pattern is shifted in at the same time. The "$\max\{s_i, s_o\}$" part in (1) reflects this fact.

III. Proposed Approach

An efficient LFSR reseeding technique is proposed in [20]. It allows the generation of a single scan slice from multiple seeds, or multiple scan slices from a single seed. An additional tester channel is needed to control when reseeding occurs. In this paper, without loss of generality, we choose to use the compression technique of [20] because of its high encoding efficiency. The proposed test-scheduling method can also be used with other linear-decompression-based compression techniques [22], [23].

A. Test Architecture

The architecture of the proposed approach is shown in Fig. 3. Each core is individually scheduled for test during one or more clock ranges. If core A is scheduled for test during clock range $[t_0, t_1)$, then A starts receiving data from the LFSR through the phase shifter at clock cycle t_0, and finishes scanning out the responses before clock cycle t_1. We refer to t_0 and t_1 as *start cycle* and *end cycle*, respectively. Outside $[t_0, t_1)$, core A is in the inactive mode. Therefore, each core should have a separate *Test_Enable* control signal, which is active only during the scheduled clock ranges. The *Test_Enable* signal is AND-ed with the system clock, as shown in Fig. 4. The *Test_Enable* signals are generated using on-chip counters according to the scheduling data that are also stored on-chip. Our experimental results show that in most cases, one core is assigned one clock range; hence, the storage size for the scheduling data is very

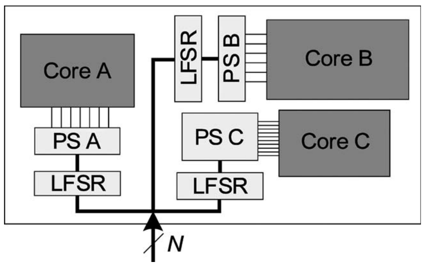


Fig. 5. Alternative test architecture to reduce routing overhead.

small. For handling test responses, any compaction scheme can be used.

Each core is associated with a modulo-$(\max\{s_i, s_o\} + 1)$ counter that controls when it should shift in test data, capture output responses, and shift out output responses. The output of the modulo counter is connected to the *Scan_Enable* inputs of all scan cells, as shown in Fig. 4. The output of the modulo counter is reset to zero in each capture cycle, incremented by one in each shift cycle, and again, reset to zero in the next capture cycle.

Another advantage of the proposed architecture is that the single LFSR can be arbitrarily duplicated for all or a set of cores to reduce the area overhead of global routing. Fig. 5 shows the case in which each core has its own LFSR. Consequently, the large phase shifter in Fig. 3 is split into smaller ones (shown as PS A, B, and C). Compared with the architecture shown in Fig. 3, which routes a huge number of wires from the phase shifter to the cores, the area overhead of global routing is significantly reduced since only a small number of wires need to be routed from test pins to the LFSRs.

As shown in Fig. 3, the number of internal TAM lines is no longer restricted by the number of scan input output (IO) pins of the SOC, which are used as scan chain inputs/outputs. Compared with existing test scheduling techniques [21], we have more freedom to increase the number of internal TAM lines. Each internal TAM line is connected to an output stage of the phase shifter, which is usually an XOR gate [24]. Therefore, in this paper, we assume there is no constraint on the number of internal TAM lines. The number of external TAM lines depends on the number of scan IO pins. In this paper, when we mention TAM lines without stating whether they are internal or external, we refer to internal TAM lines.

B. Equivalent Core

At any clock cycle, the LFSR expands its seed to test data, and simultaneously feeds multiple cores through the phase shifter. Each seed is calculated from care bits that belong to multiple cores. From the LFSR's point of view, the SOC is tested as a monolithic core, referred to as the *equivalent core* of the SOC. By carefully designing the TAM and test wrappers, together with proper test scheduling, an equivalent core can be obtained whose testing time is minimized. Thereafter, the LFSR reseeding technique of [20] is applied for the equivalent core. TAT is significantly reduced because: 1) multiple cores are tested in parallel and 2) when some cores are in the capture or

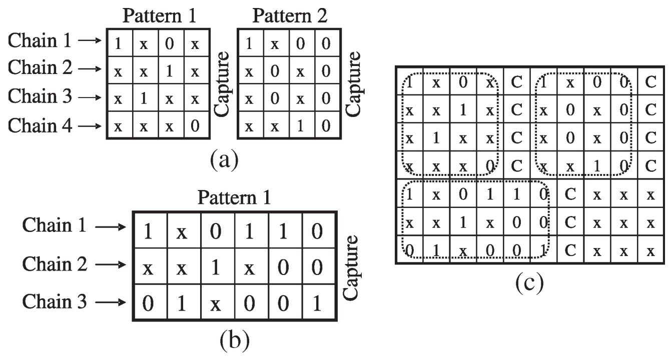


Fig. 6. Two cores and their equivalent core. (a) Core A. (b) Core B. (c) Equivalent Core.

inactive mode, other cores are in the shift mode and receiving data from the LFSR.

Fig. 6 shows two cores A and B and their equivalent core. In Fig. 6, each row represents a wrapper scan chain (WSC) and each column represents a scan slice. Core A has four WSCs and two patterns with each pattern having four scan slices. Core B has three WSCs and one pattern that has six scan slices. Both cores are scheduled for test starting from clock cycle 0. At clock cycle 5, Core A is in the capture mode (marked as "C" or "Capture") while core B continues receiving data. The equivalent core has seven WSCs and nine scan slices.

C. Problem Formulation

The LFSR reseeding technique of [20] requires that a seed encode at least one scan slice. This implies that if the maximum number of care bits for all scan slices of the equivalent core is $S_{\max}$, then the seed size should be $S_{\max} + m$, where m is small (preferably 20, see [25]). In this paper, we assume that $S_{\max}$ is a user-defined parameter. The proposed TAM, test wrapper, and test data compression cooptimization problem is referred to as $\mathcal{P}_{\mathrm{TWC}}$ (TWC stands for TAM, Wrapper, and Compression), and can be formally stated as follows.

$\mathcal{P}_{\mathrm{TWC}}$: Consider an SOC having $|\mathbf{C}|$ cores (where $\mathbf{C}$ is the set of cores). Given $S_{\max}$ and the test set parameters for each core, i.e., the number of input, output, and bidirectional terminals, and the test set with unspecified bits, determine the internal TAM width and a wrapper design for each core, and a test schedule to form an equivalent core, such that the testing time for the SOC (or the equivalent core) is minimized. The number of care bits in each scan slice of the equivalent core cannot exceed $S_{\max}$.

Ideally, given an equivalent core, if W tester channels are used to test it, where $W = S_{\max} + m$ is the seed size of the LFSR, the overall TAT is minimized. With fewer tester channels, sometimes the scan clock must be paused to wait for a new seed to be completely transferred. However, experimental results show that, particularly for large industrial circuits, most seeds can encode a sufficiently large number of scan slices, such that the next seed can be transferred on time. To improve encoding efficiency, a larger seed size $W' = kS_{\max} + m$, $k = 2, 3, \ldots$, can be used. In this case, each seed can encode at least k scan slices, and the ideal number of tester channels remains W.

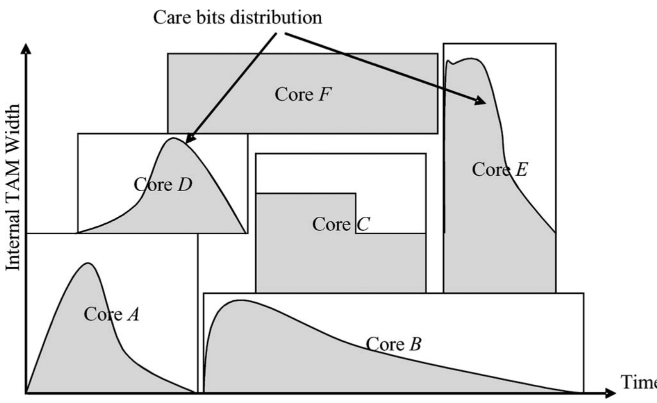


Fig. 7. Slice-based scheduling.

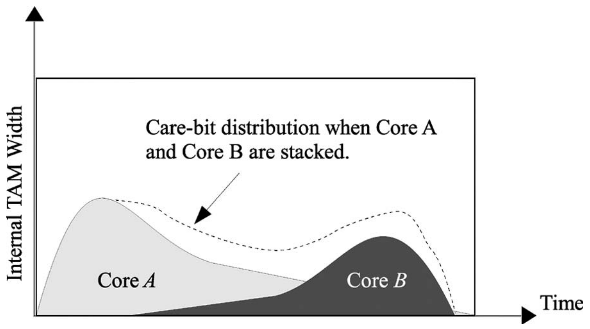


Fig. 8. Care bit distribution when two cores are partially stacked.

IV. Scheduling Algorithm

We next propose a scheduling algorithm, referred to as *TWCScheduler*. Most existing scheduling techniques work on a per-core basis, i.e., each core as a whole is viewed as a block and is packed into a rectangular bin [21]. *TWCScheduler*, as shown in Fig. 7, works on a per-slice basis. In Fig. 7, each core is shown as a rectangle. The height of the rectangle is the number of internal TAM lines assigned to the core, and the width is the corresponding TAT. The care bit distributions of each core are drawn in gray inside their rectangles. All cores that are in the shift mode at a given clock cycle t are "stacked" with each other. Cores are "stackable" at t only if their total number of care bits at t does not exceed $S_{\max}$. In Fig. 8, the care bit distribution when two cores A and B are partially stacked is shown in dashed line.

During the scheduling process, *TWCScheduler* may: 1) change the shape of the blocks, i.e., change the number of internal TAM lines assigned to each core; and 2) place the blocks at proper places, i.e., allocate clock ranges to test the cores. If necessary, *TWCScheduler* may vertically split a core into multiple blocks with identical heights, such that the core is tested during more than one clock range. This splitting action is referred to as *preemption*.

Before a core is scheduled, its test patterns are sorted in ascending or descending order according to the total number of care bits they have. This is motivated by the fact that, given two cores, if we sort the patterns of one core in an ascending order and patterns of the other core in a descending order, the two cores are more likely to be stackable, as shown in Fig. 8.

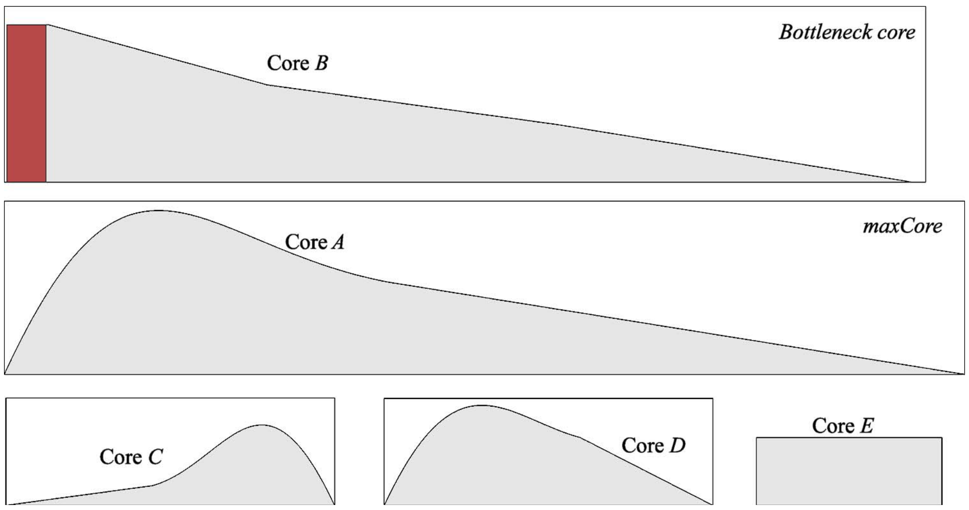


Fig. 9. Illustration of *maxCore*, bottleneck core (with highly specified patterns shown in dark), and other cores.

The high-level flow of *TWCScheduler* is shown in Procedure 1.

Procedure 1 High-level flow of *TWCScheduler*

```
1: Calculate Pareto-optimal TAM widths for each core;
2: Find maxCore;
3: Find bottleneck cores;
4: Preempt bottleneck cores;
5: Schedule maxCore;
6: Schedule other cores one by one;
```

A. *Identify maxCore*

Among all the cores, *TWCScheduler* first identifies one *maxCore*. Given $S_{\max}$, each Core i has a maximum acceptable Pareto-optimal TAM width, referred to as $W_{i,\max}$, such that if the TAM width supplied to Core i exceeds $W_{i,\max}$, there exists at least one scan slice that contains more than $S_{\max}$ care bits. Consequently, when Core i is assigned $W_{i,\max}$ TAM lines, its minimum TAT, referred to as $T_{i,\min}$, is achieved. Core j is the *maxCore* if and only if $\forall i \neq j$, $T_{i,\min} \leq T_{j,\min}$ ($T_{j,\min}$ is denoted as $T_{\min}$). Intuitively, $T_{\min}$ is the lower bound for the overall TAT for the SOC.

When the lower bound is achieved, an *optimal solution* to $\mathcal{P}_{\text{TWC}}$ is found. *TWCScheduler* always assigns to the *maxCore* its maximum Pareto-optimal TAM width, such that an optimal solution is achievable. Section V will show that for most cases an optimal solution can be found.

B. *Identify and Preempt Bottleneck Cores*

Next, *TWCScheduler* identifies bottleneck cores. A Core i is a *bottleneck core* if it satisfies $\forall W_{i,k} < W_{i,\max}$, $1 \leq k \leq N_i$, $T_i(W_{i,k}) > T_{\min}$. Given an SOC and $S_{\max}$, bottleneck cores may not always exist. *TWCScheduler* always supplies a bottleneck Core i with $W_{i,\max}$ TAM lines such that an optimal solution is still achievable.

Fig. 9 shows an example for an SOC consisting of five cores. Among these five cores, Core A is the *maxCore* because $T_{A,\min}$ is greater than $T_{\min}$ of all the other cores. Core B is a bottleneck core since although $T_{B,\min} < T_{A,\min}$, its testing time would be greater than $T_{A,\min}$ if the internal TAM width assigned to Core B is less than $W_{B,\max}$. Recall that $T_{B,\min}$ will not be achieved unless $W_{B,\max}$ bits of TAM lines are assigned to Core B. Cores C, D, and E are not bottleneck cores.

If a bottleneck Core i has some highly specified test patterns that have more than $S_{\max} - \delta$ care bits in some scan slices, where δ is another user-defined parameter, *TWCScheduler* will preempt this core. Those highly specified patterns are scheduled earlier than other patterns, which will be scheduled later together with other nonbottleneck cores. These patterns are shown in dark in Fig. 9.

The motivation for preemption is twofold: 1) Since highly specified patterns usually target more stuck-at faults, applying them first can potentially lead to a reduced average testing time if "abort-at-first-fail" test strategies are used; 2) since it is less likely that highly specified patterns can be simultaneously applied with other patterns from other cores, it will save CPU time by directly scheduling them at the beginning of the test session.

C. *Schedule maxCore*

TWCScheduler always attempts to make the overall TAT equal to $T_{\min}$, the shortest possible TAT for *maxCore*. This requires that *maxCore* and bottleneck cores be supplied with their maximum acceptable Pareto-optimal TAM widths. The proposed scheduling algorithm never decreases the TAM widths assigned to these cores.

If there exist highly specified patterns from bottleneck cores, these patterns are first scheduled, followed by *maxCore*; otherwise, *maxCore* is scheduled first. The patterns for *maxCore* and the highly specified patterns from all bottleneck cores are sorted in a descending order with regard to the their numbers of care bits.

For example, in Fig. 10, the highly specified patterns of Core B are shown in dark. These patterns are first applied to the SOC without being stacked with patterns from other cores; the remaining patterns of Core B are scheduled together with other cores. Therefore, core B is preempted.

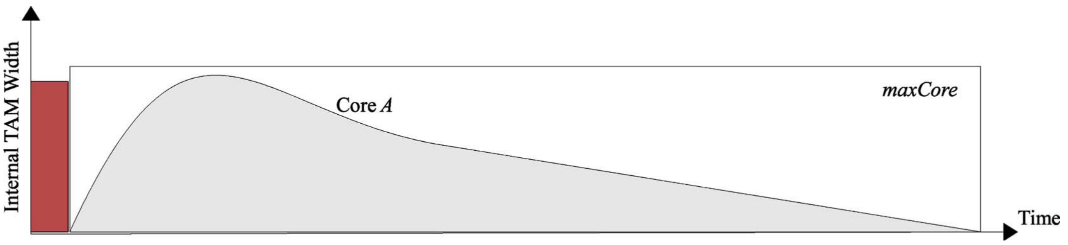


Fig. 10. Scheduling results after preempting bottleneck cores and scheduling *maxCore*.

TABLE I
DATA STRUCTURES

width(i)	Current internal TAM width assigned to Core i.
TAT(i)	TAT of Core i when supplied with *width*(i) TAM lines.
ncbCore(i, t)	Number of care bits in the t-th scan slice of Core i.
StartTime(i)	Latest start cycle assigned to Core i.
EndTime(i)	Latest end cycle assigned to Core i.
begun(i)	Boolean that indicates Core i has begun.

TABLE II
SUPPORTING PROCEDURES

sortPattern (i, *dir*)	Sorts patterns of Core i, the sort direction is specified by *dir* $\in$ {*DESC*, *ASC*}.
designWrapper (i, w)	Assigns w internal TAM lines to Core i, rearranges scan slices and updates *ncbCore*(i).
doSchedule (i, *start*, *end*)	Schedules Core i for test in clock range [*start*, *end*), and updates *timeLine*.

D. Schedule Remaining Cores

After *maxCore* and the highly specified patterns are scheduled, as shown in Fig. 10, the scheduling algorithm iterates over all the remaining cores and schedules them one by one in a random order using a greedy search strategy. For each of these cores, the scheduling algorithm attempts to schedule it such that the test of it can finish as early as possible, i.e., to find an optimal end time. Once a core is scheduled, its testing time will not be changed; the remaining cores might be stacked on top of it (Fig. 8 shows how two cores are stacked).

For a nonbottleneck core or a bottleneck core that is not preempted, an optimal end time can be found given its assigned TAM width and pattern sort direction (either ascending or descending). The scheduling algorithm iterates over all of the possible combinations of its Pareto-optimal TAM widths and pattern sort directions, and schedules this core using the earliest end time.

For a preempted bottleneck core, the scheduling algorithm will not decrease its assigned TAM width. Its remaining patterns are sorted in both directions and two end times can be obtained. The earlier one is used to schedule it.

E. Algorithm Implementation

TWCScheduler maintains an array *timeLine*, where *timeLine(t)* is the total number of care bits at clock cycle t from cores that are in the shift mode. Initially, *timeLine* contains all zeros. Whenever a core is scheduled, *timeLine* is updated to incorporate the care bits of this core. Once scheduling is finished, *timeLine(t)* becomes the number of care bits in the tth slice of the equivalent core.

Table I summarizes the data structures used in *TWCScheduler*. Table II lists important supporting procedures.

Procedure *trySchedule* is the most time-consuming and is shown in Procedure 2. It attempts to schedule Core i within [*start*, *end*) as early as possible. First, test patterns are sorted according to *dir* (Line 1). Then, Core i and *timeLine* are compared slice by slice to see if Core i can be scheduled starting from *startTime* (Lines 4–13). Initially, *startTime* is set to *start* (Line 2). If a conflict occurs (Line 8), *startTime* is incremented by 1 and the comparison is restarted (Line 9). If Core i can be scheduled, *trySchedule* calls *doSchedule* to record the scheduling result and to update *timeLine*, and returns 1 (Lines 14–17); otherwise, returns 0 (Lines 10, 18).

```
Procedure 2 trySchedule($i$, start, end, dir)
1: sortPattern($i$, dir);
2: $startTime = start$;
3: $currTime = startTime$; $currSlice = 0$;
4: while $currSlice < TAT(i)$ and $currTime < end$ do
5:     $ncb1 = timeLine(currTime)$;   $ncb2 = ncbCore(i, currSlice)$;
6:     if $ncb1 + ncb2 \le S_{\max}$ then
7:         currTime ++; currSlice ++;
8:     else
9:         $currSlice = 0$; startTime ++;
10:        if $startTime + TAT(i) \ge end$ then return 0;
11:        $currTime = startTime$;
12:    end if
13: end while
14: if $currSlice == TAT(i)$ then
15:     doSchedule($i$, startTime, $startTime + TAT(i)$);
16:     return 1;
17: end if
18: return 0;
```

Procedure *TWCScheduler* is shown in Procedure 3. Lines 1–2 are initialization operations and have been discussed earlier in Section IV. In Lines 3–10, bottleneck cores are preempted before *maxCore* is scheduled in Lines 11–12. The patterns of *maxCore* and all bottleneck cores are sorted in a descending order with regard to the their numbers of care bits in favor of "abort-at-first-fail" strategies.

Lines 13–33 form the main loop that schedules all other cores except *maxCore*. If a Core i is a bottleneck core and has been preempted, *trySchedule* tries to schedule its remaining patterns after *EndTime(i)*, when its heavily specified patterns have been applied (Line 15). If a Core i is a nonbottleneck core and/or has not begun (Line 16), a greedy search strategy is performed to find a schedule for it. We iterate over its Pareto-optimal TAM widths in a descending order (Line 18), and assign w TAM lines to it (Line 19). For each w, *trySchedule* is called twice with different sort directions (Lines 21–28). The purpose of this greedy strategy is to find a Pareto-optimal TAM width w and a sort direction that minimize *EndTime(i)* (Line 23–27). When a solution is found that is better than previous solutions, it is saved in Line 25. When the search process is finished, the known best solution is restored and *timeLine* is updated accordingly in Line 31.

Some early termination conditions are exploited to quickly terminate the greedy search. Line 20 checks if the current w will result in a TAT longer than *minTime*. If so, then w and other smaller TAM widths will not result in better solutions and should not be tried. Line 26 checks if *EndTime(i)* equals to its TAT, which implies that the core has been assigned a start cycle of zero. If so, then we have found a best solution for this core. Line 29 checks if the known best solution has been obtained with a Pareto-optimal TAM width larger than w. If this happens, then in most cases other smaller widths will not result in better solutions, since they usually result in much longer TATs.

Procedure 3 *TWCScheduler*($\mathbf{C}$, $S_{\max}$, δ)

```
1: Calculate Pareto-optimal TAM widths for each core;
2: Find maxCore; Find bottleneck cores;
3: $currTime = 0$;                    //Preempt bottleneck cores
4: for all Core $i$ that is a bottleneck core do
5:    sortPattern($i$, DESC); designWrapper($i, W_{i,\max}$);
6:    Find all patterns of Core $i$ that have at least one scan
      slice with more than $S_{\max} - \delta$ care bits;
7:    $length$ = testing time to apply those patterns;
8:    doSchedule($i$, $currTime$, $currTime + length$);
9:    $begun(i) = 1$; $currTime = currTime + length$;
10: end for
11: $j$ = index of $maxCore$;             //Schedule maxCore
12: designWrapper($j, W_{j,\max}$); trySchedule($j, 0, \infty$, DESC);
13: for all Core $i$ in $|\mathbf{C}|$, $i \neq j$ do
14:   if $begun(i) == 1$ then
15:      trySchedule($i$, EndTime(i), $\infty$, DESC);
16:   else
17:      $minTime = \infty$; $minW = -1$;
18:      for $k = N_i$ to 1 do
19:         $w = W_{i,k}$; designWrapper($i$, $w$);
20:         if $TAT(i) \geq minTime$ then break;
21:         for $dir \in \{DESC, ASC\}$ do
22:            $r = trySchedule(i, 0, minTime, dir)$;
23:            if $r == 1$ and $EndTime(i) < minTime$ then
24:               $minTime = EndTime(i)$; $minW = w$;
25:               $minDir = dir$; saveSchedule(i);
26:               if $EndTime(i) == TAT(i)$ then break;
27:            end if
28:         end for //dir
29:         if $minW > w$ then break;
30:      end for //$w$
31:      restoreSchedule(i);
32:   end if
33: end for //Core $i$
```

F. CPU Time Optimization

Procedure *trySchedule* compares Core i against array *timeLine* slice by slice, trying to find a proper start clock cycle for Core i. For large industrial circuits, this process may take several hours for a midsized core (e.g., cores listed in Table V in Section V). To optimize *trySchedule*, whenever *startTime* is changed (Lines 2 and 9 of *trySchedule*), a new procedure *checkStart* is called to quickly check if conflicts will occur. If conflicts occur, *checkStart* returns zero and *startTime* is directly incremented by one, without entering the time-consuming loop in Lines 4–13. To call *checkStart*, the following code snippet is inserted after Lines 2 and 9, respectively.

while $checkStart(i, startTime) == 0$ **do** *startTime* ++;

Procedure *checkStart* (shown in Procedure 4) uses three caches for quick identification of conflicts. Each cache is a 1-D array that references to a series of slices or elements in *timeLine*.

1) Cache A stores all scan slices of Core i that have at least δ care bits.
2) Cache B stores all elements of *timeLine* that have at least $S_{\max} - 3$ care bits.
3) Cache C stores all elements of *timeLine* that have at least $S_{\max} - \delta$ care bits.

The constants (3 and δ) are chosen through extensive experiments.

Cache A is updated when Core i is assigned a new number of internal TAM lines in Procedure *designWrapper*. Caches B and C are updated when *timeLine* is updated in Procedure *doSchedule*. Since the time cost to update these caches is linear to the size of the core, and the update operations do not occur frequently, the cost to maintain these caches are trivial.

Cache B and C can be viewed as Level 1 and 2 caches of *timeLine*. We do not remove duplicate elements from the Level 2 cache that also belong to the Level 1 cache. To check Cache A (B or C) for conflicts, each slice in it is compared against the corresponding slice in *timeLine* (*ncbCore*). If the total number of care bits is greater than $S_{\max}$, then a conflict occurs. In most cases, Cache A contains fewer elements and is first checked.

This optimization technique significantly accelerates Procedure *TWCScheduler*. Without optimization, the scheduler does not finish after 20 h for the SOC described in Table V. After optimization, it only takes about 30 min.

Procedure 4 checkStart(i, *startTime*)

```
1: check elements in Cache $B$ for conflicts;
2: if Cache $A$ contains fewer elements than Cache $C$ then
3:    check elements in Cache $A$ for conflicts;
4:    check elements in Cache $C$ for conflicts;
5: else
6:    check elements in Cache $C$ for conflicts;
7:    check elements in Cache $A$ for conflicts;
8: end if
```

TABLE III
BENCHMARK SOC d695

Core	No. of Primary Inputs	No. of Primary Outputs	No. of Scan Cell	No. of Patterns	No. of Scan Chain	Max Scan Chain Length	Min Scan Chain Length	No. of Care Bits
s38584	38	304	1,426	136	32	45	44	35,287
s38417	28	106	1,636	99	32	51	51	52,582
c6288	32	32	0	29	0	0	0	910
c7552	207	108	0	122	0	0	0	10,831
s838	35	35	32	86	1	32	32	2,344
s9234	36	39	211	159	4	54	52	10,601
s13207	62	152	638	236	16	40	39	11,313
s15850	77	150	534	126	16	34	33	12,657
s5378	35	49	179	111	4	46	44	6,505
s35932	35	320	1,728	16	32	54	54	18,251

TABLE IV
RESULTS FOR d695

Core	S_{max}= 32				S_{max}= 64			
	TAM	TAT	Start	End	TAM	TAT	Start	End
s38584	32	7,662	0	830	39	6,301	0	6,301
			1,415	8,293				
s38417	32	5,599	830	1,389	32	5,599	0	5,599
			2,520	7,615				
c6288	8	149	473	622	11	119	0	119
c7552	16	1,715	3,009	4,724	42	735	95	830
s838	3	2,870	955	3,825	3	2,870	0	2,870
s9234	5	8,799	2,161	10,960	5	8,799	0	8,799
s13207	20	9,716	1,333	**11,049**	20	9,716	0	**9,716**
s15850	21	4,444	3,454	7,898	21	4,444	5	4,449
s5378	5	5,263	4,551	9,814	5	5,263	11	5,274
s35932	19	1,852	7,264	9,116	38	934	769	1,703

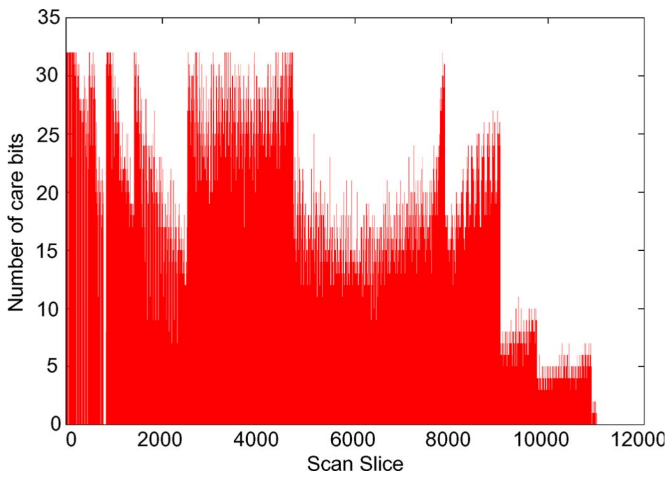


Fig. 11. Care bit distribution over scan slices of the equivalent core of d695.

V. EXPERIMENTAL RESULTS

First, we run *TWCScheduler* on the d695 benchmark SOC [21]. Test patterns for the cores are compacted by Mintest [26]. Table III lists detailed information about d695. We assume that the internal scan chains of the cores cannot be modified.

Scheduling results for d695 with $S_{\max} = 32, 64$ and $\delta = 10$ are reported in Table IV. Column "*TAM*" reports the number of internal TAM lines assigned to each core. Column "*TAT*" shows the TAT. Clock ranges assigned to each core are listed in Columns "Start" and "End." Two bottleneck cores, s38584 and s38417, are preempted when $S_{\max} = 32$. Core s13207 is *maxCore* for both values of $S_{\max}$. The overall TAT of the SOC is the same as the end cycle of s13207 (in bold). The CPU time is less than 1 s. The care bit distribution over scan slices of the resulting equivalent core is shown in Fig. 11.

Next, we present results for an SOC named NIM that consists of nine real-life industrial cores. Table V describes these cores. For cores C1–C4 and C7–C9, primary inputs and outputs are scannable and are part of the scan chains. Therefore, the numbers of inputs or outputs for these cores are listed as zero.

Table VI reports scheduling results for NIM with $S_{\max} = 16, 32, 48, 64$ and $\delta = 10$. The CPU times are also listed. Table VI is similar in format to Table IV. Row "CPU time" lists the execution time in minutes and seconds. As shown from the table, smaller values of $S_{\max}$ may result in much higher CPU time. Unlike d695, the scheduler finds no bottleneck cores and does not perform preemption. For all cases, an optimal solution has been found. When $S_{\max} = 64$, the exact test data volume is 46 049 951 b, if the LFSR size is 1044 ($kS_{\max} + 20$, $k = 16$, see Section III) stages and 64 ($532/k$) ATE channels are used.

The following interesting observation can be made for NIM, but not for d695. The rate at which the TAT for the SOC decreases is relatively more compared to the rate at which $S_{\max}$ increases. This is because the test sets for the industrial circuits have lower care bit densities compared to the test sets for the International Symposium on Circuits and Systems (ISCAS) circuits in d695. A small increment in $S_{\max}$ will enable a relatively large increment in the total number of WSCs that can be driven by the LFSR in parallel. We also note that the solution obtained with $S_{\max} = 64$ is a particularly noteworthy optimal solution. The *maxCore*, C8, has at most 100 scan chains (Table V). If a smaller $S_{\max}$ is used, i.e., $48 < S_{\max} < 64$, the overall TAT may still be 4 110 383 cycles, but the TATs for the other cores become higher.

Next, we compare this paper to some related prior work, as listed in Table VII. To compare with that in [18], we only considered the five cores for d695 that were used in [18]. We carried out the same set of experiments that are reported in Table IV. The resulting TAT for the proposed work is the same as that when all cores are considered, i.e., 11 049 clock cycles when $S_{\max} = 32$. For 32 scan chains, the TAT reported by

TABLE V
BENCHMARK SOC NIM

Core	No. of Primary Inputs	No. of Primary Outputs	No. of Scan Cell	No. of Patterns	No. of Scan Chain	Max Scan Chain Length	Min Scan Chain Length	No. of Care Bits
C1	0	0	798	189	4	200	198	6,527
C2	0	0	4,990	310	13	480	5	144,687
C3	0	0	65,426	1,396	171	400	74	1,287,102
C4	0	0	259,493	11,544	999	260	45	5,311,612
C5	1,596	1,800	43,414	1,529	140	311	310	1,078,829
C6	297	288	26,970	4,900	100	295	294	1,796,157
C7	0	0	81,008	2,859	128	662	456	5,969,376
C8	0	0	22,205	18,207	100	227	213	7,045,053
C9	0	0	108,863	18,142	512	214	211	18,259,914

TABLE VI
RESULTS FOR NIM

Core	S_{max}= 16				S_{max}= 32				S_{max}= 48				S_{max}= 64			
	TAM	TAT	Start	End	TAM	TAT	Start	End	TAM	TAT	Start	End	TAM	TAT	Start	End
C1	4	38,189	0	38,189	4	38,189	0	38,189	4	38,189	0	38,189	4	38,189	0	38,189
C2	12	149,590	373	149,963	12	149,590	0	149,590	12	149,590	0	149,590	12	149,590	0	149,590
C3	34	2,778,632	87,891	2,866,523	95	1,110,614	268	1,110,882	168	560,196	23,506	583,702	168	560,196	0	560,196
C4	999	3,013,244	2,775,231	5,788,475	999	3,013,244	202,141	3,215,385	999	3,013,244	17,930	3,031,174	999	3,013,244	0	3,013,244
C5	11	6,291,340	103,049	6,394,389	22	3,321,629	191	3,321,820	33	2,373,029	1,234	2,374,263	49	1,425,959	0	1,425,959
C6	12	11,546,755	989,641	12,536,396	24	5,778,278	11	5,778,289	41	4,332,483	2,970	4,335,453	47	2,891,589	0	2,891,589
C7	48	5,619,899	5,493,910	11,113,809	128	1,896,179	2,663,063	4,559,242	128	1,896,179	82,075	1,978,254	128	1,896,179	0	1,896,179
C8	60	7,986,403	10,922,164	18,908,567	100	4,110,383	3,902,623	8,013,006	100	4,110,383	3,218,939	7,329,322	100	4,110,383	0	**4,110,383**
C9	84	26,996,783	0	**26,996,783**	247	11,557,090	0	**11,557,090**	387	7,710,774	0	**7,710,774**	512	3,900,744	1,072	3,901,816
CPU time	26 min				32 min				20 min				65 sec			

TABLE VII
COMPARISON RESULTS

	Proposed	[19]		[20]	[27]
		seed-only	seed-mux		
S_{max}	32	-	-	-	
ATE Chn	34	-	-	-	64
LFSR Size	532	-	-	-	
TAT	11,049	11,658	9,612	>50,000	9,869
Data Vol	181,821	419,688	442,152	-	-
Pat Num	1,120	-	-	-	881

[18] is 11 658 clock cycles (for the "seed-only" variant) and 9612 clock cycles (for the "seed-mux" variant) for Mintest-compacted test patterns. The number of ATE channels is not reported in [18]. The exact test data volume is 181 821 b (the LFSR size is 532 stages and there are 34 ATE channels). The test data volume reported in [18] is 419 688 b (seed-only) and 442 152 b (seed-mux). The TAT reported in [19, Fig. 5] is higher than 50 000 clock cycles when apparently 32 internal scan chains are used.

We also compare with the TAM optimization and test scheduling techniques mentioned in [27], which do not use compression. The best TAT reported in [27] for d695 with a TAM width of 64 b is 9869 cycles. The TAT achieved by the proposed work is 11 407 cycles when $S_{\max} = 32$ (with $S_{\max} + m$ ATE channels). Although the TAT is slightly higher, the proposed method applies 1120 test patterns to the cores, while the TAT in [27] is obtained for only 881 patterns. More test patterns are expected to result in higher test quality.

VI. DYNAMIC ATPG AND COMPRESSION PROCEDURE

The optimization technique presented in Sections III–V is based on a static test compaction and compression approach in that it requires a predetermined set of test cubes for each core. The major drawback of using predetermined test cubes is that it usually results in larger test sets, since once a test cube is generated, it cannot be randomly filled to detect more faults. Although a few sophisticated algorithms such as [26] can produce highly compacted test sets, they are not implemented in most commercial ATPG tools, and hence, it is not known if they can handle industrial designs with reasonable CPU time.

In this section, we present a dynamic ATPG and test-compression approach for the test architecture shown in Fig. 3. Note that this dynamic approach cannot handle IP cores whose structural information is not available. Therefore, we cannot apply the dynamic method to the NIM SOC, for which we are only provided the test data for the cores.

To ensure that the optimization method is scalable, we make the following assumptions: 1) All cores are tested starting from time zero and hence the test control scheme in Fig. 4 only stores information on when the testing of the corresponding core is completed; 2) the internal scan chain structure in each core cannot be altered; and 3) dedicated IO WSCs are created for PIs and POs. Each IO WSC consists of no internal scan cells and cannot be longer than the longest internal scan chain. This assumption implies that the number of clock cycles to apply a test pattern to a core is equal to the length of its longest scan chain plus one capture cycle.

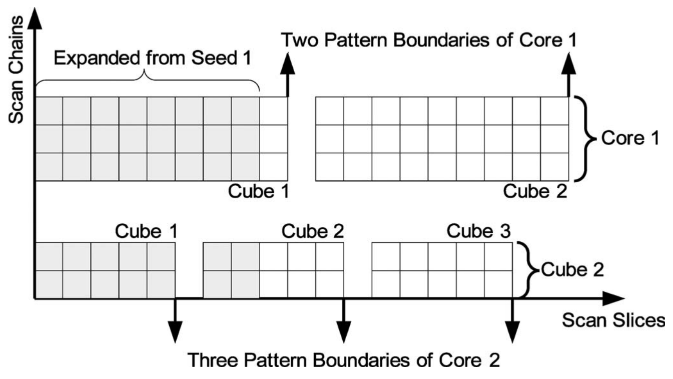


Fig. 12. Illustration of the dynamic ATPG and compression procedure.

A. *Proposed Algorithm*

Similar to existing dynamic test-compaction methods [4], [28], test cubes are dynamically generated and merged with other existing test cubes. When a newly generated test cube is compacted, care must be taken to ensure that each scan slice applied to the "Equivalent Core" (defined in Section III) contains no more than $S_{\max}$ care bits. Once a certain number of scan slices with sufficient care bits to compute a new LFSR seed are obtained, these slices are randomly filled by the LFSR and applied to the "Equivalent Core." If these slices cross test pattern boundaries for some cores, as shown in Fig. 12, fault simulation is performed for these cores using the newly generated test patterns and faults detected by these patterns are dropped. This dynamic ATPG and compression procedure continues until satisfactory fault coverage is obtained for all the cores.

Procedure 5 High-level flow of the dynamic method

```
1: while (1) do
2:   $numDone = numAtpgDone = 0$;
3:   $numCore =$ the number of cores;
4:   for (i = 0 to numCore-1) $tag[i] = 0$;
5:   $newPatCnt = 0$;
6:   //Stage 1: generate and merge test cubes
7:   while $(numCore > 0)$ do
8:     for all Core $i$ do
9:       if $tag[i] == 1$ then continue
10:      if $done[i] == 1$ then
11:        $tag[i] = 1$; numDone++; numCore–; continue
12:      end if
13:      if $atpgDone[i] == 1$ then
14:        if $hasUndetFlts[i] == 0$ then numAtpgDone++;
15:        $tag[i] = 1$; numCore–; continue;
16:      end if
17:      while (1) do
18:        Try to generate a new test cube;
19:        if no cube generated then
20:          $atpgDone[i] = 1$;
21:          if $hasUndetFlts[i] == 0$ then numAtpgDone++;
22:          $tag[i] = 1$; numCore–; break;
23:        else
24:          Try to merge the newly generated cube;
25:          if can be merged then
26:            newPatCnt++; break; //goto the next core.
27:          else
28:            Reject this cube and save the faults detected by it;
29:            $hasUndetFlts[i] = 1$;
30:            nReject[i]++;
31:            if nReject[i] reaches a user-defined up limit then
32:              $nReject[i] = 0$; $tag[i] = 1$; numCore–; break;
33:            end if
34:          end if //merge cube
35:        end if //if new cube generated
36:      end while //cube generation loop
37:    end for //for all cores
38:  end while //while (numCore)
39:  if $numDone ==$ the number of cores break;
40:  $noPatRound = newPatCnt == 0 ? noPatRound + 1 : 0$;
41:  //Stage 2: compression and fault simulation
42:  $minTime =$ the earliest pattern boundary time among all the cores;
43:  GetSeed(minTime, numDone, numAtpgDone, noPatRound);
44:  if A new seed is generated then
45:    Expand this seed to obtain fully specified test patterns;
46:    for all Core $i$ do
47:      if $done[i] == 1$ then continue;
48:      $nReject[i] = 0$; Run fault simulation;
49:      if $atpgDone[i] == 1$ then
50:        if $hasUndetFlts[i] == 1$ then $atpgDone[i] = 0$; restore the faults saved in 28;
51:        else if No more not simulated patterns then $done[i] = 1$;
52:      end if
53:      if new patterns simulated then $hasUndetFlts[i] = 0$;
54:    end for
55:  else
56:    for all Core $i$ do
57:      if $done[i] == 1$ then continue;
58:      $nReject[i] = 0$;
59:      if $atpgDone[i] == 1$ and $hasUndetFlts[i] == 1$ then
60:        $hasUndetFlts[i] = 0$; $atpgDone[i] = 0$;
61:        restore the faults saved in 28;
62:        Adjust the pattern storage queue for Core $i$ such that the first test cube ewly generated in proc:dynamic:atpg is appended to the end of the queue, instead of being merged with existing unfilled cubes;
63:      end if
64:    end for
65:  end if
66: end while
```

TABLE VIII
VARIABLES USED IN PROCEDURE 5

done[i]	Core i has been successfully tested, i.e., a satisfactory fault coverage has been achieved and all test cubes have been compressed.
atpgDone[i]	No more test cubes can be generated for Core i during the current execution of Stage 1.
hasUndetFlts[i]	Core i has undetected faults because some of its test cubes are rejected in line 28.
tag[i]	Core i is tagged to be skipped during the current execution of Stage 1.
nReject[i]	The number of test cubes of Core i that are rejected in line 28.
numDone	The total number of cores whose *done* flags are set.
numAtpgDone	The total number of cores whose *atpgDone* flags are set.
newPatCnt	The number of test cubes that are generated during the current execution of Stage 1.
noPatRound	The number of consecutive executions of Stage 1 that lead to no newly generated test cubes.
minTime	The earliest pattern boundary time among all the cores.

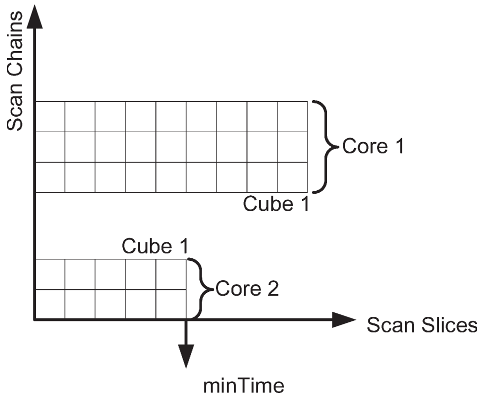


Fig. 13. Schedule after the first execution of Stage 1: Two test cubes are obtained.

Procedure 5 provides a detailed description for the proposed dynamic ATPG and compression method. Table VIII lists the supporting variables that are used throughout Procedure 5.

The whole procedure consists of two stages. In Stage 1 (Lines 6–38), all the cores are iterated one by one. In each iteration, one new test cube is generated (Line 18) and merged to existing test cubes that are not compressed yet (Line 24). If no more cubes can be generated or merged, the corresponding core is tagged (Lines 20–22, 28–33) and skipped during later iterations (Lines 9–16). For example, Fig. 13 shows how the SOC in Fig. 12 is scheduled after the first execution of Stage 1: two test cubes are obtained for the two cores by merging one or more test cubes returned in Line 18.

After one execution of Stage 1 is finished, the earliest pattern boundary time among all the cores *minTime* is computed in Line 42. In Fig. 13, *minTime* is marked by a downward arrow.

In Stage 2 (Lines 41–65, the test cubes that are generated during Stage 1 are compressed and fault simulation is performed. In Line 43, a seed is obtained from the existing uncompressed test cubes obtained during the previous executions of Stage 1. Line 43 ensures that no scan slices after *minTime* is used to derive the seed. Otherwise, as can be shown in Fig. 13, a new seed might be generated if scan slices after *minTime* are included; hence, no more test cubes can be appended to Cube 1 of Core 2, since the scan slices after *minTime* would be fully specified by expanding the seed.

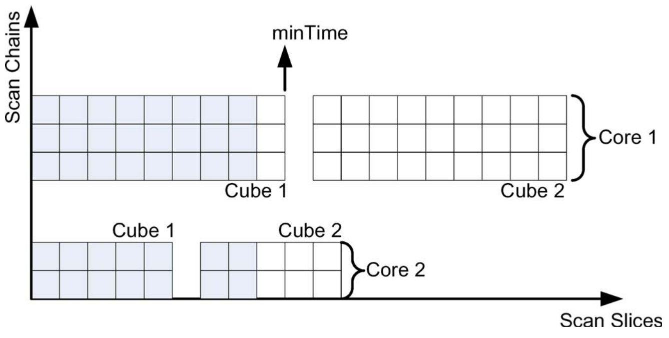


Fig. 14. Schedule after the second execution of Stage 1.

TABLE IX
RESULTS FOR d695_REDUCED: TetraMAX ATPG

Core	Pat. Len	Slice	TetraMAX ATPG		
			P	TD	TAT
s38584	1,464	46	146	213,744	6,716
s38417	1,664	56	101	168,064	5,656
s13207	700	41	270	189,000	11,070
s15850	611	35	134	81,874	4,690
SoC	-	-	651	652,682	28,132

P: No. of test patterns; TD: Test-data volume (bits); TAT: Test-application time (cycles)

TABLE X
RESULTS FOR d695_REDUCED: DYNAMIC ATPG AND COMPRESSION

Core	$S_{max} = 32$				$S_{max} = 64$			
	P	Seed	TAT	TE	P	Seed	TAT	TE
s38584	206	-	9,476	-	201	-	9,246	-
s38417	214	-	11,984	-	212	-	11,872	-
s13207	297	-	12,177	-	291	-	11,931	-
s15850	158	-	5,530	-	158	-	5,530	-
SoC	875	2,709	12,177	153,045	862	1,252	11,931	117,099

Seed: No. of LFSR seeds; TE: Encoded test-data volume (bits);

For better compression ratio, in most conditions, Line 43 will not return a new seed until there exist sufficient number of care bits in these test cubes. For example, in Fig. 13, a new seed is not generated from time 0 to *minTime* because the number of care bits in scan slices 0 to *minTime* is much less than $S_{\max}$. Hence, in the example of Fig. 13, no seed is generated and no fault simulation is performed during the first execution of Stage 2. It is also shown in Fig. 12 that the first seed (Seed 1) is generated from scan slices 0 to *minTime*+3 (this is done after the second execution of Stage 1). The first seed cannot cover scan slice *minTime*+4, otherwise there would be more than $S_{\max}$ care bits.

However, Line 43 will always return a new seed regardless the number of care bits when: 1) *numDone*+*numAtpgDone* equals the total number of cores or 2) *noPatRound* exceeds a user-defined upper limit. Condition 1) is triggered when no more test cubes can be generated. Condition 2) is triggered

TABLE XI
RESULTS FOR IWLS-4: DYNAMIC AND NONDYNAMIC ATPG

Core	No. of Primary Inputs	No. of Primary Outputs	No. of Scan Cells	No. of Scan Chain	Max Chain Length	Min Chain Length	No. of Faults	FC%	Dynamic ATPG			Non-dynamic ATPG		
									P	TD	TAT	P	TD	TAT
ac97_ctrl	56	47	2,199	15	151	141	57,597	99.54%	50	112,750	7,600	228	514,140	34,656
DMA	683	259	2,189	20	144	136	92,986	87.99%	266	763,952	38,570	880	2,527,360	127,600
pci_bridge32	160	201	3,359	20	176	175	91,772	99.04%	200	703,800	35,400	1,047	3,684,393	185,319
usb_funct	114	108	1,746	20	93	93	57,956	99.03%	119	221,340	11,186	210	390,600	19,740
SoC	-	-	-	-	-	-	-	-	635	1,801,842	92,756	2,365	7,116,493	367,315

P: No. of test patterns; TD: Test-data volume (bits); TAT: Test-application time (cycles)

when no more test cubes can be merged after a certain number of executions. Both conditions prevent potential dead loops.

Stages 1 and 2 are inside the same loop and are executed alternatively, until all cores have been marked as *done* (Line 39), i.e., satisfactory fault coverage has been reached for each core and all test cubes have been compressed.

Fig. 14 shows how test scheduling is carried out for the SOC after the second execution of Stage 1. Two more test cubes are obtained, and the variable *mintime* is moved to the first pattern boundary of Core 1. During the second execution of Stage 2, Seed 1 is generated and fault simulation is performed for Core 2.

B. Experimental Results

To evaluate the effectiveness of the proposed dynamic ATPG and compression method, we have developed an experimental environment based on the Synopsys TetraMAX tool. A C++ program was developed to implement the algorithm. This program communicates with TetraMAX via UNIX named pipes for test-pattern generation and fault simulation. A TCL script is executed within TetraMAX to serve requests from the C++ program. A dedicated instance of TetraMAX is required for each core. Due to the limited availability of TetraMAX licenses, we used a reduced version of the d695 SOC that only consists of four cores: s38584, s38417, s13207, and s15850.

We first use TetraMAX to generate fully specified and compacted test patterns using the commands "set_atpg -merge high -fill random" and "run_atpg -auto_compression." Table IX lists the results. The column "Slice," i.e., the number of clock cycles to apply one pattern to the core, is equal to column "Max Scan Chain Length" in Table III plus one. We let the TAT of each core equal the product of "Slice" and the number of patterns (column "P"), i.e., the time used to shift out the test response of the last pattern is ignored. For the entire SOC, a total of 651 test patterns are applied to the cores. The test data volume "TD" is 652 682 b, and the overall TAT is 28 132 cycles. To derive the overall TAT in Table X, we assume that the cores are tested serially and that sufficient ATE channels are available to drive all the WSCs.

Comparing Table IX with Table III, we note that the number of fully specified test patterns generated by TetraMAX is even larger than the number of test cubes generated by MinTest.

The results obtained using the proposed dynamic approach with $S_{\max} = 32$ and $S_{\max} = 64$ are shown in Table X. The number of LFSR stages is equal to $S_{\max} + 20$. As shown from Tables IX and , the TAT achieved by the dynamic approach is approximately 42% of the overall TAT in Table IX. The compression is 77% and 82% for $S_{\max} = 32$ and $S_{\max} = 64$, respectively. This experiment indicates that larger LFSR size results in higher encoding efficiency and higher compression ratio.

We next compare the proposed dynamic approach with the proposed static scheduling algorithm. For the reduced d695 SOC, the static algorithm yields similar results, as shown in Table IV. The overall TAT is still 11 049 and 9716 for $S_{\max} = 32$ and $S_{\max} = 64$, respectively. The TAT achieved by the dynamic approach is 10%–20% higher than the TAT achieved by the static approach. However, since the underlying ATPG engines are different for the two approaches, this difference is not unexpected. For $S_{\max} = 32$, the test-data volume achieved by the static approach is 129 685 b with a 532-stage LFSR and 34 ATE channels, and 255 813 b with 52-stage LFSR and 34 ATE channels.

In summary, for the reduced d695 core, the dynamic approach yields similar results compared with the static approach. However, for larger industrial designs, since the test cubes usually contain much less care bits than the ISCAS benchmark circuits, and since commercial ATPG tools are most likely to be used instead of MinTest, we expect that the dynamic approach will find wider applications and yield better results.

Next, we use another SOC [referred to as International Workshop on Logic and Synthesis (IWLS)-4], which we have crafted using four midsized IWLS benchmark circuits [29], to compare the effectiveness of our dynamic and static scheduling methods. Table XI lists the circuit information and TetraMAX ATPG results for the four cores. For dynamic ATPG, TetraMAX commands "set_atpg -merge high -fill random" and "run_atpg -auto" are used. The nondynamic ATPG test cubes, generated using commands "set_atpg -merge low" and "run_atpg," are used for the static method. We also tried "set_atpg -merge medium," but it yielded almost fully specified test cubes that cannot be effectively compressed. As shown from Table XI, nondynamic ATPG generated significantly larger test sets.

Table XII lists the results of the dynamic- and static-scheduling methods. Compared with the dynamic ATPG test patterns, the dynamic method achieves 6.37× and 5.71× reduction in test data volume (equal to TD/TE) for the two reported cases ($S_{\max} = 64$ and $S_{\max} = 128$). Note that the TE values reported in Table XII include the control data corresponding

TABLE XII
RESULTS FOR IWLS-4: DYNAMIC AND STATIC SCHEDULING

Core	Dynamic								Static							
	S_{max}= 128, LFSR=148				S_{max}= 64, LFSR=84				S_{max}= 128, LFSR=148				S_{max}= 64, LFSR=84			
	P	Seed	TAT	TE	P	Seed	TAT	TE	TAM	TAT	Seed	TE	TAM	TAT	Seed	TE
ac97_ctrl	121	-	18,392	-	124	-	18,848	-	15	34,087	-	-	15	34,087	-	-
DMA	792	-	114,840	-	867	-	125,715	-	20	127,744	-	-	20	127,744	-	-
pci_bridge32	511	-	90,447	-	510	-	90,270	-	20	185,495	-	-	20	185,495	-	-
usb_funct	371	-	34,874	-	392	-	36,848	-	20	19,833	-	-	20	19,833	-	-
SoC	372	1,144	114,840	284,152	1,893	2,262	125,715	315,723	-	185,495	2,994	628,607	-	185,495	6,278	712,847

Seed: No. of LFSR seeds; TE: Encoded test-data volume (bits);

to TAT. Since the number of LFSR seeds is much smaller than the magnitude of the TAT, the control data contain long runs of consecutive 0 s and can be further compressed using ATE pattern repeat [30]. If we exclude the control data, the reduction in test data volume increases to 10.64× and 9.48×, respectively.

Compared to the nondynamically compacted baseline ATPG method, static scheduling yields 11.32× and 9.98× reduction in test data volume. However, compared with the dynamic-scheduling method, the performance of the static method is less impressive. This can be attributed to the fact that the nondynamic ATPG test cubes are not optimized. After static scheduling, testing of all the cores start from time 0.

The experimental results for IWLS-4 show that the dynamic method is more flexible while the effectiveness of the static method is highly dependent on the quality of the predetermined test cubes. Nevertheless, the static method on its own still yields significant reduction in both test data volume and TAT compared with with the baseline case of nondynamically compacted ATPG test cubes.

VII. Conclusion

We have presented an SOC testing approach that integrates test data compression, TAM/test wrapper design, and test scheduling. The LFSR reseeding technique from [20] is used as the compression engine. All cores in the SOC share a single on-chip LFSR, i.e., at any clock cycle one or more cores can simultaneously receive data from the LFSR. To reduce the overall TAT for the SOC, it is necessary to increase the throughput of the LFSR (i.e., the number of care bits the LFSR generates per clock cycle), and configure the cores with as many WSCs as possible. These objectives are accomplished using the proposed scheduling algorithm *TWCScheduler* that determines appropriate test wrappers and test schedules for each core.

Experimental results for d695, an SOC crafted from IWLS benchmarks, and an SOC with industrial circuits show that significant reduction in TAT can be achieved. For most cases, an optimal solution can be found such that the TAT of the SOC is the same as that of the most time-consuming core. The scheduling algorithm is also scalable for large industrial circuits. For the larger benchmark SOC, we used in this paper that consists of nine industrial cores, the CPU time ranges from 1 to 30 min for different values of $S_{\max}$. The proposed approach has small hardware overhead and is easy to deploy. Only one LFSR, one phase shifter, and some scheduling and modulo counters need to be added to the SOC.

We have also presented an alternative optimization approach that combines dynamic test compression with the proposed test architecture. Experimental results show that the dynamic-scheduling method is more flexible since the performance of the static method depends on the nature of the predetermined test cubes.

References

[1] S. Hellebrand, H.-G. Liang, and H. J. Wunderlich, "A mixed mode BIST scheme based on reseeding of folding counters," in *Proc. Int. Test Conf.*, 2000, pp. 778–784.

[2] A. A. Al-Yamani and E. J. McCluskey, "Built-in reseeding for serial BIST," in *Proc. IEEE VLSI Test Symp.*, 2003, pp. 63–68.

[3] H.-G. Liang, S. Hellebrand, and H. J. Wunderlich, "Two-dimensional test data compression for scan-based deterministic BIST," in *Proc. Int. Test Conf.*, 2001, pp. 894–902.

[4] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, "Embedded deterministic test," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 5, pp. 776–792, May 2004.

[5] P. Varma and S. Bhatia, "A structured test re-use methodology for core-based system chips," in *Proc. Int. Test Conf.*, 1998, pp. 294–302.

[6] Y. Zorian and E. J. Marinissen, "System chip test: How will it impact your design?" in *Proc. Des. Autom. Conf.*, 2000, pp. 136–141.

[7] E. J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M. Ricchetti, and Y. Zorian, "On IEEE P1500's standard for embedded core test," *J. Electron. Test.: Theory Appl. (JETTA)*, vol. 18, no. 4/5, pp. 365–383, Aug. 2002.

[8] S. K. Goel and E. Marinissen, "Layout-driven SOC test architecture design for test time and wire length minimization," in *Proc. Des., Autom. Test Eur. Conf.*, 2003, pp. 738–743.

[9] E. Larsson and H. Fujiwara, "Power constrained preemptive TAM scheduling," in *Proc. IEEE ETW*, 2002, pp. 119–126.

[10] M. Nourani and J. Chin, "Test scheduling with power-time tradeoff and hot-spot avoidance using MILP," *Proc. Inst. Elect. Eng.—Comput. Digital Tech.*, vol. 151, no. 5, pp. 341–355, Sep. 2004.

[11] D. Zhao and S. Upadhyaya, "Power constrained test scheduling with dynamically varied TAM," in *Proc. IEEE VLSI Test Symp.*, 2003, pp. 273–278.

[12] V. Immaneni and S. Raman, "Direct access test scheme—Design of block and core cells for embedded ASICs," in *Proc. Int. Test Conf.*, 1990, pp. 488–492.

[13] I. Ghosh, S. Dey, and N. Jha, "A fast and low cost testing technique for core-based system-on-chip," in *Proc. Des. Autom. Conf.*, 1998, pp. 542–547.

[14] N. Touba and B. Pouya, "Testing embedded cores using partial isolation rings," in *Proc. IEEE VLSI Test Symp.*, 1997, pp. 10–16.

[15] E. Larsson and Z. Peng, "An integrated framework for the design and optimization of SOC test solutions," *J. Electron. Test.: Theory Appl. (JETTA)*, vol. 18, no. 4/5, pp. 385–400, Aug.–Oct. 2002.

[16] Q. Xu and N. Nicolici, "Time/area tradeoffs in testing hierarchical SOCs with hard mega-cores," in *Proc. Int. Test Conf.*, 2004, pp. 1196–1202.

[17] V. Iyengar, A. Chandra, S. Schweizer, and K. Chakrabarty, "A unified approach for SOC testing using test data compression and TAM optimization," in *Proc. Des., Autom. Test Eur. Conf.*, 2003, pp. 1188–1189.

[18] A. B. Kinsman and N. Nicolici, "Time-multiplexed test data decompression architecture for core-based SOCs with improved utilization of tester channels," in *Proc. Eur. Test Symp.*, 2005, pp. 196–201.

[19] P. T. Gonciari and B. M. Al-Hashimi, "A compression-driven test access mechanism design approach," in *Proc. Eur. Test Symp.*, 2004, pp. 100–105.

[20] E. H. Volkerink and S. Mitra, "Efficient seed utilization for reseeding based compression," in *Proc. IEEE VLSI Test Symp.*, 2003, pp. 232–237.

[21] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, "Test wrapper and test access mechanism co-optimization for system-on-chip," *J. Electron. Test.: Theory Appl. (JETTA)*, vol. 18, no. 2, pp. 213–230, Apr. 2002.

[22] C. V. Krishna and N. A. Touba, "Adjustable width linear combinational scan vector decompression," in *Proc. Int. Conf. Comput.-Aided Des.*, 2003, pp. 863–866.

[23] S. Mitra and K. S. Kim, "XPAND: An efficient test stimulus compression technique," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 163–173, Feb. 2006.

[24] J. Rajski, N. Tamarapalli, and J. Tyszer, "Automated synthesis of large phase shifters for built-in self-test," in *Proc. Int. Test Conf.*, 1998, pp. 1047–1056.

[25] B. Koenemann, "LFSR-coded test patterns for scan design," in *Proc. Eur. Test Conf.*, 1991, pp. 237–242.

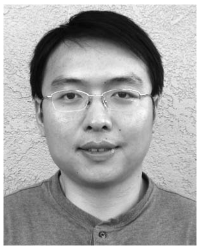
[26] I. Hamzaoglu and J. Patel, "Test set compaction algorithms for combinational circuits," in *Proc. Int. Conf. Comput.-Aided Des.*, 1998, pp. 283–289.

[27] A. Sehgal, V. Iyengar, and K. Chakrabarty, "SOC test planning using virtual test access architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 12, pp. 1263–1276, Dec. 2004.

[28] S. Hellebrand, B. Reeb, S. Tarnick, and H.-J. Wunderlich, "Pattern generation for a deterministic BIST scheme," in *Proc. Int. Conf. Comput.-Aided Des.*, 1995, pp. 88–94.

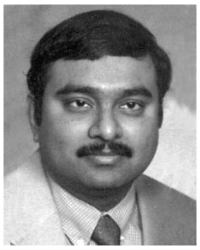
[29] [Online]. Available: http://www.iwls.org/iwls2005/benchmarks.html

[30] H. Vranken, F. Hapke, S. Rogge, D. Chindamo, and E. Volkerink, "ATPG padding and ATE vector repeat per port for reducing test data volume," in *Proc. Int. Test Conf.*, 2003, pp. 1069–1078.



Zhanglei Wang received the B.Eng. degree in computer and electrical engineering from Tsinghua University, Beijing, China, in 2001 and the M.S.E. and Ph.D. degrees in computer and electrical engineering from Duke University, Durham, NC, in 2004 and 2007, respectively.

He is currently a Hardware Engineer with Cisco Systems, Inc., San Jose, CA. His research interests include test compression, test pattern grading, test generation, high-speed test, and system-level test and diagnosis.



Krishnendu Chakrabarty (S'92–M'96–SM'01–F'08) received the B.Tech. degree from the Indian Institute of Technology, Kharagpur, India, in 1990 and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor, in 1992 and 1995, respectively.

He is currently a Professor of electrical and computer engineering with Duke University, Durham, NC. He is also a Chair Professor in software theory with the School of Software, Tsinghua University, Beijing, China. His current research projects include the following: testing and design-for-testability of integrated circuits; digital microfluidics and biochips, circuits and systems based on DNA self-assembly, and wireless sensor networks. He has authored seven books on these topics, published 300 papers in journals and refereed conference proceedings, and given over 120 invited, keynote, and plenary talks.

Dr. Chakrabarty is an Associate Editor of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE TRANSACTIONS ON VLSI SYSTEMS, IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS, and the *Association for Computing Machinery (ACM) Journal on Emerging Technologies in Computing Systems*. He also serves as an Editor of *IEEE Design and Test of Computers* and of the *Journal of Electronic Testing: Theory and Applications (JETTA)*. He is a recipient of the National Science Foundation Early Faculty (CAREER) Award, the Office of Naval Research Young Investigator Award, the Humboldt Research Fellowship from the Alexander von Humboldt Foundation, Germany, and several best papers awards at IEEE conferences. He is a Distinguished Engineer of ACM. He is a 2009 Fellow of the Japan Society for the Promotion of Science. He is recipient of the 2008 Duke University Graduate School Dean's Award for excellence in mentoring. He served as a Distinguished Visitor of the IEEE Computer Society during 2005–2007, and as a Distinguished Lecturer of the IEEE Circuits and Systems Society during 2006–2007. Currently, he serves as an ACM Distinguished Speaker.

Seongmoon Wang received the B.S. degree in electrical engineering from Chungbuk National University, Cheongju, Korea, in 1988, the M.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1991, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1998.

He was a Design Engineer with GoldStar Electron, Korea, and a Discrete Fourier Transform Engineer with Syntest Technologies and 3Dfx Interactive. He is currently a Senior Research Staff Member with NEC Laboratories America, Inc., Princeton, NJ. His main research interests include design for testability, computer-aided design, and self-repair/diagnosis techniques of very large scale integration.