SoC Testing Using LFSR Reseeding, and Scan-Slice-Based TAM Optimization and Test Scheduling*

Zhanglei Wang†, Krishnendu Chakrabarty† and Seongmoon Wang‡
†ECE Dept., Duke University, Durham, NC
E-mail: {zw8,krish}@ee.duke.edu
‡NEC Laboratories America, Princeton, NJ
E-mail: swang@nec-labs.com

***Abstract*— We present an SoC testing approach that integrates test data compression, TAM/test wrapper design, and test scheduling. An improved 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 from the test cubes for multiple cores. We also propose a scan-slice-based scheduling algorithm that tries to maximize the number of care bits the LFSR can produce at each clock cycle, such that the overall test application time is minimized. Experimental results for both ISCAS circuits and industrial circuits show that optimal test application time, which is determined by the largest core, can be achieved. The proposed approach has small hardware overhead and is easy to deploy. Only one LFSR, one phase shifter, and a few counters should be added to the SoC. The scheduling algorithm is also scalable for large industrial circuits. The CPU time for a large industrial design ranges from 1 to 30 minutes.**

I. Introduction

Recent growth in design complexity and the integration of embedded cores in systems-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. Some state-of-the-art compression methods such as [1] 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 that are not provided to the system integrator with detailed structural information. 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 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. To tackle this problem, test access mechanism (TAM) and test wrappers have been proposed as key components of an SoC test architecture [2]. TAMs deliver pre-computed 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 ATE, testing time, and thereby affect test cost. Many techniques have been proposed for TAM/wrapper design. However, these techniques either do not consider test data compression, or they utilize relatively inefficient compression techniques [3].

In [4], test patterns for each core in an SoC are compressed separately using 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 [4] 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 [5], 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 [5] how the estimation function can be derived, 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 [6] as the compression engine because of its high encoding efficiency. In this paper, we assume that the SoC is comprised of hard cores and cores whose structural information is not available. 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 proper TAM design and test scheduling, more cores can be tested in parallel, and the test application time 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.

Section II describes the proposed SoC testing approach. The associated scheduling algorithm is presented in detail in Section III. Section IV reports experimental results and Section V concludes the paper.

II. Proposed Approach

An improved LFSR reseeding technique is proposed in [6]. It allows the generation of a single scan slice from multiple

*The work of Z. Wang and K. Chakrabarty was supported in part by the National Science Foundation under grant No. 0204077.

978-3-9810801-2-4/DATE07 © 2007 EDAA

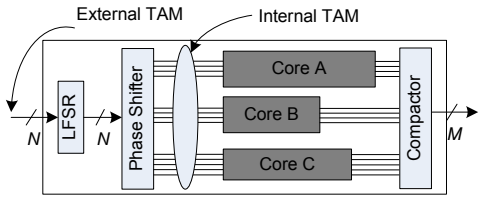

Fig. 1. Test architecture.

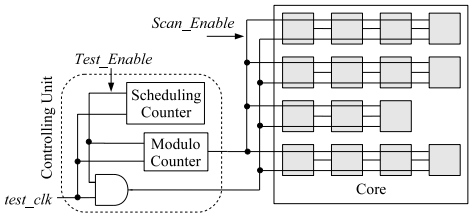

Fig. 2. Test control scheme.

seeds, or multiple scan slices from a single seed. An additional tester channel is needed to control when reseeding occurs. In this work, we choose to use the compression technique of [6] because of its high encoding efficiency. The test data volume achieved by [6] can be estimated as $CB/En + ctrl$, where CB is the number of care bits, En is the Encoding Efficiency, and $ctrl$ is the volume of the controlling data. Without loss of generality, we consider the test data volume that is obtained with $En = 90\%$ and $ctrl$ (in bits) equal to TAT (in clock cycles). For large industrial circuits, the value of En is considerably higher. Hence the estimated test data volume is a pessimistic over-estimate.

The architecture of the proposed approach is shown in Fig. 1. 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. 2. 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 will only be assigned one clock range, hence the storage for the scheduling data is very small. For handling test responses, any compaction scheme can be used.

Each core is associated with a modulo 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. 2. Section III provides more details.

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 [6] is applied for the equivalent

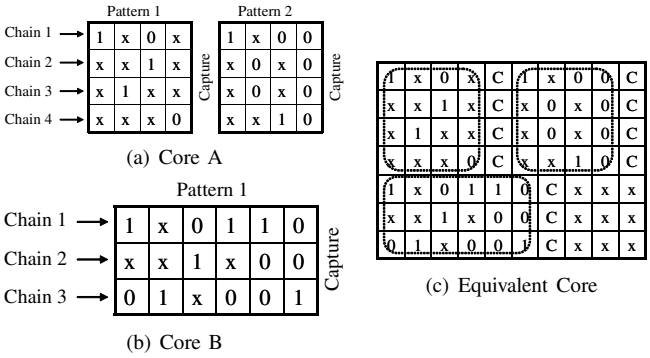

Fig. 3. Two cores and their equivalent core.

core. TAT is significantly reduced because: (1) Multiple cores are tested in parallel, and (2) When some cores are in the capture or inactive mode, other cores are in the shift mode and receiving data from the LFSR.

Fig. 3 shows two cores A and B and their equivalent core. In Fig. 3, each row represents a wrapper scan chain (WSC) and each column represents a scan slice. Core A has 4 WSCs and two patterns with each pattern having 4 scan slices. Core B has 3 WSCs and one pattern that has 6 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 7 WSCs and 9 scan slices.

As shown in Fig. 1, the number of internal TAM lines is no longer restricted by the number of scan IO pins of the SoC, which are used as scan chain inputs/outputs. Compared with existing test scheduling techniques [7], 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 [8]. Therefore, in this work 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.

The LFSR reseeding technique of [6] 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 [9]). In this work, we assume that S_{max} is a user-defined parameter. The proposed TAM, test wrapper, and test data compression co-optimization problem is referred to as $\mathcal{P}_{TWC}$ (TWC stands for TAM, Wrapper, and Compression), and can be formally stated as follows:

$\mathcal{P}_{TWC}$: 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 test application time is minimized.

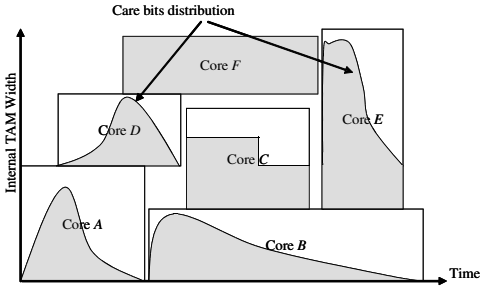

Fig. 4. Slice-based scheduling.

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, especially 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.

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 [7]. *TWCScheduler*, as shown in Fig. 4, works on a per-slice basis. In Fig. 4, 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 test application time. 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}. 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 idential heights, such that the core is tested during more than one clock range. This splitting action is referred to as *preemption*.

III. Scheduling Algorithm

It was shown in [7] that, for a given core, the test application time 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 2-tuple (W_j, $T_i(W_j)$), where W_j is the TAM width supplied to the wrapper and $T_i(W_j)$ is the test application time 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 $W_k \leq W_j$ and $T_i(W_k) \leq T_i(W_j)$, 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 [7] to compute Pareto-optimal TAM widths for a given core.

For the rest of the paper, we use $W_{i,k}$ to denote the k-th 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 test application time of Core i with TAM width $W_{i,k}$ is $T_i(W_{i,k})$. All Pareto-optimal TAM widths of Core i are sorted in an ascending order such that $\forall(k,l)$, $1 \leq k, l \leq N_i$, $l > k \Rightarrow W_{i,l} > W_{i,k}$.

Given a core, let s_i (s_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 [7]:

$$T = (1 + \max\{s_i, s_o\}) \cdot p + \min\{s_i, s_o\} \quad (1)$$

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) 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. The modulo counter mentioned in Section II is a modulo-($\max\{s_i, s_o\} + 1$) counter and drives the *Scan_Enable* signal, which controls scan operations of all scan cells in the core. The output of the modulo counter is reset to 0 in each capture cycle, incremented by 1 in each shift cycle, and again reset to 0 in the next capture cycle.

A. Algorithm overview

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 t-th slice of the equivalent core.

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 observation 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.

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;

The high level flow of *TWCScheduler* is shown in Procedure 1. 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 $\mathcal{P}_{TWC}$ is found. *TWCScheduler* always assigns to the

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*.

maxCore its maximum Pareto-optimal TAM width, such that an optimal solution is achievable. Section IV will show that for most cases an optimal solution can be found.

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. Meanwhile, 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 non-bottleneck cores. The motivation for preemption is two-fold. (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.

In summary, *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. Highly specified patterns of bottleneck cores are first scheduled, followed by *maxCore*. The patterns for *maxCore* and all bottleneck cores are sorted in a descending order in favor of "abort-at-first-fail" strategies. The remaining cores are scheduled one by one in a random order, using a greedy search strategy that will be discussed later.

B. Data structures and Supporting procedures

Table I summerizes 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 attemps 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 ≤ S_max then
 7:         currTime ++; currSlice ++;
 8:     else
 9:         currSlice = 0; startTime ++;
10:         if startTime + TAT(i) ≥ 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;
```

C. Procedure TWCScheduler

Procedure *TWCScheduler* is shown in Procedure 3. Lines 1-2 are initialization operations and have been discussed earlier in Section III-A. 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 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 heavilily specified patterns have been applied (Line 15). If a Core i is a non-bottleneck 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 test application time 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 test application time, which implies that the core has been assigned a start cycle of 0. 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 test application times.

D. Optimize trySchedule

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 mid-sized core (e.g., cores listed in Table V in Section IV). 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

Procedure 3 *TWCScheduler*(**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, ∞, *DESC*);
13: **for all** Core i in $|\mathbf{C}|$, $i \neq j$ **do**
14: **if** *begun*(i) $== 1$ **then**
15: *trySchedule*(i, *EndTime*(i), ∞, *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

conflicts occur, *checkStart* returns 0 and *startTime* is directly incremented by 1, 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.

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

Procedure *checkStart* (shown in Procedure 4) uses three caches for quick identification of conflicts. Cache A stores all scan slices of Core i that have at least δ care bits. Cache B stores all elements of *timeLine* that have at least $S_{max} - 3$ care bits. Cache C stores all elements of *timeLine* that have at least $S_{max} - \delta$ care bits. These numbers are chosen through extensive experiments. These caches are updated when *timeLine* is updated in Procedure *doSchedule*, and when Core i is assigned a new number of internal TAM lines in Procedure *designWrapper*. 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 B contains fewer elements and is first checked.

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

IV. Experimental Results

First, we run *TWCScheduler* on the d695 benchmark SoC [7]. Test patterns for the cores are compacted by Mintest. Table III lists detailed information about d695. We assume

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

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 test application time. 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 test application time of the SoC is the same as the end cycle of s13207 (in bold). The CPU time is less than 1 second.

Next, we present results for an SoC named NIM that consists of 9 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 0.

Table VI reports scheduling results for NIM with $S_{max} =$

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

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			

$16, 32, 48, 64$ and $\delta = 10$. Table VI is similar to Table IV. Row "CPU time" lists the execution time in minutes and seconds. As can be seen 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 bits, if the LFSR size is 1044 ($kS_{max}+20, k = 16$, see Section II) 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 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 an especially 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 our work to some related prior work. To compare with [4], we only considered the five cores for d695 that were used in [4]. 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 [4] is 11,658 clock cycles (for the "seed-only" variant) and 9,612 clock cycles (for the "seed-mux" variant) for Mintest-compacted test patterns. The number of ATE channels is not reported in [4]. The estimated test data volume for the proposed method is 190,250 bits (161,281 care bits, E_n=90%, $ctrl$=11,049). The exact test data volume is 181,821 bits (the LFSR size is 532 stages and there are 34 ATE channels). The test data volume reported in [4] is 419,688 bits (seed-only) and 442,152 bits (seed-mux). The TAT reported in Figure 5 of [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 [10], which do not use compression. The best TAT reported in [10] for d695 with a TAM width of 64 bits is 9,869 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 work applies 1120 test patterns to the cores, while the TAT in [10] is obtained for only 881 patterns. More test patterns are expected to result in higher test quality.

V. CONCLUSIONS

We have presented an SoC testing approach that integrates test data compression, TAM/test wrapper design, and test scheduling. The LFSR reseeding technique from [6] 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 test application time 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 wrapper scan chains 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 both d695 and an SoC with industrial circuits show that significant reduction in test application time 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 the paper that consists of 9 industrial cores, the CPU time ranges from 1 to 30 minutes for different values of S_{max}. The proposed approach has small hardware overhead and is easy to deploy.

REFERENCES

[1] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, "Embedded deterministic test," *IEEE Trans. CAD*, vol. 23, pp. 776–792, May 2004.

[2] E. J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M. Ricchetti, and Y. Zorian, "On IEEE P1500's standard for embedded core test," *JETTA*, vol. 18, pp. 365–383, Aug. 2002.

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

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

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

[6] E. H. Volkerink and S. Mitra, "Efficient seed utilization for reseeding based compression," in *Proc. VTS*, 2003, pp. 232–237.

[7] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, "Test wrapper and test access mechanism Co-Optimization for System-on-Chip," *JETTA*, vol. 18, pp. 213–230, 2002.

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

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

[10] A. Sehgal, V. Iyengar, and K. Chakrabarty, "SOC test planning using virtual test access architectures," *IEEE Trans. VLSI Systems*, vol. 12, pp. 1263–1276, dec. 2004.