SOC Test-Architecture Optimization for the Testing of Embedded Cores and Signal-Integrity Faults on Core-External Interconnects

QIANG XU and YUBIN ZHANG
The Chinese University of Hong Kong
and
KRISHNENDU CHAKRABARTY
Duke University

The test time for core-external interconnect shorts and opens is typically much less than that for core-internal logic. Therefore, prior work on test-infrastructure design for core-based system-on-a-chip (SOC) has mainly focused on minimizing the test time for core-internal logic. However, as feature sizes shrink for newer process technologies, the test time for signal integrity (SI) faults on interconnects cannot be neglected. The test time for SI faults can be comparable to, or even larger than, the test time for the embedded cores. We investigate the impact of interconnect SI tests on SOC test-architecture design and optimization. A compaction method for SI faults and algorithms for test-architecture optimization are also presented. Experimental results for the ITC'02 benchmarks show that the proposed approach can significantly reduce the overall testing time for core-internal logic and core-external interconnects.

Categories and Subject Descriptors: B.7.3 [**Integrated Circuits**]: Reliability and Testing—*Testability*

General Terms: Reliability, Design, Algorithms

Additional Key Words and Phrases: Core-based system-on-chip, interconnect testing, test-access mechanism (TAM), test scheduling

A preliminary version of this article was published in *Proceedings of the 2007 IEEE/ACM Design Automation Conference (DAC)*, 676–681.
Q. Xu is also affiliated with CAS-CUHK Shenzhen Institute of Advanced Integration Technology.
This work was supported in part by the Hong Kong SAR RGC Earmarked Research Grant 417406 (to Q. Xu, with K. Chakrabarty as a named collaborator) and 417807 and in part by Hi-Tech Research and development Program of China (863) under grant No. 2007AA01Z109.
Authors' addresses: Q. Xu and Y. Zhang, Department of Computer Science & Engineering, The Chinese University of Hong Kong, N.T., Hong Kong; K. Chakrabarty, Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708.

ACM Reference Format:
Xu, Q., Zhang, Y., and Chakrabarty, K., 2009. SOC test-architecture optimization for the testing of embedded cores and signal-integrity faults on core-external interconnects, ACM Trans. Des. Autom. Electron. Syst. 14, 1, Article 4 (January 2009), 27 pages, DOI = 10.1145/1455229.1455233 http://doi.acm.org/10.1145/1455229.1455233

1. INTRODUCTION

As feature sizes shrink and clock frequencies increase for high-performance system-on-a-chip (SOC) designs, signal integrity (SI), that is, the ability of an input signal to generate correct responses in a circuit [Guler and Kilic 1999], is becoming a major concern for the interconnects between embedded cores [Kao et al. 2001]. SI problems, typically caused by capacitance and inductance between interconnects, include overshoots, undershoots, glitches, oscillations, excessive signal delay, and even signal speedup [Kundu et al. 2005]; see Figure 1. SI-related problems are aggravated in core-based SOC designs because interconnects transporting signals between embedded cores tend to be long, hence they suffer more from crosstalk effects [Nordholz et al. 1998]. If the noise-induced voltage swing and timing skews depart from the noise-immune region, functional error may occur.

Traditionally, SI problems have been treated as design errors, and a number of physical design and fabrication solutions [Becer et al. 2004; Chen et al. 2004; Massoud et al. 2002; Zhang and Sapatnekar 2004] have been proposed in the literature to tackle them. These design techniques rely on accurate simulation of SI effects, which are affected by many parameters (e.g., characteristics of interconnects and transistors, input data and environmental noise). Unfortunately, these parameters are interdependent and our lack of complete knowledge of this interdependence leads to uncertainty and inaccuracies in the simulation of SI loss [Wang et al. 2007]. Moreover, process variations and manufacturing defects may aggravate the SI-related problems [Natarajan et al. 1998]. Since it is unacceptable to over-design the circuit to tolerate signal integrity loss in all cases and it is impossible to predict the occurrence of defects, manufacturing test strategies are essential for detecting SI-related errors [Cuviello et al. 1999; Sirisaengtaksin and Gupta 2002; Tehranipour et al. 2003].

Various SI fault models [Cuviello et al. 1999; Kundu et al. 2005; Tehranipour et al. 2004] and associated test methodologies [Bai et al. 2000; Tehranipour et al. 2003] have been proposed in the literature. SI-related problems are aggravated in core-based SOC designs because interconnects carrying signals between embedded cores tend to be long and hence they suffer more from parasitic effects [Nordholz et al. 1998]. Despite this problem, most prior work in SOC test-architecture optimization has focused on core-internal test (InTest) only [Ebadi and Ivanov 2003; Goel and Marinissen 2002; Iyengar et al. 2002; Larsson and Peng 2002; Larsson and Fujiwara 2003; Nahvi and Ivanov 2004; Xu and Nicolici 2004; Zhao and Upadhyaya 2005; Zou et al. 2003] and neglected the problem posed by core-external interconnect SI faults. The test time for SI faults is long because of the need to exercise a large number of signal-state combinations for the interconnects [Sirisaengtaksin and Gupta 2002; Tehranipour

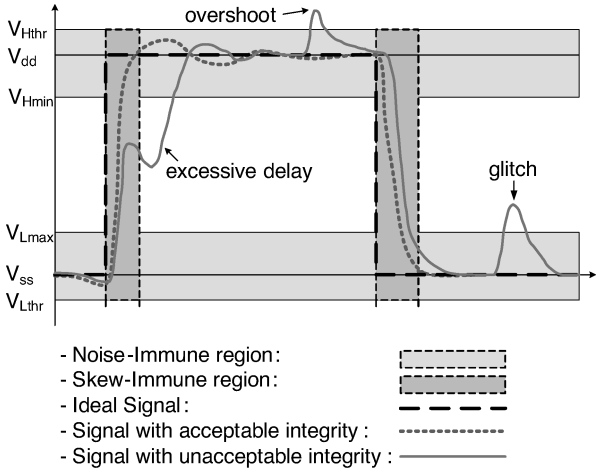

Fig. 1. Illustration of signal-integrity loss.

et al. 2003]. For nanometer SOCs running at speeds of several hundred MHz and higher, the test time for SI faults can be as high as or even exceed the test time for the embedded cores. Therefore, the goal of this article is to study, for the first time, the impact of SI faults on SOC test-architecture design. The main contributions of this article are as follows:

—We present a two-dimensional SI test pattern compaction strategy to reduce the interconnect SI test data volume.

—We develop algorithms for SOC test-architecture optimization to minimize the overall SOC testing time for both interconnect SI faults and core-internal faults.

—We show that for the ITC 2002 SOC Test benchmarks, the test time obtained using the proposed method is significantly less than that using two baseline methods: (i) a test access architecture optimized for core-internal test and then used for core-external SI fault testing; (ii) a test-access architecture optimized for both core-internal test and core-external test, but where only pattern count reduction is employed for the tests for SI faults.

The remainder of this article is organized as follows. Section 2 reviews related prior work and provides motivation for the work described in this paper. Section 3 presents the proposed test pattern compaction method for SI faults. In Section 4, SOC test-architecture optimization techniques for handling both core-internal faults and core-external SI faults are described. Experimental results for benchmark SOCs [Marinissen et al. 2002] are presented in Section 5. Finally Section 6 concludes this article.

2. RELATED WORK AND MOTIVATION

Early attempts for testing SI-related problems modeled crosstalk at the circuit level [Attarha and Nourani 2002; Chen et al. 1999]. Although more accurate than gate-level models, the complexity of the associated test-pattern generation

procedures limits its usefulness for SOC interconnects. Cuviello et al. [1999] proposed a behavioral-level SI fault model, called the *maximal aggressor (MA)* model. This approach assumes that all aggressors[1] make the same simultaneous transition (in the same direction) and act collectively to generate a glitch when the victim is quiescent, or a delay error when the victim makes an opposite transition. Therefore, $6N$ test-vector pairs are needed to detect SI faults for a set of N interconnects. Since in reality, inter-core interconnects in SOC can be of any arbitrary topology, to reduce test pattern count, Sirisaengtaksin and Gupta [2002] extended the MA fault model to the so-called maximum-affecting-line (MAL) fault model by taking the physical layout information into account. If all the physical defects are capacitive or resistive, all MA/MAL faults can be targeted using a pattern count that is linear in the number of interconnects. When inductance is considered, however, such test patterns may not be able to generate maximum noise/delay on the victim line [Chen et al. 1999; Naffziger 1999]; hence, Tehranipour et al. [2004] presented a *multiple transition (MT)* fault model that covers all transitions on victim and multiple transitions on aggressors. The number of test patterns for this MT fault model, however, is exponential in the number of interconnects under test. To address this problem, an empirically-determined locality factor k showing how far the effect of aggressors remains significant, was introduced. For a set of N interconnects, the number of test patterns for the reduced-MT fault model is approximately $N \cdot 2^{2k+2}$.

Built-In Self-Test (BIST) has been a popular test method used to detect SI-related errors [Sekar and Dey 2002; Tehranipour et al. 2004]. In this approach, driver side of interconnects are equipped with test generators to generate transitions on the aggressors and victims, while at the receiver side, various types of integrity-loss sensor (ILS) cells are embedded to detect SI-related errors. Bai et al. [2000] introduced on-chip test generators and error detectors at the core boundaries, based on the MA fault model [Cuviello et al. 1999]. Nourani and Attarha [2001] presented two ILS cell designs to detect voltage distortions and timing violations, respectively. Later, several other ILS designs [Caignet et al. 2001; Tabatabaei and Ivanov 2002] were introduced, which are more accurate in measuring voltage and/or timing violations, at the cost of large area overheads. Assuming the existence of logic BIST structures in an SOC, Sekhar and Dey [2002] presented a self-test solution, called LI-BIST, for both the core internal logic and the SOC interconnects. Zhao et al. presented an online testing technique to capture noise-induced logic failures in functional buses [Zhao et al. 2004]. Yang et al. [2001] used boundary scan and IDDT to test functional buses. Finally, Chen et al. [2001] discussed how to test SI defects on the data bus and the address bus by executing a test program on the microprocessor.

A test method that relies on hardware-based test generators may cause over-testing and/or under-testing since not all test patterns generated in the test mode are valid in the normal functional mode of the SOC. In addition, since the SOC interconnect topology can be arbitrary (see Figure 2) and it is hard

[1]An interconnect on which the error effect takes place is defined as the *victim*, while the affecting interconnects are referred to as its *aggressors*.

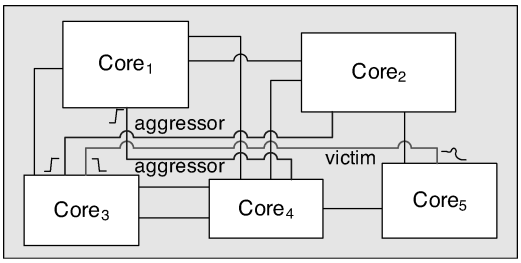

Fig. 2. An arbitrary interconnect topology in an SOC, with a victim and the corresponding aggressors.

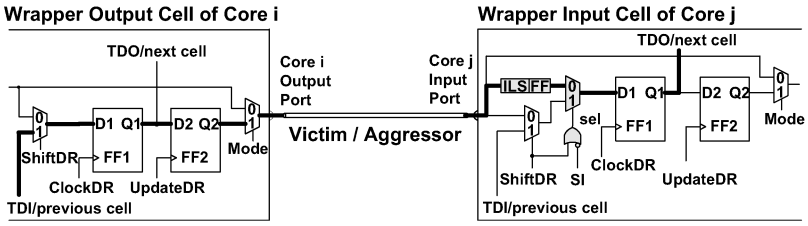

Fig. 3. Wrapper cell design for SI test (redrawn from Tehranipour et al. [2003]).

to predict it during test-hardware insertion, the interconnects between several cores may be close enough to result in SI errors [Sirisaengtaksin and Gupta 2002]. It is very difficult, if not impossible, to take interconnect proximity into account for these hardware-based test techniques. Therefore, in this work, we assume that the test stimuli are loaded from an external tester to the core-test wrapper. To apply SI test at the core-level, as shown in Figure 3 [Tehranipour et al. 2003], the wrapper-output cell (WOC) should be able to provide the necessary consecutive transitions to interconnects; the wrapper-input cell (WIC) needs to be equipped with a signal integrity loss sensor [Bai et al. 2000; Tehranipour et al. 2003] to capture the signal with noise and/or delay error.

Most prior work in SOC test-architecture optimization [Xu and Nicolici 2005] only takes core internal testing into account, which is mainly because testing interconnect shorts/opens requires little time and therefore core-external (ExTest) testing can be ignored in the test-architecture optimization process. However, when high SI fault coverage is desired for today's SOCs, the testing time for SOC interconnects can be comparable to or even higher than the testing time for the core-internal logic. To understand this issue, let us estimate the interconnect SI testing time for a representative video-processing SOC [Dutta et al. 2001; Goel et al. 2004], which contains two 32-bit programmable interconnect (PI) buses, each connecting to a number of embedded cores (e.g., MIPS/TriMedia processor, mpeg-2 decoder, transport stream processor, and IEEE 1394 controller). Without loss of generality, suppose ten cores connect to each PI buses and assume that each core on average sends data to two other cores on the bus. Hence the number of victim interconnects under test on each PI bus is $N = 2 \times 10 \times 32 = 640$. Based on the previous discussion, without test set compaction, $6N \times 2 = 7680$ test vector pairs are needed for the MA fault model;

Table I. Format of the SI Test Patterns

	$core_1$ WOC						...	$core_i$ WOC					...	$core_n$ WOC				**Bus**			
p_0	...	x	↑	x	x	↓	...	x	1	↑	...	x	...	x	x	x	...	x	x	1	...
p_1	...	↑	x	↓	x	x	...	x	x	x	...	↑	...	x	x	↑	...	x	x	1	...
...							...						...								
p_x	...	x	x	x	x	x	...	0	x	↓	...	x	...	↓	x	x	...	1	x	x	...

while roughly $N \cdot 2^{2k+2} \times 2 = 327680$ test vector pairs are needed for the reduced-MT fault model with the locality factor $k = 3$. Since the total numbers of all the core I/Os for a typical SOC is in the range of several thousand, the test time for MA faults is in the range of millions of clock cycles for serial ExTest, while the test time for reduced-MT faults is two orders of magnitude higher. On the other hand, as reported in Goel et al. [2004], the SOC test time for core-internal logic is less than two million clock cycles when the total number of test access mechanism (TAM) wires is 140, which in turn is less than the testing time for the previous SI faults. Moreover, with shrinking feature sizes of deep-submicron technology, short interconnects may also suffer from SI problems [Nordholz et al. 1998]. Therefore, it is likely that we need to test for SI faults on hundreds or even thousands of interconnects in the SOC. Prohibitively high test time is needed if an effective test-pattern compaction scheme is not employed and the SOC test-architecture is not optimized for both core-internal logic test and interconnect SI test.

Three important conclusions can be drawn from this discussion:

—Effective test set compaction strategy should be utilized to reduce the volume of test data for interconnect SI faults;

—Parallel external testing is required in order to reduce the test time for interconnect SI faults;

—The SOC test-architecture needs to minimize the overall testing time for both core-internal logic and core-external interconnects.

These observations motivate the work presented in this article.

3. TWO-DIMENSIONAL SI TEST-SET COMPACTION

We assume that the test stimuli for SI faults are given to us a priori; these stimuli can take the form of functional patterns, pseudorandom patterns, and/or patterns generated for various SI fault models [Cuviello et al. 1999; Sirisaengtaksin and Gupta 2002; Tehranipour et al. 2004]. Since a victim interconnect is mainly affected by its neighboring aggressors [Kundu et al. 2005], the signal integrity test patterns typically feature a large number of don't-care bits. The format of the SI test vector pairs applied at the wrapper output cells of the embedded cores is shown in Table I. The entry 'x' represents a don't-care bit; '0/1' indicates that the corresponding core output terminal stays at 0/1 in consecutive cycles, and ↑ (↓) represents a positive (negative) transition. For each test pattern, we also add a postfix to denote whether this test pattern utilizes a shared bus line (as discussed in the following paragraph)—a '1' indicates that the specific bus line is utilized while 'x' implies that it is a "don't-care".

Algorithm 1 - TestCompaction

INPUT: P_o
OUTPUT: P_c

```
1. initialize P_c = ∅; P_u = P_o;        /* P_u = {P_u[1], P_u[2], ...} */
2. while (|P_u| > 0) {
3.     set p_c = P_u[1] ; P_m = {P_u[1]};
4.     for (i = 2 to |P_u|) {
5.         if(p_c and P_u[i] are compatible) {
6.             merge p_c and P_u[i] to p_c;
7.             P_m = P_m ∪ {P_u[i]}; }
.      }
8.     P_u = P_u \ P_m; P_c = P_c ∪ {p_c};
. }
9.  return P_c.
```

Fig. 4. Procedure for SI test pattern count reduction.

Test-pattern-count reduction. Because of the large number of don't-care bits in each test pattern, it is possible to reduce the volume of test data by compacting multiple test vectors into one vector when they are compatible (i.e., their intersection is nonempty). Note that since bus lines are shared by the cores and they may connect many cores at the same time, several SI test patterns may trigger the same bus line from different core boundaries; these patterns cannot be compacted into one test pattern. The postfix that we add to each SI test pattern is used to identify such situations. If the bit values for a specific position in the postfix of two SI test patterns are both '1', they are marked as incompatible (e.g., p_0 and p_1 in Table I). The problem of finding a compacted test set of minimum size for a given test set is very similar to the traditional test compaction problem and can be formulated as a maximal clique-partitioning problem [Jha and Gupta 2003]. The pattern compaction problem is mapped to a graph, where each vertex corresponds to a test pattern and an edge is added between two vertices if the corresponding test patterns are mutually compatible. A set of compatible SI test patterns form a clique in this graph; our objective to find a minimum number of disjoint cliques that cover all the vertices in the graph. The clique partitioning problem, however, is known to be NP-complete [Garey and Johnson 1979], and approximation algorithms, that is, those with bounded approximation error, suffer from high computational complexity [Arora 1998].

To reduce computation time, we use a simple greedy heuristic as shown in Figure 4. The algorithm takes the original test set P_o as input. A compacted test pattern is generated in each inner loop (Lines 4–7) by merging the first pattern p_1 in the uncompacted test set P_u with the compatible patterns that follow in one pass. The algorithm terminates when all test patterns are compacted, and it outputs the compacted test set P_c. Obviously this greedy strategy is not optimal, and the quality of the resulting P_c depends on the order of the test patterns. To address this problem, we randomize the order of test patterns several times, apply our greedy heuristic, and select the best result, that is, the ordering that leads to the smallest compacted set P_c. Suppose the number of

original test patterns is n and the test pattern width is m. In the worst case, no test pattern is compatible with any other pattern. The time complexity of the above heuristic is $O(mn^2)$.

The preceeding compaction scheme to reduce test pattern count can be viewed as reducing the volume of the test data in a vertical manner.

Test-pattern-length reduction. If we compact all the test patterns together, the length of every compacted pattern will be very large—it will be equal to the sum of the number of WOCs for the different cores. Since each SI test pattern involves only a few cores' terminals (referred to as *care cores* of the SI test pattern), we can bypass the boundaries of the remaining *don't-care cores* (e.g., Core 1 for p_x in Table I) and reduce the length of this test pattern. The above strategy can be viewed as compacting the test pattern in a horizonal manner.

That is, instead of compacting all the test patterns together, we first partition the set of cores into several smaller groups of cores (say, N_g groups). Next we classify the SI test patterns in such way that the test patterns, whose care cores are all within the same core group, form an SI test group. The length of each test pattern is now reduced to the sum of the number of WOCs of this core group, instead of the WOCs of all cores. Let woc_i denote the number of WOCs for core group i. For the remaining test patterns whose care cores fall into multiple core groups, we simply group them as a whole and their length remains the sum of the lengths of the WOCs for all the cores, denoted as woc_{SOC}. The test data volume V_c after two-dimensional compaction is as follows:

$$V_c = \left(\sum_{i=1}^{N_g} p_i \times woc_i + p_r \times woc_{SOC} \right) \times 2, \tag{1}$$

where p_i and p_r represent the number of compacted test patterns in SI test group i and the number of compacted remaining test patterns, respectively. The value '2' in this equation is added because each SI test pattern contains two vectors.

To achieve better compression, we should minimize the number of remaining patterns, and at the same time, balance the test-pattern lengths for the partitions. This problem can be formulated as a hypergraph partitioning problem, with each vertex in the hypergraph corresponding to a core. The weight of each vertex is the number of WOCs of the core corresponding to the vertex and it is used to balance the partitions. A hyperedge is added for each test pattern that connects all its care cores (vertices). Since there might be multiple test patterns having the same care cores, we use the weight of each hyperedge to represent this information. The hypergraph partitioning problem has been well-researched in the literature and we use the *hMetis* package [Selvakkumaran and Karypis 2003] to solve this problem. As shown in Figure 5, for the horizontal SI test pattern compaction of a hypothetical SOC containing seven cores, the patterns corresponding to the cut hyperedge 7-4-6 need to load the WOCs for all the cores, while the other patterns can be applied with shorter pattern lengths. For simplicity, the vertex and edge weights are not shown in the figure.

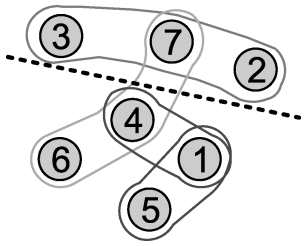

Fig. 5. Hypergraph partitioning for SI test pattern length reduction.

The main objective of the *hMetis* package is to minimize the number of hyperedges that are cut (i.e., the remaining patterns in our problem), while our objective is to achieve the minimum test data volume as shown in Equation (1). To obtain better results, we use the partitioning result obtained from *hMetis* as an initial solution, and on top of it employ the FM partitioning algorithm [Fiduccia and Mattheyses 1982] with the cost function shown in Equation (1) to refine the original solution. That is, we try to move one core at a time between partitions, and check whether the test data volume is reduced. If this is indeed the case, we fix this movement; otherwise, we try an alternative movement. Our experiments show that the above refinement step is able to further reduce test data volume by $3 \sim 5$ percent when compared to the solution obtained from *hMetis*.

4. TEST-ACCESS ARCHITECTURE DESIGN AND OPTIMIZATION

We consider, as a starting point, that every core in the SOC uses wrapper cells as shown in Figure 3 [Tehranipour et al. 2003]. These wrappers are compatible with the IEEE 1500 standard [IEEE Std. 1500 2004] with some additional hardware added to the wrappers for signal integrity test, including a new wrapper instruction to enter the signal-integrity test (*SITest*) mode. In addition, the user-defined logic (e.g., the glue logic between embedded cores) is also treated as a wrapped core. In other words, the SOC is assumed to contain only wrapped logic blocks and interconnect wires that are affected by signal integrity faults.

In addition, we use the TestRail TAM architecture in this work [Marinissen et al. 1998]. While it is possible to use the alternative Test Bus architecture [Varma and Bhatia 1998] to support parallel external testing [Xu and Nicolici 2003], the TestRail architecture is more amenable for core-external testing [Goel and Marinissen 2002].

4.1 SI Test-Architecture Optimization: Problem Formulation

As discussed in Section 2, the testing time for interconnect SI faults can be comparable to or even higher than the testing time for core-internal logic. Therefore, it is necessary for system integrators to optimize the SOC test-architecture for both kinds of tests in order to reduce the overall testing time. The optimization problem addressed in this section can be formulated as follows:

Problem P_{SI_opt}: Given the maximum TAM width W_{max} for the SOC, and

—the test set parameters for each embedded core, including the number of input and output terminals, the number of test patterns for core internal logic, the number of scan chains and the length of each scan chain;
—the test set parameters for each group of compacted interconnect SI tests obtained using the method proposed in Section 3, including the set of cores involved and the number of SI test patterns;

Determine the wrapper design for each core, the TAM resources assigned to each core and a test schedule for the entire SOC such that: (i) the sum of the TAM width used at any time does not exceed W_{max}; (ii) the total SOC testing time T_{SOC} is minimized.

One of the subproblems of P_{SI_opt} is to design and optimize the test wrapper for each core. Since the test application time of a core is dependent on the length of the maximum wrapper scan chain,[2] the main objective in wrapper design and optimization is to build balanced wrapper scan chains. This is a well-researched problem [Marinissen et al. 2000; Iyengar et al. 2002], and we use the *Combine* procedure from [Marinissen et al. 2000] for solving it in *InTest* mode. For a core wrapper in SI test mode, wrapper scan chains contains wrapper cells only and we can therefore assume that balanced wrapper input/output scan chains are achieved. Based on the TestRail architecture, we propose to solve Problem P_{SI_opt} in two steps. First, we describe how to schedule SI tests for a given TAM design, as shown in Section 4.2. Next, we describe our solution for the general problem of how to design and optimize the SOC test-architecture from scratch by adapting an existing method [Goel and Marinissen 2002]; this approach is presented in Section 4.3.

4.2 SI Test Scheduling for a Given TAM Design

Wrapper cells are used for both core-external interconnect SI test and core-internal logic test at the same time. Hence, to avoid test-resource conflicts, we schedule the two types of tests at different times. Therefore, $T_{SOC} = T^{in}_{SOC} + T^{si}_{SOC}$, where T^{in}_{SOC} and T^{si}_{SOC} denote the test time for the core-internal logic and the test time of the core-external interconnects, respectively.

The need for combining interconnect SI test with core-internal test makes test-architecture optimization more difficult compared to the case when only core-internal test is considered. This difficulty results from the fact that interconnect SI test patterns may involve multiple TAMs at the same time, since victim and aggressors in a crosstalk environment may link cores connected to different TAMs. To highlight this problem, we examine how T_{SOC} can be calculated for a given TAM design.

Consider the hypothetical SOC shown in Figure 2. Suppose that after two-dimensional test compaction, the SI test has been placed in three groups, where the SI_1 group involves all the five embedded cores (these are the remaining test patterns after partitioning), SI_2 group involves $Core_1$, $Core_4$ and $Core_5$, and SI_3 group involves $Core_2$ and $Core_3$. Two possible TAM designs and their

[2]Wrapper scan chains are constructed by concatenating core-internal scan chains and WBR cells for InTest.

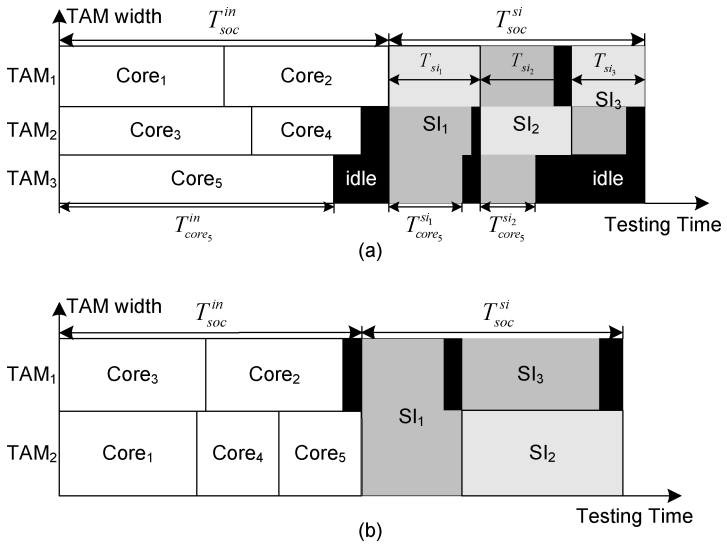

Fig. 6. Example TAM designs and their corresponding test schedules.

corresponding test schedules are shown in Figure 6. Let $T^{in}_{core_i}$ denote the core-internal test time for of Core i, and let $T^{si_j}_{core_i}$ denote the interconnect test time for SI test group j contributed by core i. For the schedule shown in Figure 6(a),

$$\begin{aligned}
T^{in}_{SOC} &= T^{in}_{tam_1} = T^{in}_{core_1} + T^{in}_{core_2} \\
T^{si}_{SOC} &= T_{si_1} + T_{si_2} + T_{si_3} \\
T_{si_1} &= \max\left\{T^{si_1}_{core_1} + T^{si_1}_{core_2}, T^{si_1}_{core_3} + T^{si_1}_{core_4}, T^{si_1}_{core_5}\right\} \\
&= T^{si_1}_{core_1} + T^{si_1}_{core_2} \\
T_{si_2} &= \max\left\{T^{si_2}_{core_1}, T^{si_2}_{core_4}, T^{si_2}_{core_5}\right\} \\
&= T^{si_2}_{core_4} \\
T_{si_3} &= \max\left\{T^{si_3}_{core_2}, T^{si_3}_{core_3}\right\} \\
&= T^{si_3}_{core_2}
\end{aligned}$$

For the second test schedule shown in Figure 6(b),

$$\begin{aligned}
T^{in}_{SOC} &= T^{in}_{tam_2} = T^{in}_{core_3} + T^{in}_{core_4} + T^{in}_{core_5} \\
T^{si}_{SOC} &= T_{si_1} + \max\left\{T_{si_2}, T_{si_3}\right\} = T_{si_1} + T_{si_2} \\
T_{si_1} &= \max\left\{T^{si_1}_{core_1} + T^{si_1}_{core_4} + T^{si_1}_{core_5}, T^{si_1}_{core_2} + T^{si_1}_{core_3}\right\} \\
&= T^{si_1}_{core_1} + T^{si_1}_{core_4} + T^{si_1}_{core_5} \\
T_{si_2} &= T^{si_2}_{core_1} + T^{si_2}_{core_4} + T^{si_2}_{core_5} \\
T_{si_3} &= T^{si_3}_{core_2} + T^{si_3}_{core_3}
\end{aligned}$$

In this example, T^{in}_{SOC} is the maximum core-internal test time of the individual TAMs for a given TAM architecture. The sequence of the core internal tests does not affect the value of T^{in}_{SOC}, hence the test time can easily calculated from

Data structure SI Test s

1. $C(s)$;	/* The set of cores involved */
2. $pattern(s)$;	/* Number of patterns */
3. $begin(s)$;	/* Schedule begin time */
4. $end(s)$;	/* Schedule end time */
5. $time_{si}(s)$;	/* SI Testing time */
6. $R_{tam}(s)$;	/* The set of TAMs involved */
7. $r_{btn}(s)$;	/* The bottleneck TAM for this SI test */

Data structure TestRail r

1. $C(r)$;	/* The set of cores on TestRail r */
2. $width(r)$;	/* TAM width of r */
3. $time_{in}(r)$;	/* Internal testing time */
3. $time_{si}(r)$;	/* Utilized SI testing time */
4. $time_{used}(r)$;	/* Utilized total testing time */

Fig. 7. Data structures for SI test and TestRail.

the TAM architecture. The calculation of T^{si}_{SOC}, however, is less straightforward because multiple TAMs may be involved. First, we need to calculate T_{si_j} for each SI test group j, which is determined by a single TAM denoted as the *bottleneck TAM* for this SI test (e.g., TAM_2 for SI test SI_2). Second, we need to schedule the SI tests to minimize T^{si}_{SOC}. We next elaborate on these two steps.

Data structure. The data structures that we use to store the SI test group information and the TestRail configuration are presented in Figure 7. The two data structures are updated whenever the SOC TAM design is changed. In particular, in data structure for *TestRail* r, we use $time_{in}(r)$, $time_{si}(r)$ and $time_{used}(r)$ to denote the internal testing time, the SI testing time and the utilized testing time on TAM r, respectively. For example, for TAM_3 shown in Figure 6(a):

—$time_{in}(r) = T^{in}_{core_5}$;
—$time_{si}(r) = T^{si_1}_{core_5} + T^{si_2}_{core_5}$;
—$time_{used}(r) = time_{in}(r) + time_{si}(r) = T^{in}_{core_5} + T^{si_1}_{core_5} + T^{si_2}_{core_5}$.

We use $time_{used}(r)$ to compare the actual utilization of TAM resources for different TAMs.

Calculation of test time for individual SI test. The pseudocode for the procedure to calculate the testing time for each SI test group is shown in Figure 8.

The procedure takes the TestRail architecture R_{SOC} and all the SI test groups S_{SOC} as inputs and calculates $time_{si}(s_i)$ for each SI test group s_i. In the inner loop (Lines 3–8), we check all the TAMs that are involved in SI test s_i to identify the TAM that determines $time_{si}(s_i)$. Line 4 finds out $C_{involved}$, that is, all the cores on TAM r_j that are involved in SI test s_i. Line 5 then calculates $time_{si}(r_j)$, the signal integrity testing time contributed by TAM r_j. We record the SI testing time $time_{si}(s_i)$ (Line 7) and the bottleneck TAM $r_{btn}(s_i)$ for SI test s_i (Line 8). Finally the procedure returns the SI tests with updated SI testing time (Line 9).

Algorithm 2 - CalculateSITestTime

INPUT: R_{SOC}, S_{SOC}
OUTPUT: S'_{SOC}

```
1. for all s_i ∈ S_SOC {
2.     initialize time_si(s_i) = 0;
3.     for all r_j ∈ R_tam(s_i) {
4.         C_involved = C(r_j) ∩ C(s_i);
5.         time_si(r_j) = Σ_{c_i ∈ C_involved} time_si(c_i);
6.         if(time_si(r_j) > time_si(s_i)) {
7.             time_si(s_i) = time_si(r_j);
8.             r_btn(s_i) = r_j; }
.     }
. }
9. return S'_SOC.
```

Fig. 8. Procedure for calculate SI testing time.

To calculate the time of an SI test group, the test time of each of the cores and each of the TAMs involved in this SI test needs to be computed. Therefore, the complexity of the *CalculateSITestTime* algorithm is $O(N_{SI}N_cN_{TAM})$, with N_{SI}, N_c and N_{TAM} corresponding to the number of SI test groups, the number of cores and the number of TAMs, respectively.

Scheduling of SI tests. Once $time_{si}(s_i)$ for each individual SI test s_i is known, we can schedule the SI tests to acquire the SOC signal integrity testing time T^{si}_{SOC}. This procedure is shown in Figure 9. Line 1 performs procedure *CalculateSITestTime* to calculate the testing time for each SI test. Line 2 initializes *unSchedSI*, the unscheduled SI tests and *currSchedTAMs*, the TAMs that are utilized by the SI tests currently under schedule. Line 3 initializes *currTime*, that is, the begin time for to-be-scheduled SI test. After the initialization, the following loop schedule SI test one by one (Lines 4-17). Inside the loop, we first try to find a SI test s^* that can be scheduled with begin time *currTime*, that is, s^* does not utilize any TAM in *currSchedTAMs* (Line 5). If such s^* can be found, we schedule it by updating $begin(s^*)$, $end(s^*)$, *currSchedTAMs*, and *unSchedSI* (Lines 7–10). If s^* is the last scheduled SI test, we shall update T^{si}_{SOC} as the end time of TAM $end(s^*)$. If all the unscheduled SI tests utilize the TAM resources in *currSchedTAMs* and hence cannot be scheduled with begin time *currTime*, we find *nextTime*, that is, the time for the first SI test that is expected to end after *currTime* (Line 14). We then update the begin time of the to-be-scheduled SI tests (Line 15) and *currSchedTAMs* based on the SI tests still under schedule (Line 16). Finally the procedure returns the SOC SI testing time T^{si}_{SOC} and the SI tests with updated schedule information.

Let us take the test architecture and test schedule shown in Figure 6(b) as an example to demonstrate the *ScheduleSITest* algorithm. After the internal tests have been scheduled, all the TAM resources are available and we choose to schedule SI test SI_1. However, before SI_1 is completed, there is no TAM available because SI_1 uses them to shift test patterns to all cores' wrapper cells. Therefore, the other SI tests have to wait for the completion of SI_1. Afterwards, the TAM resources occupied by SI_1 are released and SI_2 and SI_3 can be scheduled with TAM_2 and TAM_3 in parallel.

Algorithm 3 - ScheduleSITest

INPUT: R_{SOC}, S_{SOC}
OUTPUT: S'_{SOC}, T^{si}_{SOC}

1. $S'_{SOC} = CalculateSITestTime(R_{SOC}, S_{SOC})$;
2. initialize $unSchedSI = S'_{SOC}$; $currSchedTAMs = \emptyset$;
3. initialize $currTime = 0$;
4. **while** $unSchedSI \neq \emptyset$ {
5. find $s^* \in unSchedSI$ for which $R_{tam}(s^*) \cap currSchedTAMs = \emptyset$;
6. **if** found {
7. set $begin(s^*) = currTime$;
8. set $end(s^*) = begin(s^*) + time_{si}(s^*)$;
9. $currSchedTAMs = currSchedTAMs \cup R_{tam}(s^*)$;
10. $unSchedSI = unSchedSI \setminus \{s^*\}$;
11. **if**($unSchedSI == \emptyset$) {
12. $T^{si}_{SOC} = end(s^*)$; }
13. } **else** {
14. calculate $nextTime = end(s')$ such that
. $end(s') > currTime$ and $end(s')$ is the minimum;
15. set $currTime = nextTime$;
16. update $currSchedTAMs$;
. }
. }
17. **return** S'_{SOC}, T^{si}_{SOC}.

Fig. 9. Procedure for scheduling SI tests.

During the initialization process of the *ScheduleSITest* procedure, we need to conduct procedure *CalculateSITestTime* with complexity $O(N_{SI}N_cN_{TAM})$. Then in the inner loop, we schedule every SI test and the complexity is $O(N^2_{SI}N_{TAM})$ because the complexity to find an unscheduled SI test and to check the TAM availability and to update the occupancy information of this SI test is proportional to $N_{SI}N_{TAM}$. Since we have $N_{SI} \leq N_c$, the overall complexity of the *ScheduleSITest* procedure is $O(N_{SI}N_cN_{TAM})$.

4.3 TAM Design and Optimization

The preceeding discussion for calculating T^{si}_{SOC} is based on a given TAM architecture. What makes Problem P_{SI_opt} more difficult is that the testing time for a SI test $time_{si}(s)$ is not known until the SOC test-architecture is determined This makes P_{SI_opt} fundamentally different from the problem of designing and optimizing an SOC test-architecture for core internal-logic only. In the latter case, the testing time for each core can be pre-determined for a given TAM width [Xu and Nicolici 2005]. Unlike many test scheduling algorithms that schedule cores one after another and terminate after all cores are scheduled, the *TR—Architect* algorithm proposed in [Goel and Marinissen 2002] generates an initial test-architecture with all cores assigned to TAMs in the beginning and then optimizes this architecture in an iterative manner. This strategy is particularly attractive for interconnect SI test, since we are able to calculate the SI testing time in each optimization step. Therefore, we propose to adapt the *TR—Architect* algorithm for solving Problem P_{SI_opt} in this article. At the

Algorithm 4 -IdentifyBtnTAMs

INPUT: R_{SOC}, S_{SOC}
OUTPUT: R_{btn}

```
1. find r' with maximum internal testing time;
2. initialize R_btn = {r'};
3. find r* with maximum SI testing time;
4. for every core c ∈ C(r*) {
5.    for all s ∈ S_SOC that satisfies c ∈ C(s) {
6.       if r_btn(s) ∉ R_btn {
7.             R_btn = R_btn ∪ {r_btn(s)}; }
.     }
.  }
8. return R_btn.
```

Fig. 10. Procedure for identifying bottleneck TAMs of the SOC.

same time, this adaptation is not straightforward, as described in the following paragraphs.

Identifying bottleneck TAMs. The basic idea of the *TR—Architect* algorithm is to optimize T^{in}_{SOC} at the TAM level by merging TAMs and/or distributing free TAM wires to the bottleneck TAM, that is, the TAM with the longest T^{in}_{tam}. As a result, we define the *bottleneck TAMs* of the SOC (in contrast to the single bottleneck TAM for an SI test) to be those which are critical to the test time; T_{SOC} is reduced if extra wires are assigned to them; the remaining TAMs are referred to as *non-bottleneck TAMs* of the SOC. In *TR—Architect*, there exists only a single bottleneck TAM at a time during the optimization process. Either two non-bottleneck TAMs are merged with less TAM width to release freed TAM resources to the bottleneck TAM, or the bottleneck TAMs is merged with another TAM to decrease T^{in}_{SOC} [Goel and Marinissen 2002].

In our problem, as we try to minimize $T_{SOC} = T^{in}_{SOC} + T^{si}_{SOC}$, it is possible that multiple bottleneck TAMs exist at the same time. That is, in addition to the bottleneck TAM for core-internal logic test, each SI test has its own bottleneck TAM, which may affect the total SOC testing time T_{SOC}. For example, for the schedule shown in Figure 6(a), the bottleneck TAM for SI_2 (i.e., TAM_2) is a bottleneck TAM for the SOC; on the other hand, for the schedule shown in Figure 6(b), the bottleneck TAM for SI_3 (i.e., TAM_1) is not a bottleneck TAM for the SOC. For the schedule shown in Figure 6(a), TAM_1 and TAM_2 are bottleneck TAMs and TAM_3 is a non-bottleneck TAM, while for the schedule shown in Figure 6(b), TAM_2 is a bottleneck TAM and TAM_1 is a non-bottleneck TAM.

The procedure to identify SOC bottleneck TAMs is shown in Figure 10. The bottleneck TAM for core-internal logic test is guaranteed to affect T_{SOC}. Therefore, in Line 1 and Line 2, we find this bottleneck TAM r' and it is identified as a SOC bottleneck TAM (e.g., TAM_1 in Figure 6(a)). Next, in Line 3, we find the TAM r^* with the longest SI test time. Each core on r^* might be involved in several SI tests. For every one of these SI tests, we identify its bottleneck TAM r_{btn}. Since the SI test bottleneck TAMs identified in this way must affect the total SOC testing time T_{SOC}, we treat each of them as the bottleneck TAM of the

Algorithm 5 - distributeFreeWires

INPUT: R_{SOC}, S_{SOC}, *numFreeWires*
OUTPUT:R'_{SOC}

1. initialize $R'_{SOC} = R_{SOC}$;
2. **for** i=1 **to** *numFreeWires* {
3. $R'_{btn} = IdentifyBtnTAMs(R'_{SOC}, S_{SOC})$;
4. find $r' \in R'_{btn}$ such that
. T'_{SOC} is the minimum when $width(r') = width(r') + 1$;
5. $width(r') = width(r') + 1$; update $time_{in}(r')$;
6. update $time_{si}(r)$ and $time_{used}(r)$ for all $r \in R'_{SOC}$;
. }
7. **return** R'_{SOC}.

Fig. 11. Procedure for distributing free TAM wires.

SOC (Lines 4–7). From this procedure, it can be seen that the bottleneck TAM for those SI tests that are not involved with any core on r^* can be ignored, for example, the bottleneck TAM for SI_3 shown in Figure 6(b). For the test architecture and test schedule in the example of 6(a), TAM_1 is the bottleneck TAM for internal test. On the other hand, both TAM_1 and TAM_2 are bottleneck TAMs for SI tests. Therefore, the bottleneck TAM set for this schedule is composed of TAM_1 and TAM_2.

In the *IdentifyBtnTAMs* algorithm, every SI test involving a core on the TAM with the longest SI test time is checked for its bottleneck TAM . Therefore, the complexity of this algorithm is $O(N_{SI}N_c)$.

Algorithm for problem P_{SI_opt}. Next we introduce our algorithm for Problem P_{SI_opt}. Similar to the *TR—Architect* algorithm, we first create an initial TestRail architecture and optimize it by merging TAMs and distributing free TAM wires afterwards. There are two key questions during the optimization process, namely, How to find out the merging candidate and merge them and How to distribute free TAM wires. Because there may exist multiple bottleneck TAMs at the same time in our problem, the answers to these two questions highlight the main differences between our algorithm and the *TR—Architect* algorithm proposed in [Goel and Marinissen 2002].

The procedure for distributing free TAM wires is shown in Figure 11. The procedure takes the given TestRail architecture R_{SOC}, all the SI tests S_{SOC} and the number of free TAM wires *numFreeWires* as inputs. The free TAM wires are distributed iteratively to the bottleneck TAMs (Lines 2–6). Since we may have multiple bottleneck TAMs at the same time, we select one of them based on the criteria that T_{SOC} is the minimum after obtaining the extra TAM wire (Line 4). Because R_{SOC} is changed whenever a free TAM wire is assigned (Line 5), $time_{si}(r)$ and $time_{used}(r)$ for every $r \in R_{SOC}$ are updated (Line 6). Finally the procedure outputs the new TestRail architecture R'_{SOC} with all free TAM wires assigned.

Let us take the test architecture and test schedule shown in Figure 6(a) as an example to explain the *distributeFreeWires* algorithm. Suppose there is one more free wire to be distributed to one of the three TAMs. The algorithm tries to distribute it to either TAM_1 or TAM_2 (they are the bottleneck TAMs), compares

Algorithm 6 - mergeTAMs

INPUT: R_{SOC}, S_{SOC}, r_1
OUTPUT: R'_{SOC}

1. initialize $R'_{SOC} = R_{SOC}$; $R_{find} = R_{SOC} \setminus \{r_1\}$;
2. initialize $T_{min} = T_{SOC}$
3. **for** i=1 **to** $|R_{find}|$ {
4. set $width_{min} = \max\{width(r_i), width(r_1)\}$;
5. set $width_{max} = width(r_i) + width(r_1)$;
6. set $maxFreeWires = width_{max} - width_{min}$;
7. set $R_{find} = R_{find} \setminus \{r_i\}$;
8. **for** numFreeWires=0 **to** $maxFreeWires$ {
9. set $r_{temp} = r_i \cup r_1$; $width(r_{temp}) = maxFreeWires - numFreeWires$;
10. set $R_{temp} = R_{find} \cup \{r_{temp}\}$;
11. $R_{temp} = distributeFreeWires(R_{temp}, S_{SOC}, numFreeWires)$;
12. **if**($T_{temp} \leq T_{min}$) {
13. $T_{min} = T_{temp}$;
14. $R'_{SOC} = R_{temp}$;}
. }
. }
15. **return** R'_{SOC}.

Fig. 12. Procedure for merging TAMs.

the overall SOC testing time for the two choices, and finally selects the one with smaller testing time.

To distribute a free wire, we need to identify all bottleneck TAMs at the current time and schedule the SI tests with one of the bottleneck TAMs added with one more wire, which means that we need to run SI test scheduling $O(N_{TAM})$ times in the worst case. Therefore, if the number of free wires is N_{FW}, the overall complexity of the *distributeFreeWires* procedure is $O(N_{FW}N_{SI}N_cN_{TAM}^2)$.

The procedure for merging TAMs is shown in Figure 12. In this procedure, with the given TestRail architecture R_{SOC}, all the SI tests S_{SOC} and one of the merging candidate r_1 as inputs, we look for another TAM candidate in $R_{find} = R_{SOC} \setminus \{r_1\}$, which leads to the lowest testing time after merging with r_1. After initialization (Lines 1 and 2), we enumeratively try every TAM r_i in R_{find} as the other merging candidate (Lines 3-14). What's more, we also try to merge r_i and r_1 with different TAM width in the range of $width_{min} = \max\{width(r_i), width(r_1)\}$ (Line 4) and $width_{max} = width(r_i) + width(r_1)$ (Line 5). The intuition behind this is that we may be able to merge two TAMs with less TAM width and the extra free TAM wires can be assigned to other bottleneck TAMs to reduce T_{SOC}. The procedure outputs the TestRail architecture R'_{SOC} with the lowest testing time after merging (Line 15). It is also possible that we cannot find a merging plan to reduce T_{SOC}. In such case, the original TestRail architecture is returned.

Again, let us take the test architecture and test schedule shown in Figure 6(a) as an example to explain the *mergeTAMs* procedure. Consider the case that TAM_1 is the candidate TAM and we need to find another TAM to be merged with TAM_1 and redistribute the TAM wires. We try each and every one of the other TAMs (TAM_2 and TAM_3 in this case) to be merged with TAM_1. If

TAM_1 is merged with TAM_2, the newly merged TAM, TAM_{12}, will be assigned $\max\{width(r_1), width(r_2)\}$ wires initially. By now there are two TAMs (TAM_{12} and TAM_3) and $width(r_1) + width(r_2) - \max\{width(r_1), width(r_2)\}$ free wires. We then distribute these free wires to the two TAMs one wire at a time to achieve maximum total test time reduction. The merging of TAM_1 and TAM_3 is similar to the above procedure. We will then select the merging with smaller test time and the corresponding TAM architecture will be generated from the *mergeTAMs* algorithm.

In the *mergeTAM* procedure, to find a TAM to be merged with the candidate TAM for maximum test time reduction, we need to try all other TAMs (i.e., $N_{TAM}-1$ times). For each of these TAMs, we need to call the *distributeFreeWires* procedure multiple times (the worst case complexity is $O(W_{max})$ times). Since the worst-case complexity for the utilized *distributeFreeWires* procedure is $O(W_{max}N_{SI}N_cN_{TAM}^2)$, the overall complexity of the *mergeTAM* procedure is $O(W_{max}^2N_{SI}N_cN_{TAM}^3)$.

The pseudocode for our top-level algorithm $TAM\_Optimization$ for Problem P_{SI_opt} is presented in Figure 13, which is adapted from the *TR—Architect* algorithm [Goel and Marinissen 2002]. First, we create a start solution (Lines 1–16). This mainly consists of three steps. In Step 1 (Lines 2–5), we assign each core to a one-bit wide TAM and we calculate the testing time of core internal logic $time_{in}(r)$, the testing time of interconnects $time_{si}(r)$ and the actual utilized testing time $time_{used}(r)$ for every $r \in R_{SOC}$. In case $W_{max} < |R_{SOC}|$, we do not have enough TAM wires and hence we need to merge TAMs together (Lines 7–13). We first sort R_{SOC} based on the total utilized testing time in each TAM (Line 9), then $r_{W_{max}+1}$ is merged iteratively with another TAM r_i. We select this merging candidate r_i based on the criteria that T_{SOC} is the minimum after merging with $r_{W_{max}+1}$ (Line 10). Since R_{SOC} is changed after merging, $time_{si}(r)$ and $time_{used}(r)$ for every $r \in R_{SOC}$ are updated (Line 13). In the case $W_{max} > |R_{SOC}|$, we have extra free TAM wires left and procedure *distributeFreeWires* is called to distribute them.

Next, we optimize the TAM architecture by merging the TAM with the lowest $time_{used}$ with another TAM (Lines 17-23). We first sort R_{SOC} in nonincreasing order and we select $r_{|R_{SOC}|}$ as one of the merging candidate r_1, then we call procedure *mergeTAMs* to search for another TAM to merge with r_1 and possibly redistribute TAM resources to reduce T_{SOC}. This is an iterative procedure and it stops when no reduction in T_{SOC} can be achieved (Lines 22–23). Afterwards, we try to further optimize the TAM architecture by trying to merge the TAM with the longest $time_{used}$ with another TAM (Lines 25–30) and merging other TAMs (Lines 31-36). Finally, *TAM_Optimization* tries to minimize T_{SOC} by iteratively moving one core from bottleneck TAMs of the SOC to another TAM, if possible (Line 37).

As can be observed in Figure 13, the computational complexity of the *TAM_Optimization* algorithm is mainly determined by the bottom-up and top-down optimization procedures, which require the $mergeTAM$ procedure to be carried out $O(W_{max})$ times in the worst case. Therefore, the complexity of the overall algorithm is $O(W_{max}^3N_{SI}N_cN_{TAM}^3)$.

Algorithm 7 - TAM_Optimization

INPUT: C_{SOC}, W_{max}, S_{SOC}
OUTPUT: R_{SOC}

1. initialize $R_{SOC} = \emptyset$;
. /* Create a start solution */
2. **for** all $c_i \in C_{SOC}$ {
3. create TAM r_i such that $C(r_i) = \{c_i\}$;
4. $width(r_i) = 1$; $time_{in}(r_i) = time_{in}(c_i)$;
5. $R_{SOC} = R_{SOC} \cup \{r_i\}$;
. }
6. calculate $time_{si}(r)$ and $time_{used}(r)$ for all $r \in R_{SOC}$;
7. **if** ($W_{max} < |R_{SOC}|$) {
8. **for**($i = W_{max} + 1$ to $|R_{SOC}|$) {
9. **sort** R_{SOC} such that $time_{used}(r_1) \geq$
. $time_{used}(r_2) \geq ... \geq time_{used}(r_{|R_{SOC}|})$;
10. find $r_i (1 \leq i \leq W_{max})$ such that
. T_{SOC} is the minimum when merging with $r_{W_{max}+1}$;
11. $r_i = r_i \cup r_{W_{max}+1}$; update $time_{in}(r_i)$;
12. $R_{SOC} = R_{SOC} \setminus r_{W_{max}+1}$;
13. update $time_{si}(r)$ and $time_{used}(r)$ for all $r \in R_{SOC}$;
. }
14. } **else if** ($|R_{SOC}| < W_{max}$) {
15. set $numFreeWires = W_{max} - |R_{SOC}|$;
16. $R_{SOC} = distributeFreeWires(R_{SOC}, S_{SOC}, numFreeWires)$;
. }
. /* Optimize the TestRail architecture from bottom-up */
17. set $isImproved = true$;
18. **while**($isImproved$ **AND** $|R_{SOC}| > 1$) {
19. set $initTestingTime = T_{SOC}$;
20. **sort** R_{SOC} such that $time_{used}(r_1) \geq time_{used}(r_2) \geq ... \geq time_{used}(r_{|R_{SOC}|})$;
21. $R_{SOC} = mergeTAMs(R_{SOC}, S_{SOC}, r_{|R_{SOC}|})$;
22. **if**($T_{SOC} == initTestingTime$) {
23. $isImproved = false$; }
. }
. /* Optimize the TestRail architecture from top-down*/
24. set $isImproved = true$;
25. **while**($isImproved$ **AND** $|R_{SOC}| > 1$) {
26. set $initTestingTime = T_{SOC}$;
27. **sort** R_{SOC} such that $time_{used}(r_1) \geq time_{used}(r_2) \geq ... \geq time_{used}(r_{|R_{SOC}|})$;
28. $R_{SOC} = mergeTAMs(R_{SOC}, S_{SOC}, r_1)$;
29. **if**($T_{SOC} == initTestingTime$) {
30. $isImproved = false$; $R_{skip} = \{r_1\}$; }
. }
31. **while**($R_{skip} \neq R_{SOC}$) {
32. set $R_{temp} = R_{SOC} \setminus R_{skip}$; $initTestingTime = T_{SOC}$;
33. find $r^* \in R_{temp}$ such that $time_{used}(r^*)$ is the maximum;
34. $R_{SOC} = mergeTAMs(R_{SOC}, S_{SOC}, r^*)$;
35. **if**($T_{SOC} = initTestingTime$) {
36. $R_{skip} = R_{skip} \cup \{r_1\}$; }
. }
37. $coreReshuffle(R_{SOC}, S_{SOC})$;
38. **return** R_{SOC}, T_{SOC}.

Fig. 13. Algorithm for solving P_{S1_opt}.

Table II. Comparison of Compressed Test Data Volume with Different SI Test Pattern Counts and Different Core Groups

		$N_r = 1{,}000$			$N_r = 5{,}000$		
	N_g	N_c	D_s	ΔD_s (%)	N_c	D_s	ΔD_s (%)
SOC g1023	1	35	383950	/	156	1711320	/
	2	39	324230	15.55	161	1347898	21.24
	4	47	314018	18.21	170	1235982	27.78
	8	58	**311198**	18.95	192	**1233058**	27.95
		$N_r = 10{,}000$			$N_r = 50{,}000$		
	N_g	N_c	D_s	ΔD_s (%)	N_c	D_s	ΔD_s (%)
SOC p34392	1	258	1486596	/	1247	7185214	/
	2	285	1196228	19.53	1294	5748670	19.99
	4	316	1221490	17.83	1375	5394738	24.92
	8	376	**1185938**	20.22	1555	**5371424**	25.24
		$N_r = 10{,}000$			$N_r = 50{,}000$		
	N_g	N_c	D_s	ΔD_s (%)	N_c	D_s	ΔD_s (%)
SOC p93791	1	270	5750460	/	1266	26963268	/
	2	285	4715026	18.01	1304	21219284	21.30
	4	308	4355544	24.26	1355	19734208	26.81
	8	317	**3944650**	31.40	1401	**19067892**	29.28

N_r: Initial interconnect test pattern count; N_g: Number of partitions;
N_c: Number of compacted patterns; D_s: Test data volume;
$\Delta D_s = \frac{D_s(N_g=1)-D_s}{D_s(N_g=1)} \times 100\%$.

5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the proposed solution, experiments were carried out for three ITC'02 benchmark SOCs from Marinissen et al. [2002], namely, g1023, p34392, and p93791. Without loss of generality, we do not consider hierarchy in the testing of core-internal logic. Since the topology of these benchmark SOCs and the connection between embedded cores are not available, we cannot obtain the test patterns for core-external interconnect SI faults for these benchmark SOCs. Therefore, we generate random test patterns for our experiments in the following manner. For the smaller SOC g1023, we generate $1,000$ and $5,000$ random patterns, respectively. For p34392 and p93791 we generate $10,000$ and $50,000$ random patterns, respectively. Each test pattern targets one victim and N_a ($2 \leq N_a \leq 6$) random aggressors. Suppose the victim wire connects two cores $Core_a$ and $Core_b$. Then at least $N_a - 2$ aggressor lines are between these two cores. In addition, we assume that a 32-bit bus is utilized in all the three SOCs. The probability that the bus is used by a test pattern is set to 50%. If the bus is used for a particular pattern, we randomly generate $1 \sim N_a$ specified bits in the postfix of the pattern (see Section 3).

Table II shows the results for our two-dimensional test compaction scheme. We partition the SOCs in N_g parts using the *hMetis* package [Selvakkumaran and Karypis 2003]. Therefore, the row with $N_g = 1$ is for the case when the test set is compressed without partitioning. The parameters N_c and D_s denote the compacted test pattern count and test data volume (calculated as the sum of the test pattern length times the test pattern count in each SI test group), respectively. ΔD_s is the percentage reduction in test data volume compared

Table III. Test Application Time Comparison for SOC g1023

SOC g1023								
	$N_r = 1,000$							
W_{max}	$T_{TR\text{-}Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	119791	93913	92723	94519	**92708**	92708	22.61	1.28
16	70161	**50349**	52073	50842	50840	50349	28.24	0.00
24	44603	**33652**	35038	34998	33900	33652	24.55	0.00
32	35145	26196	27290	26045	**24895**	24895	29.16	4.97
40	28619	22697	22672	**21163**	21619	21163	26.05	6.76
48	28619	20989	20618	18920	**18184**	18184	36.46	13.36
56	28619	20986	20618	19255	**19213**	19213	32.87	8.45
64	28619	20941	20618	**18998**	19139	18998	33.62	9.28
	$N_r = 5,000$							
W_{max}	$T_{TR\text{-}Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	296814	182173	175354	160032	**155091**	155091	47.75	14.87
16	197211	92026	103685	84900	**80199**	80199	59.33	12.85
24	111395	67172	60975	59042	**56124**	56124	49.62	16.45
32	100249	51278	43629	43750	**43261**	43261	56.85	15.63
40	98098	40350	39222	38528	**35343**	35343	63.97	12.41
48	98098	35835	32088	31921	**30212**	30212	69.20	15.69
56	98098	32780	31435	**28758**	29737	28758	70.68	12.27
64	98098	32260	29947	**25770**	27246	25770	73.73	20.12

N_r: Initial interconnect test pattern count; W_{max}: Given SOC TAM width;
$T_{TR-Arch}$: Test time obtained by optimizing the SOC TAM architecture for InTest only;
T_{g_i}: Test time obtained using the proposed $TAM\_Optimization$ algorithm with the SOC cores partitioned into i groups;
$T_{min} = \min_i\{T_{g_i}\}$; $\Delta T_{TR-Arch} = \frac{T_{TR-Arch} - T_{min}}{T_{TR-Arch}} \times 100\%$; $\Delta T_g = \frac{T_{g_1} - T_{min}}{T_{g_1}} \times 100\%$.

to the case when $N_g = 1$. It can be observed from the table that, with test pattern merging only, the compaction over the original test set is only $\Delta V = 3\%$ (i.e., $\frac{N_c}{N_r} \times 100\%$ when $N_g = 1$). With test-pattern-length reduction by SI test grouping, we are able to further reduce the test data volume by more than 20% on top of ΔV.

Tables III, IV, and V present results for the SOC test application time, measured in terms of the number of clock cycles. We compare between the following cases: (i) optimizing T^{in}_{SOC} using only the *TR—Architect* algorithm [Goel and Marinissen 2002] ($T_{TR\text{-}Arch}$); (ii) optimizing T_{SOC} using our proposed algorithm *TAM_Optimization* for several SI test pattern counts N_r and the SI test grouping strategy. Note that $T_{[Goel\,and\,Marinissen\ 2002]}$ is determined by optimizing the SOC TAM architecture in terms of core-internal test time T^{in}_{SOC} only, and then computing the total test time T_{SOC} by adding to T^{in}_{SOC} the time needed for SI test. The parameter T_{g_i} denotes the SOC test time obtained using the proposed *TAM_Optimization* algorithm when the SI tests are partitioned into i parts; $T_{min} = \min_i\{T_{g_i}\}$, which corresponds to the test-architecture that we choose; $\Delta T_{TR\text{-}Arch}$ and ΔT_g are computed as $\Delta T_{TR-Arch} = \frac{T_{TR-Arch} - T_{min}}{T_{TR-Arch}} \times 100\%$ and $\Delta T_g = \frac{T_{g_1} - T_{min}}{T_{g_1}} \times 100\%$, respectively. Note that ΔT_g quantifies the benefit derived from our two-dimensional compaction strategy over the one-dimensional compaction scheme that reduces only the test-pattern count. We can see that more than 20% test-time reduction can be achieved in some cases

Table IV. Test Application Time Comparison for SOC p34392

SOC p34392								
	$N_r = 10,000$							
W_{max}	$T_{TR-Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	2499024	2460860	2189687	2154845	**2153117**	2153117	13.84	12.51
16	1215631	1100901	1157715	**1094507**	1097405	1094507	9.96	0.58
24	863107	801474	828945	816120	**793520**	793520	8.06	0.99
32	644122	612926	636092	674518	**609820**	609820	5.33	0.51
40	604177	583720	564348	**563295**	601969	563295	6.77	3.50
48	604177	563253	**558339**	559762	558829	558339	7.59	0.87
56	604177	561607	**556440**	556910	556972	556440	7.90	0.92
64	604177	561607	556404	556430	**555728**	555728	8.02	1.05
	$N_r = 50,000$							
W_{max}	$T_{TR-Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	2862976	2582663	2685154	2523919	**2477706**	2477706	13.46	4.06
16	1436474	1323698	1313744	**1312050**	1385386	1312050	8.66	0.88
24	1120118	1035895	950950	**910648**	955818	910648	18.70	12.09
32	872581	720887	714779	**700121**	704810	700121	19.76	2.88
40	832636	657092	652525	**633428**	641855	633428	23.92	3.60
48	832636	638850	624037	**607619**	610755	607619	27.02	4.89
56	832636	619399	607829	**599433**	600075	599433	28.01	3.22
64	832636	619399	601601	**593223**	600177	593223	28.75	4.23

Table V. Test Application Time Comparison for SOC p93791

SOC p93791								
	$N_r = 10,000$							
W_{max}	$T_{TR-Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	4695241	4064748	4144176	**4034628**	4075377	4034628	14.07	0.74
16	2298848	2072745	2029390	**2018862**	2037524	2018862	12.18	2.60
24	1622729	1412641	**1392753**	1434277	1392762	1392753	14.17	1.41
32	1217985	1039646	1030373	1057084	**1026253**	1026253	15.74	1.29
40	1041244	841096	**829314**	856368	832855	829314	20.35	1.40
48	959503	708385	713458	710631	**697217**	697217	27.34	1.58
56	808771	606411	600039	611748	**590176**	590176	27.03	2.68
64	784023	528642	527789	521335	**497329**	497329	36.57	5.92
	$N_r = 50,000$							
W_{max}	$T_{TR-Arch}$ (cc)	T_{g_1} (cc)	T_{g_2} (cc)	T_{g_4} (cc)	T_{g_8} (cc)	T_{min} (cc)	$\Delta T_{TR-Arch}$ (%)	ΔT_g (%)
8	6037849	5407578	5106050	5124448	**5044333**	5044333	16.45	6.72
16	3286880	3134458	2579717	2588326	**2506449**	2506449	23.74	20.04
24	2374697	1825802	1782340	1759197	**1700872**	1700872	28.38	6.84
32	1943073	1432611	1405361	1389043	**1344381**	1344381	30.81	6.16
40	1485965	1124129	1138165	**1075325**	1080395	1075325	27.63	4.34
48	1522634	945505	923041	906744	**883350**	883350	41.99	6.57
56	1665028	836821	814316	824757	**751578**	751578	54.86	10.19
64	1902274	708727	675423	707474	**654393**	654393	65.60	7.67

(e.g., for SOC g1023, when $W_{max} = 64$ and $N_r = 5,000$). The magnitude of this reduction depends on the initial SI test set and the core configurations. It should be noted that the maximum value of N_g does not necessarily lead to minimum testing time. This is mainly because, when we have a large value of N_g, we have more SI tests to schedule and conflicts are more likely to arise during the scheduling process, thus leading to longer testing time.

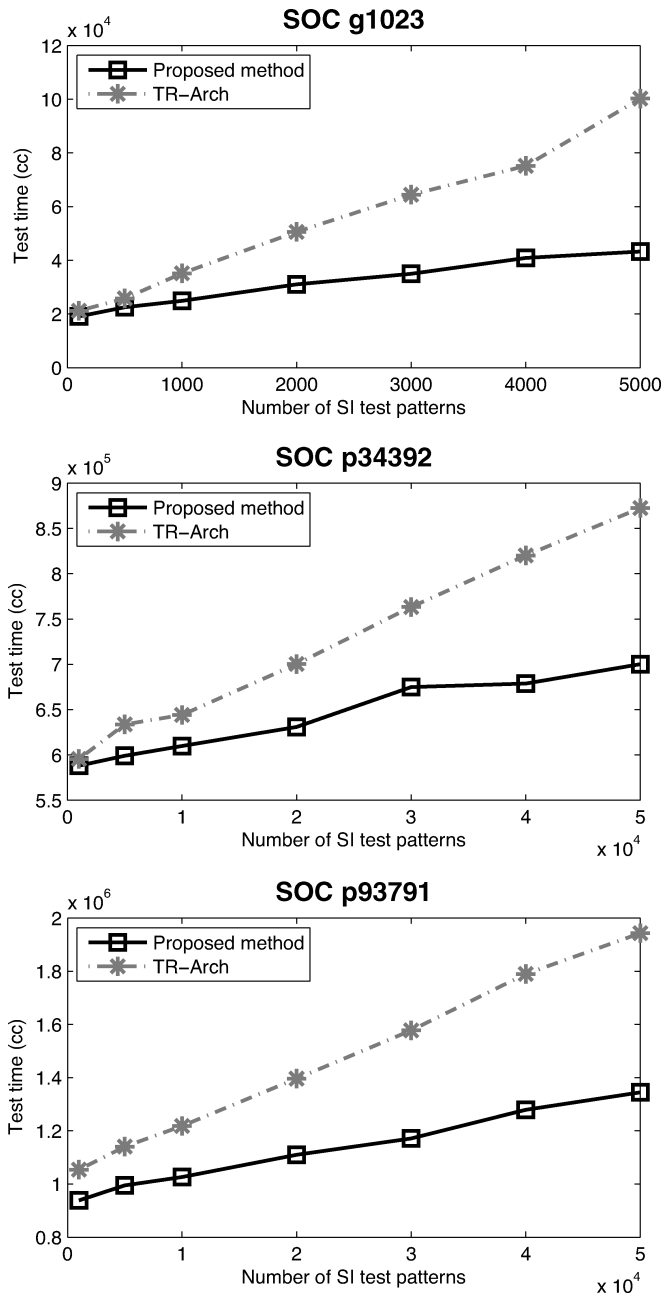

Fig. 14. Test time comparison with different SI test pattern counts.

From Tables III, IV, and V, we note that obliviously optimizing SOC test-architectures, without considering interconnect SI faults, leads to much higher test time. This gap grows with an increase in the pattern count for the SI faults and the associated percentage of SI testing time in T_{SOC}. We can also see that when W_{max} is small, there is no significant advantage in using proposed algorithm; in a few cases, worse results are obtained compared to SI-oblivious TAM optimization (e.g., for SOC p34392, when $W_{max} = 8$ and $N_r = 10,000$). This is mainly because the TAM design solution space is small for smaller values of W_{max}, therefore, similar TAM architectures are obtained with different optimization criteria. When W_{max} is higher, we have more freedom during the TAM design process and hence the improvement offered by the new optimization procedure is more noticeable. We can also observe that, for SOC p34392, when $W_{max} > 32$, T_{min} remains nearly the same. This is because the testing time for $Core_{18}$, the largest embedded core, dominates T_{SOC}.

We attribute the few exceptions to the nature of the heuristics that explore a limited part of the solution space.

When the number of SI test pattern grows, it is more important to optimize the test-architecture for both core-internal faults and interconnect SI faults. In Figure 14, we vary the original SI test-pattern count while keeping the TAM width at 32 bits. We compare the test time obtained using the proposed method with the test time for the baseline method based on Goel and Marinissen [2002]. The number of (given) SI test patterns is increased from 100 to 5,000 for SOC g1023, and from 1,000 to 50,000 for SOC p34392 and p93791, respectively. It can be observed that the gap between the two solutions becomes larger when the number of SI test patterns increases, which highlights the importance of optimizing the SOC test-architecture for interconnect SI faults for newer technology generations.

6. CONCLUSION

As feature sizes shrink with newer process technologies, and clock frequencies increase, the test cost due to interconnect signal integrity faults can be considerable. To cope with this problem, we have presented a new TAM optimization flow for core-based SOCs that considers test times for both core-internal logic and core-external signal integrity faults on interconnects. This is in contrast to prior work on test infrastructure design for core-based system-on-a-chip, which has focused on minimizing only the test time for core-internal logic. We have investigated the impact of interconnect SI tests on SOC test-architecture design and optimization. We have also presented a compaction method for SI test sets such that the test data volume is reduced. Experimental results for the ITC'02 benchmarks show that the proposed approach can significantly reduce the overall testing time for core-internal logic and core-external interconnects. The test times obtained using this approach are noticeably less than that obtained by a baseline based on the *TR—Architect* algorithm, which only considers the core-internal test time during optimization. As part of future work, we are considering the role of different core frequencies for reducing the test time [Xu and Nicolici 2006]. We are also investigating how interconnect layout information can be used for more effective test-infrastructure optimization.

ACKNOWLEDGMENTS

The authors thank Professor Nicola Nicolici of McMaster University for motivating discussions and insightful comments.

REFERENCES

ARORA, S. 1998. The approximability of NP-hard problems. In *Proceedings of the Annual ACM Symposium on Theory of Computing*. 337–348.

ATTARHA, A. AND NOURANI, M. 2002. Test pattern generation for signal integrity faults on long interconnects. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 336–341.

BAI, X., DEY, S., AND RAJSKI, J. 2000. Self-test methodology for at-speed test of crosstalk in chip interconnects. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 619–624.

BECER, M., VAIDYANATHAN, R., OH, C., AND PANDA, R. 2004. Crosstalk noise control in an SoC physical design flow. *IEEE Trans. Comput. Aid. D. 23,* 4, 488–497.

CAIGNET, F., DELMAS-BENDHIA, S., AND SICARD, E. 2001. The challenge of signal integrity in deep-submicrometer CMOS technology. In *Proceedings of the IEEE 89,* 4, 556–573.

CHEN, L., BAI, X., AND DEY, S. 2001. Testing for interconnect crosstalk defects using on-chip embedded processor cores. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 317–322.

CHEN, T.-S., LEE, C.-Y., AND KAO, C.-H. 2004. An efficient noise isolation technique for SOC application. *IEEE Trans. Electr. Dev. 51,* 2, 255–260.

CHEN, W.-Y., GUPTA, S. K., AND BREUER, M. A. 1999. Test generation for crosstalk-induced delay in integrated circuits. In *Proceedings of the IEEE International Test Conference (ITC)*. 191–200.

CUVIELLO, M., DEY, S., BAI, X., AND ZHAO, Y. 1999. Fault modeling and simulation for crosstalk in system-on-chip interconnects. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.

DUTTA, S., JENSEN, R., AND RIECKMANN, A. 2001. Viper: a multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Des. Test Comput. 18,* 5, 21–31.

EBADI, Z. S. AND IVANOV, A. 2003. Time domain multiplexed TAM: implementation and comparison. In *Proceedings of the Design, Automation, and Test in Europe (DATE)* 732–737.

FIDUCCIA, C. M. AND MATTHEYSES, R. M. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 175–181.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Publishers.

GOEL, S. K., CHIU, K., MARINISSEN, E. J., NGUYEN, T., AND OOSTDIJK, S. 2004. Test infrastructure design for the nexperia^{TM} home platform PNX8550 system chip. In *Proceedings of the Conference Design, Automation, and Test in Europe (DATE)*. 108–113.

GOEL, S. K. AND MARINISSEN, E. J. 2002. Effective and efficient test architecture design for SOCs. In *Proceedings of the IEEE International Test Conference (ITC)*, 529–538.

GULER, M. AND KILIC, H. 1999. Understanding the importance of signal integrity. *IEEE Circuits Devic. 15,* 6, 7–10.

IEEE STD. 1500. 2004. *IEEE Standard for Embedded Core Test—IEEE Std. 1500-2004*. IEEE, New York.

IYENGAR, V., CHAKRABARTY, K., AND MARINISSEN, E. J. 2002. Co-optimization of test wrapper and test access architecture for embedded cores. *J. Elect. Test. 18,* 2, 213–230.

JHA, N. AND GUPTA, S. 2003. *Testing of Digital Systems*. Cambridge University Press.

KAO, W. H., LO, C.-Y., BASEL, M., AND SINGH, R. 2001. Parasitic extraction: current state of the art and future trends. In *Proceedings of the IEEE 89,* 5, 729–739.

KUNDU, S., ZACHARIAH, S. T., CHANG, Y.-S., AND TIRUMURTI, C. 2005. On modeling crosstalk faults. *IEEE Trans. Comput. Aid. D. 24,* 12, 1909–1915.

LARSSON, E. AND FUJIWARA, H. 2003. Test resource partitioning and optimization for SOC designs. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 319–324.

LARSSON, E. AND PENG, Z. 2002. An integrated framework for the design and optimization of SOC test solutions. *J. Elect. Test. 18,* 4/5, 385–400.

MARINISSEN, E. J. ET AL. 1998. A structured and scalable mechanism for test access to embedded reusable cores. In *Proceedings of the IEEE International Test Conference (ITC)*, 284–293.

MARINISSEN, E. J., GOEL, S. K., AND LOUSBERG, M. 2000. Wrapper design for embedded core test. In *Proceedings of the IEEE International Test Conference (ITC)*, 911–920.

MARINISSEN, E. J., IYENGAR, V., AND CHAKRABARTY, K. 2002. A set of benchmarks for modular testing of SOCs. In *Proceedings of the IEEE International Test Conference (ITC)*, 519–528.

MASSOUD, Y., MAJORS, S., KAWA, J., BUSTAMI, T., MACMILLEN, D., AND WHITE, J. 2002. Managing On-Chip Inductive Effects. *10,* 6 (December), 789–798.

NAFFZIGER, S. 1999. Design methodologies for interconnects in GHz+ICs. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*.

NAHVI, M. AND IVANOV, A. 2004. Indirect test architecture for SoC testing. *IEEE Trans. Comput. Aid. 23,* 7, 1128–1142.

NATARAJAN, S., BREUER, M. A., AND GUPTA, S. K. 1998. Process variations and their impact on circuit operation. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*. 73–81.

NORDHOLZ, P., TREYTNAR, D., OTTERSTEDT, J., GRABINSKI, H., NIGGEMEYER, D., AND WILLIAMS, T. W. 1998. Signal integrity problems in deep submicron arising from interconnects between cores. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 28–33.

NOURANI, M. AND ATTARHA, A. 2001. Built-in self-test for signal integrity. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 613–618.

SEKAR, K. AND DEY, S. 2002. LI-BIST: a low-cost self-test scheme for SoC logic cores and interconnects. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 417–422.

SELVAKKUMARAN, N. AND KARYPIS, G. 2003. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 726–733.

SIRISAENGTAKSIN, W. AND GUPTA, S. K. 2002. Enhanced crosstalk fault model and methodology to generate tests for arbitrary inter-core interconnect topology. In *Proceedings of the IEEE Asian Test Symposium (ATS)*. 163–169.

TABATABAEI, S. AND IVANOV, A. 2002. An embedded core for sub-picosecond timing measurements. In *Proceedings of the IEEE International Test Conference (ITC)*. 129–137.

TEHRANIPOUR, M. H., AHMED, N., AND NOURANI, M. 2003. Testing SoC interconnects for signal integrity using bounary scan. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 158–163.

TEHRANIPOUR, M. H., AHMED, N., AND NOURANI, M. 2004. Testing SoC interconnects for signal integrity using extended JTAG architecture. *IEEE Trans. Comput. Aid. D. 23,* 5, 800–811.

VARMA, P. AND BHATIA, S. 1998. A structured test re-use methodology for core-based system chips. In *Proceedings of the IEEE International Test Conference (ITC)*, 294–302.

WANG, L.-T., STROUND, C. E., AND TOUBA, N. A., Eds. 2007. *System-on-Chip Test Architectures: Nanometer Design for Testability*. Morgan Kaufmann Pub.

XU, Q. AND NICOLICI, N. 2003. On reducing wrapper boundary register cells in modular SOC testing. In *Proceedings of the IEEE International Test Conference (ITC)*, 622–631.

XU, Q. AND NICOLICI, N. 2004. Multi-frequency test access mechanism design for modular SOC testing. In *Proceedings of the IEEE Asian Test Symposium (ATS)*, 2–7.

XU, Q. AND NICOLICI, N. 2005. Resource-constrained system-on-a-chip test: a survey. In *Proceedings of the IEEE Conference on Computers and Digital Techniques 152,* 1, 67–81.

XU, Q. AND NICOLICI, N. 2006. Multifrequency tam design for hierarchical socs. *IEEE Trans. Comput. Aid. D. 25,* 1, 181–196.

YANG, S.-Y., PAPACHRISTOU, C. A., AND TAIB-AZAR, M. 2001. Improving bus test via I_{DDT} and boundary scan. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 307–312.

ZHANG, T. AND SAPATNEKAR, S. S. 2004. Simultaneous shield and buffer insertion for crosstalk noise reduction in global routing. In *Proceedings of the International Conference on Computer Design (ICCD)*. 93–98.

ZHAO, D. AND UPADHYAYA, S. 2005. Dynamically partitioned test scheduling with adaptive TAM configuration for power-constrained SoC testing. *IEEE Trans. Comput. Aid. D. 24,* 6, 956–965.

ZHAO, Y., DEY, S., AND CHEN, L. 2004. Double sampling data checking technique: an online testing solution for multisource noise-induced errors on on-chip interconnects and buses. *IEEE Trans. VLSI Syst. 12,* 7, 746–755.

ZOU, W., REDDY, S. M., POMERANZ, I., AND HUANG, Y. 2003. SOC test scheduling using simulated annealing. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*, 325–330.

Received December 2007; revised May 2008; accepted September 2008