A Heuristic for Concurrent SOC Test Scheduling with Compression and Sharing

Anders Larsson, Erik Larsson, Petru Eles, and Zebo Peng

Embedded Systems Laboratory
Linköpings Universitet
SE-582 83 Linköping, Sweden

***Abstract*[1]-The increasing cost for System-on-Chip (SOC) testing is mainly due to the huge test data volumes that lead to long test application time and require large automatic test equipment (ATE) memory. Test compression and test sharing have been proposed to reduce the test data volume, while test infrastructure and concurrent test scheduling have been developed to reduce the test application time. In this work we propose an integrated test scheduling and test infrastructure design approach that utilizes both test compression and test sharing as basic mechanisms to reduce test data volumes. In particular, we have developed a heuristic to minimize the test application time, considering different alternatives of test compression and sharing, without violating a given ATE memory constraint. The results from the proposed Tabu Search based heuristic have been validated using benchmark designs and are compared with optimal solutions.**

I. INTRODUCTION

The increasing complexity of modern electronic systems together with a shorten production cycle has forced the designers to employ reuse based design approaches. System-on-Chip (SOC) is an example of such reuse based design approach where pre-designed and pre-verified blocks of logic so called cores are integrated into a system. The SOC design approach eases the task of the system designers. However, testing of SOCs has been shown to be a costly step in the manufacturing process due to huge test data volume that requires large automatic test equipment (ATE) memory and long test application times.

Several techniques have been proposed to reduce the cost of SOC testing. Test architecture design and test scheduling techniques that reduce the test application times have been proposed in [1, 2, and 3]. The problem of reducing the required test data volume has been targeted by using compression methods in [4, 5, and 6] or by using test sharing as proposed in [7, 8, and 9].

Traditionally, only one or few of the above mentioned techniques are considered when reducing the cost of SOC testing. Therefore, an integrated approach has been proposed by us in [10] where both compression and sharing, together with test architecture design and scheduling are used. For the work presented in [10], however, the following two limitations can be identified; first, only sequential transportation of test data is allowed, which can lead to underutilization of the test access mechanism (TAM). Second, only small and medium sized instances of the problem can be solved due to the long optimization time required by the Constraint Logic Programming (CLP) [11] technique used to generate optimal solutions. In this work we solve the problem of TAM underutilization by allowing multiple tests to be transported concurrently and the problem of long optimization times for large designs is solved by generating a suboptimal solution using a Tabu search based heuristic.

The rest of this paper is organized as follows. In Section II the used SOC test architecture is described and in Section III a motivational example is presented. The problem is formulated in Section IV and in Section V the Tabu Search based heuristic for minimizing test application time is described. The experimental results are presented in Section VI and conclusions are in Section VII.

II. SOC TEST ARCHITECTURE

In this section the SOC test architecture for test compression and sharing is described.

We assume given a fixed number of W TAM wires, which are connected to the ATE, as illustrated in Fig. 1. Also illustrated in Fig. 1 is the placement of the decoder and comparator. The decoder is used for the decompression of compressed tests and the evaluation of the produced responses is performed using the comparator. The TAM wires are used to transport test stimuli and test responses to and from the cores. The cores are scan tested and the scanned elements at each core are formed to wrapper chains that are connected to the TAM wires. Each core c_i is associated with one dedicated test T_i, which is assumed to be given. These dedicated tests are used to generate new test alternatives for the cores by using compression and/or sharing, as illustrated in Fig. 1.

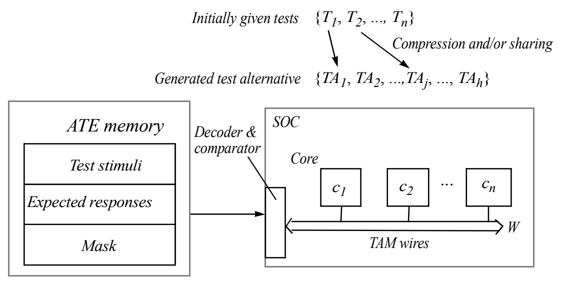
Initially given tests $\{T_1, T_2, ..., T_n\}$
Compression and/or sharing
Generated test alternative $\{TA_1, TA_2, ..., TA_j, ..., TA_h\}$
ATE memory
Test stimuli
Expected responses
Mask
Decoder & comparator
SOC
Core
c_1
c_2
...
c_n
W
TAM wires

Fig. 1. Test architecture and ATE memory organization.

1. The research is partially supported by the Strategic Integrated Electronic Systems Research (STRINGENT) program, financed by the Swedish Foundation for Strategic Research.

In this work we make use of an architecture proposed in [12], which can handle test application of compressed tests without requiring additional logic such as multiple-input signature-register (MISR). The general idea of the test architecture in [12] is to store compressed test stimuli, compressed expected responses, and compressed masks in the ATE memory, as illustrated in Fig. 1. The mask is used to identify the positions of the specified bits in the expected responses. The compressed test stimuli, expected responses, and masks are all sent to the SOC under test and decompressed on the chip.

In contrast to using a processor core for test data decompression [12], we make use of an on-chip decoder based on the fixed-length Nine-Coded Compression (9C) technique proposed in [13]. (A detailed description of the test application using 9C can be found in [10]). The 9C compression is done by generating code words that are stored in the ATE memory. The code words are transported from the ATE to the decoder using the ATE frequency f_{ATE} and the decompressed stimuli are transported and applied to the wrapper chains using the scan frequency f_{scan}. The decoder is bypassed if the test is not compressed. When compression is used the test stimuli is applied with a frequency f_{scan}, which is lower than the frequency of the ATE, f_{ATE}. The reason for this is the reduction of on-chip control signals. The value of f_{scan} is given as follows:

$$f_{scan} = \begin{cases} f_{ATE}, & \text{without using compression} \\ 9CConst \times f_{ATE}, & \text{with compression} \end{cases} \qquad (1)$$

where *9CConst* is given by the number of test bits that each codeword contain (K=8) [13], which is divided by he maximum number of clock cycles needed to apply the longest codeword that the 9C coding uses (12+8) [13]. The scan frequency f_{scan} is used to calculate the test application time τ_i for a test or test alternative, at core c_i as follows:

$$\tau_i = ((1 + max\{si_i, so_i\}) \times l + min\{si_i, so_i\}) / f_{scan}, \qquad (2)$$

where si_i and so_i are the length of the longest wrapper scan-in and scan-out chain of core c_i respectively and l is the number of test sequences. The test responses from the core are evaluated using the expected responses and the mask using a comparator.

Let us use a small example to illustrate the test architecture when sharing is used. The example depicted in Fig. 2 consists of two cores, c_1 and c_2. The example illustrated in Fig. 2(a) shows how the wrapper chains are connected to TAM wires when the cores are tested using dedicated tests, i.e., no sharing is used. Fig. 2(b) shows the connections when the core c_1 and c_2 is tested using a shared test that is broadcasted. In order to separate the different produced responses from different cores the produced responses from each core is transported on separate TAM wires.

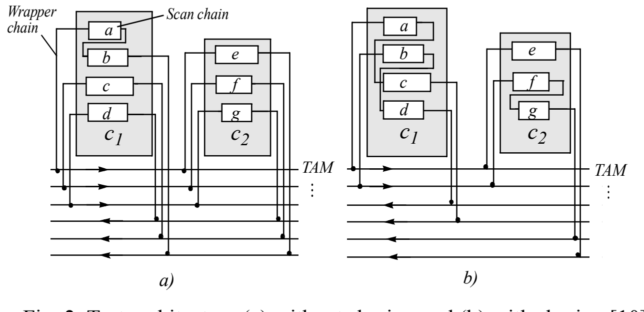

Fig. 2. Test architecture (a) without sharing and (b) with sharing [10].

The number of wrapper chains w_i for a test or alternative test depends on the number of cores z that shares the test, and is given by:

$$w_i = \lfloor W / (z+1) \rfloor \qquad (3)$$

If no sharing is used ($z=1$) $w_i = W/2$, which means that half of the TAM width is used for the transportation of test stimuli and the second half is used for produced responses as illustrated in Fig. 2(a). In the case when two cores share a test ($z=2$) $w_i = W/3$; one third of the TAM width is occupied transporting the test stimuli that are broadcasted to both cores and two thirds are used for the produced responses, one third for each core separately as illustrated in Fig. 2(b).

III. Motivational Example

In [10], sequential transportation of tests is used and concurrent test application is only achieved by sharing one test between several cores. One problem with using a sequential approach is the potential underutilization of the TAM wires. Such underutilization of the TAM wires occurs when the number of wrapper chains that a test uses is small. A small number of wrapper chains is used when a core has a small number of scan chains or if the scan chains are unbalanced, i.e., have a large difference in length, which may lead to few balanced wrapper chains [2]. Another example of when a small number of wrapper chains may be used is when so called hard cores are utilized. By allowing multiple tests to be transported, and applied, concurrently, the TAM will be utilized more efficiently and the test time can be reduced.

How the test application time can be reduced is illustrated using a small example presented in Fig. 3. The design in Fig. 3 consist of three cores each with its number of scan chains sc_1, sc_2, and sc_3 that are formed into a number of wrapper chains. In the example we assume that no compression or sharing is used. The width of the TAM W is set to 20. The number of wrapper chains that a test uses is determined such that half of the TAM wires are used for transportation of test stimuli and the second half for the produced responses. Core c_1 has 15 scan chains that are grouped into 10 wrapper chains and will occupy the full bandwidth of the TAM, however, c_2 has only 6 scan chains which means that only twelve of the TAM wires will be used when T_2 is transported, the other 8 TAM wires will not be used.

In Fig. 3(a) sequential scheduling is used and the

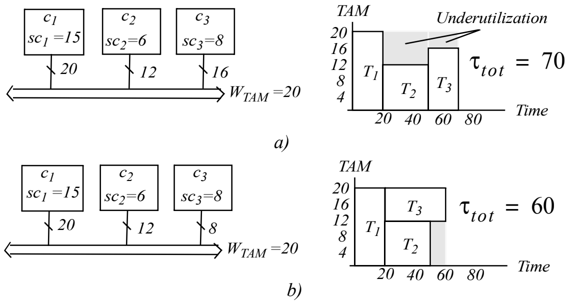

Fig. 3. Motivational example (a) sequential scheduling and (b) concurrent scheduling.

underutilization of the TAM is demonstrated. The test application time τ_{tot} for the system using sequential scheduling will be equal to 70ms. A better utilization of the TAM is illustrated in Fig. 3(b) where the scan chains of core c_3 in this case are grouped into 4 wrapper chains. By reducing the number of wrapper chains the test application time of T_3 will be longer, however, the test application time τ_{tot} for the schedule will be reduced from 70ms to 60ms since T_3 can now be scheduled at the same time as T_2.

IV. Problem Formulation

Given a system with n cores, $c_1, c_2, \ldots, c_n$, where for each core c_i the following is given:

- sc_i - the number of scan chains,
- ff_{ij} - the number of flip-flops in scan chain j,
- wi_i - the number of input wrapper cells,
- wo_i - the number of output wrapper cells,
- $nff_i = \sum_{j=1}^{sc_i} f_{ij}$ - the total number of flip-flops,
- $T_i = \{TS_i, ER_i, M_i\}$ - an initially given dedicated test consisting of test stimuli TS_i, expected responses ER_i, and a test mask M_i,
- $TS_i = (ts_{i1}, \ldots, ts_{il})$ - a sequence of l test stimuli patterns, where ts_{ik} consists of $nff_i + wi_i$ bits and each bit can be *0, 1*, or *x*,
- $ER_i = (er_{i1}, \ldots, er_{il})$ - a sequence of l expected response patterns, where er_{ik} consists of $nff_i + wo_i$ bits and each bit can be *0, 1*, or *x*,
- $M_i = (m_{i1}, \ldots, m_{il})$ - a sequence of l mask patterns, where m_{ik} consists of $nff_i + wo_i$ bits and each bit can be *0* or *1*. A *1* indicates that the corresponding bit in the produced responses is a care bit and should be checked with the expected responses otherwise it is a don't-care bit and should be masked.

Also given for the system is the number of TAM wires, W. For the ATE the number of bits that can be stored in the ATE memory M and the clock frequency of the ATE f_{ATE} are given. Furthermore, a *compres*s, and a *share* function are available in order to generate alternative tests. The tests (or test alternatives) are scheduled concurrently such that the TAM width constraint W is not violated.

Given the above, our problem is to select one test alternative[2] for each core c_i and to determine the architecture (the TAM wire usage) and start time t_i such that the test application time τ_{tot} is minimized without exceeding the memory constraint M. The test application time for a schedule with n tests is given as:

$$\tau_{tot} = max((\forall i, i = \{1, 2, \ldots, n\})(t_i + \tau_i)), \qquad (4)$$

where τ_i is the test time of core c_i and t_i is the start time when the test is applied to core c_i.

V. Test Application Time Optimization

In this section the Tabu search based algorithm used to solve the problem defined in Section IV is described.

The Tabu search algorithm takes as input a list of test alternatives. This list is generated by using the initially given tests $T_1, T_2, \ldots, T_n$ at a pre-process stage in which the compress, and share functions are used. The share function [10] takes two tests as input and generate a new test. The compress function [10] takes one test as input and generate a new test. The generated tests are added to the list of alternative tests.

2. From here and through the rest of the paper we use T_i to denote a given dedicated test or an alternative test.

The Tabu search based heuristic is used to search for the optimal solution, i.e., the solution with the shortest test application time. Tabu search [14] is a form of local neighborhood search, which at each step evaluates the neighborhood of the current solution and the best solution is selected as the new current solution. Unlike local search which stops if no improved new solution is found, Tabu search continues the search from the best solution in the neighborhood even if that solution is worse than the current one. To prevent cycling, the most recently visited solutions are marked as tabus meaning that they are not allowed to be repeated until the tabu status has expired.

The pseudo code for the proposed Tabu search based algorithm is presented in Fig. 4 and Fig. 5. Fig. 4 presents the initial solution and the inner loop of Tabu search and in Fig. 5, the outer loop is presented. The initial solution is generated using the dedicated tests for each core (line 2 and 3 in Fig. 4). Since no compression is used, this initial solution is likely to violate the ATE memory limit. In such a case, the initial solution is modified by randomly changing some of the dedicated tests to a compressed test. This change is done using a random function that is repeated for a maximum given number *MAX_ITERATIONS* of times (line 5 to 9).

When a valid initial solution has been found, the Tabu search will continue the search for a better solution by exploring the neighborhood (line 11 to 48). Here follows a description of the neighborhood and the search for improved solutions.

Each core will be assigned a list of tests, a *core_test_list*. The *core_test_list* consists of those tests that can be used to test a core. A solution consists of n tests where each position in the solution is associated with a specific core and contains one test from that cores *core_test_list*. In our algorithm, the neighborhood is determined by the possible changes of test for each core and is defined as follows: A test $T_i(k)$, where k is used to denote the position of the test in the *core_test_list*, can be replaced with either the test at position $k-1$ or at position $k+1$. The neighborhood and the corresponding moves are further illustrated in Fig. 6 using the example system in Fig. 3. In Fig. 6 the current solution contains T_1, T_9, and T_6, which are used to test c_1, c_2, and c_3, respectively. Fig. 6 also shows the *core_test_list* for c_2. The *core_test_list* shows that the possible moves for the test T_9 is T_7 $(k-1)$ and T_{11} $(k+1)$. The same principle is applied for all positions in the current solution, which means that each test in the current solution will be associated with two possible moves.

The reason for using this neighborhood is that it will lead to small changes of the current solution, hence, the search will continue in the same region of the solution space. For example, one shared test is likely to be changed to another shared test. An alternative neighborhood could be to randomly select a test. Such random move, however, will lead to a bigger change of the

```
1)  tabu_search_bld (Part one)
        //Generate initial solution
2)      for each core $c_i$
3)          solution $\cup$ {$T_i$}
4)      iterations = 0
5)      while memory_exceeded(solution)
6)          random_compress_solution(solution)
7)          iterations++
8)          if iterations > MAX_ITERATIONS then
9)              return "No solution found"
10)     best_solution = solution
    // Start the inner loop
11)     Start:
12)     moves[] = generate_neighborhood_solutions(solution)
13)     calculate_delta_tat(solution, moves)
14)     sort_according_to_delta_tat(moves)
15)     for each move $m_j$
16)         delta_tat = get_delta_tat($m_j$)
17)         if delta_tat < 0 then
18)             new_solution = get_new_solution(solution, $m_j$)
19)             if memory_exceeded(new_solution) then
20)                 delta_tat = delta_tat + mem_penalty
21)             if $m_j$ not in tabu_list or get_tat_bld(new_solution)
22)                 < get_tat_bld(best_solution) then
23)                 incr_frequeny($m_j$)
24)                 solution = new_solution
25)                 goto Accept
26)     for each move $m_j$
27)         update_move_tat($m_j$, get_frequency($m_j$))
28)     for each move $m_j$
29)         new_solution = get_new_solution(solution, $m_j$)
30)         if memory_exceeded(new_solution) then
31)                 delta_tat = delta_tat + mem_penalty
32)         if $m_j$ not in tabu_list or get_tat_bld(new_solution)
33)                 < get_tat_bld(best_solution) then
34)                 incr_frequeny($m_j$)
35)                 solution = new_solution
36)                 goto Accept
37)     $m_1$ = get_move_from_tabu_list(tabu_list)
38)     new_solution = get_new_solution(solution, $m_1$)
39)     incr_frequeny($m_1$)
40)     Accept:
41)     if get_tat_bld(solution) < get_tat_bld(best_solution) and
42)         !memory_exceeded(solution) then
43)         iterations_without_better = 0
44)         best_solution = solution
45)     else
46)         iterations_without_better++
47)     if iterations_without_better < MAX_INNER_LOOP then
48)         goto Start
```

Fig. 4. Tabu search heuristic (initial solution and inner loop).

current solution.

When a move has been applied, it is marked as a tabu and is stored in a tabu list. The tabu list will have a length of *MAX_TABUS*, hence, a move will be marked as tabu for *MAX_TABUS* iterations.

For each move, a *delta_tat* value is calculated (line 13) that corresponds to the decrease of the test application time when that move is applied to the current solution. The moves are then sorted decreasingly according to the *delta_tat* value (line 14). If

```
1) tabu_search_bld (Part two)
//Outer loop (diversification)
2)  if restarts < MAX_OUTER_LOOP then
3)      restarts++
4)      iterations_without_better = 0
5)      if cycles_detected(solution) then
6)          goto Stop
    //Generate diversified solution
7)      no_to_change = n*divers_ratio/100
8)      while(divers_count < no_to_change)
9)          gen_diversified_solution(solution, divers_count)
10)         divers_count++
11)     goto Start
12) Stop:
13) return best_solution
```

Fig. 5. Tabu search heuristic (outer loop).

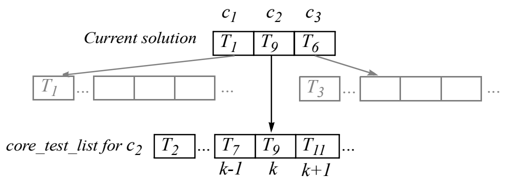

Fig. 6. Neighborhood definition.

delta_tat is less than zero a new solution is generated (line 18). In order to make the search efficient, solutions that violates the ATE memory constraint can be accepted but are penalized using a *mem_penalty* parameter (line 20). The *mem_penalty* parameter is defined as follows:

$$mem\_penalty = MEM\_CONST \times \tau_{Tot} \qquad (5)$$

If the move is not in the tabu list or if the move would generate a solution better than the best solution found so far, the current solution is assigned the new solution (line 24). In such case the frequency, i.e., the number of times the move has been applied, is increased, and the solution is accepted (line 40 to 48).

If no improving move is found the search continues by recalculating the *delta_tat* considering the frequency of the moves. In this step, moves with a high frequency are considered to likely be part of a good solution, hence, they will get priority when the search continues (line 28 to 36). If no improving move can be found, the move that lead to the smallest increase of the test application time is assigned to the current solution (line 37), which means that an uphill move will be applied. The inner loop is stopped if no improving move is found for *MAX_INNER_LOOP* consecutive tries.

While the inner loop is used to search for a solution by making small changes to the current solution, the outer loop, presented in Fig. 5, will diversify the search by generating a new solution, which is dramatically different from the current solution. This diversification will force the search into a new region of the solution space that is not reachable using the inner loop. The outer loop is executed for a maximum of *MAX_OUTER_LOOP* iterations (line 2 in Fig. 5). The value of *MAX_OUTER_LOOP* is defined as follows:

$$MAX\_OUTER\_LOOP = \alpha + \beta \times n \qquad (6)$$

where α and β are two parameters. The reason for not having a fixed value of *max_outer_loop* is to allow the search to be executed for longer time for large examples.

The diversification is done by randomly changing a number of the tests in the current solution (line 7 to 10). How many tests that are changed is controlled by a variable *divers_ratio*, which ranges from 0 to 100, *divers_ratio*=100 means that all tests in the solution will be replaced, *divers_ratio*=50 means that 50% of the tests will be replaced. The *divers_ratio* will have a large value in the beginning, which means that solutions from different regions of the solution space will be generated. The diversified solution is then used in the inner loop where it is improved. The outer loop also has a mechanism to detect if a cycle has occurred (line 5). If a cycle is detected in the outer loop, the algorithm is stopped. When the Tabu search terminates, the solution with the shortest test application time is returned (line 13).

The selected tests are then scheduled and assigned to TAM wires according to a Bottom-Left-Decreasing (BLD) algorithm [15], which has been implemented in the *get_tat_bld* function used to acquire the test application time for a solution. The pseudo code for the BLD algorithm is presented in Fig. 7 (b). First, the tests for the solution, which is given as input to the algorithm, are sorted decreasingly according to their TAM usage (line 4). The tests that occupy the full bandwidth of the TAM will be placed first. At this point, all redundant, shared, tests are removed leaving *n_tests* distinct tests to be scheduled.

Each test is then scheduled as early as possible while leaving as much empty TAM wires as possible, therefore the name, BLD. The first test will be selected and scheduled at time zero. If the TAM wires are not fully utilized a new loop is used to search for a test, which can be scheduled at the same time. Once a test has been scheduled the test application time is updated (line 12). The search is repeated until the bandwidth is fully utilized or no other test can be scheduled. When all tests have been scheduled the test application time τ_{tot} is returned.

VI. Experimental Results

For the experiments, the following *ITC'02* benchmark designs have been used: *d695, g1023, p34392*, and *p93791* [16]. The characteristics of the benchmark designs are collected in Table I. The name of the design is listed in Column one, while columns two and three contain the number of input tests and required ATE memory. The last column, Column four, lists the number of available TAM wires. The required ATE memory is the amount of memory required to store the initially given tests, i.e., without using compression or sharing. The number of available TAM wires has been given by us.

```
//Calculate the test application time using BLD scheduling
1)  get_tat_bld(solution)
2)      test_application_time = 0
3)      used_TAM = 0
4)      tests[n_tests] = sort_tests_TAM(solution)
5)      for each test $T_i$ in tests
6)          if not_scheduled($T_i$) then
7)              schedule_test_at_bottom_left($T_i$)
8)              update(test_application_time, used_tam)
9)          while(used_tam < MAX_TAM)
10)             $T_j$ = search_test(tests, MAX_TAM- used_tam)
11)             schedule_test_at_bottom_left($T_j$)
12)             update(test_application_time, used_tam)
13)     return test_application_time
```

Fig. 7. BLD scheduling algorithm.

For the design *d695*, the test stimuli and expected responses (with don't cares marked) are given in [17]. We have randomly generated the test stimuli and expected responses for the other designs, *g1023*, *p34392*, and *p93791*, using 95% don't cares [18]. It is assumed that the designs are tested using an ATE running at f_{ATE} of 100MHz. The parameters used in the Tabu search heuristic have been determined, using extensive experiments, as follows:

- *MAX_ITERATIONS* = 1000,
- *MAX_TABUS* = 15,
- *MAX_INNER_LOOP* = 10,
- α = 50,
- β = 3,
- *MEM_CONST* = 0.4.

Since the number of possible test alternatives *h* will be huge, we restrict the sharing of tests to only two tests, i.e., if two tests have been shared and a new shared test generated, this new shared test will not be shared with another test. To further reduce the number of test alternatives a maximum share ratio *MSR* is defined as follows [10]:

$$MSR = 100 - \left(\frac{max(size(TA_i), size(TA_j))}{size(TA_i) + size(TA_j)}\right) \times 100, \qquad (7)$$

where $size(TA_i)$ is the number of bits in a test alternative. By setting a limit on the *MSR* during the pre-process stage it is possible to avoid those alternatives that have little possibility to be part of the optimal solution and therefore will not be explored during the optimization. After extensive experiments the parameter *MSR* is set to 35%.

For each design, two experiments are performed. First, using an ATE memory constraint of 1/3 of the required ATE memory (Table I, Column 3), and second with an ATE memory constraint of 2/3 of the required ATE memory.

The experimental results are presented in Table II. Column one lists the designs, Column two lists the number of test alternatives considered during the optimization, and Column three lists the ATE memory constraint. Column four to nine lists the test application time and optimization time (cpu time) for three different optimization strategies. Column four to seven contains the results obtained using sequential scheduling optimized using CLP and our proposed Tabu search respectively. Column eight and nine contains the results when concurrent

TABLE I
Benchmark Characteristics

Design	No. of input tests	Memory requirement (kbit)	No. of TAM wires (W)
d695	10	3398	48
g1023	14	4789	60
p34392	19	125332	60
p93791	32	1122802	60

scheduling is used and optimized using Tabu search and BLD.

The results show that the proposed Tabu search generates solutions which are close to the optimal solution generated using CLP. For the design *p93791*, CLP was not able to find the optimal solution in reasonable time and therefore a time-out is used to terminate the algorithm. The timeout is set to 5 hours and when this time is reached the best solution found so far is reported. In the case when a small memory is used, CLP was not able to find any solution for *p93791*. On average, the test application time using Tabu search is only 8.2% longer than the optimal solution (the results from *p93791* is not included) and Tabu search requires much shorter optimization time for medium and large designs. Only for the smallest design, *d695*, the CLP outperforms Tabu search in terms of optimization time. The results also show an average of 15% decrease of the test application time when concurrent scheduling is used, compared with using sequential scheduling.

VII. Conclusions

Due to huge memory requirements and long test application times, SOC testing has been shown to be a costly step in the manufacturing process. In this work, the test application time is minimized using a concurrent test scheduling approach where multiple tests can be transported and applied to different cores at a time. Furthermore, both test compression and test sharing are used as basic mechanisms to reduce test data volumes. We have developed a Tabu search based heuristic to minimize the test application time under a given ATE memory constraint. The experimental results show that our proposed Tabu search based heuristic is able to find solutions that are close to the optimal. The results also show that the test application time can be further decreased when concurrent scheduling is used compared to sequential scheduling.

References

[1] E. J. Marinissen, S. K. Goel, and M. Lousberg, “Wrapper Design for Embedded Core Test,” *Proceedings of International Test Conference,* pp. 911-920, 2000.

[2] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, “Test Wrapper and Test Access Mechanism Co-Optimization for System-on-Chip,” *Proceedings of International Test Conference (ITC)*, pp. 1023–1032, 2001.

[3] U. Ingelsson, S. K. Goel, E. Larsson, and E. J. Marinissen, “Test Scheduling for Modular SOCs in an Abort-on-Fail Environment,” *Proceedings of IEEE European Test Symposium (ETS),* pp. 8–13, 2005.

[4] A. Chandra and K. Chakrabarty, “A unified approach to reduce SOC test data volume, scan power and testing time,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 22, Issue 3, pp. 352–363, 2003.

[5] A. Jas, J. Ghosh-Dastidar, M. Ng, and N. Touba, “An Efficient Test Vector Compression Scheme Using Selective Huffman Coding,” *IEEE Transaction on Computer-Aided Design (TCAD)*, Vol. 22, pp. 797–806, 2003.

[6] A. Chandra and K. Chakrabarty, “System-on-a-Chip Test-Data Compression and Decompression Architectures Based on Colomb Codes,” *IEEE Transaction on CAD of Integrated Circuits and Systems*, Vol. 20, Issue 3, pp. 355–368, 2001.

[7] T. Shinogi, Y. Yamada, T. Hayashi, T. Yoshikawa, and S. Tsuruoka, “Test Vector Overlapping for Test Cost Reduction in Parallel Testing of Cores with Multiple Scan Chains,“ *Digest of Papers of Workshop on RTL and High Level Testing (WRTLT)*, pp. 117–122, 2004.

[8] K-J. Lee, J-J. Chen, C-H. Huang, “Broadcasting Test Patterns to Multiple Circuits,*” IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, Issue. 12, pp. 1793–1802, 1999.

[9] A. Larsson, E. Larsson, P. Eles, and Z. Peng , “SOC Test Scheduling with Test Set Sharing and Broadcasting,” *Proceedings of IEEE Asian Test Symposium*, pp. 162–169, 2005.

[10] A. Larsson, E. Larsson, P. Eles, and Z. Peng, “Optimized Integration of Test Compression and Sharing for SOC Testing,” *Proceedings of Design, Automation and Test in Europe (DATE)*, Accepted for publication, 2007.

[11] J. Jaffar and J.-L. Lassez, “Constraint Logic Programming,” *Proceedings of the 14th. ACM Symposium on Principles of Programming Languages (POPL),* pp. 111–119, 1987.

[12] E. Larsson and J. Persson, “An Architecture for Combined Test Data Compression and Abort-on-Fail Test,” *Proceedins of the 12th Asia and South Pacific Design Conference (ASP-DAC)*, pp. 726–731, 2007.

[13] M. Tehranipoor, M. Nourani, and K. Chakrabarty, “Nine-Coded Compression Technique for Tesing Embedded Cores in SoCs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* ,Vol. 13, Issue 6, pp. 719–731, 2005.

[14] C. R. Reeves (Editor), “Modern Heuristic Techniques for Combinatorial Problems”, *Blackwell Scientific Publications*, ISBN 0-470-22079-1, 1993.

[15] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher, “Exhaustive Approaches to 2D Rectangular Perfect Packings,” *Elsevier Science Direct, Information Processing Letters,* Vol. 90, Issue 1, pp. 7-14, 2004.

[16] E. J. Marinissen, V. Iyengar, and K. Chakrabarty, “A Set of Benchmarks for Modular Testing of SOCs,” *Proceedings of the IEEE International Test Conference (ITC)*, pp. 519–528, 2002.

[17] S. Kajihara and K. Miyase, “On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits,” *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 364–369, 2001.

[18] E. Larsson, A. Larsson, and Z. Peng, “Linköping University SOC Test Site,” *http://www.ida.liu.se/labs/eslab/soctest*, 2006.

TABLE II
Experimental Results

Design	No. of test alternatives	Memory constraint (kbit)	CLP (sequential scheduling) [10]		Tabu search heuristic (sequential scheduling)		Tabu search heuristic (concurrent scheduling)	
			Test application time (ms)	CPU time (s)	Test application time (ms)	CPU time (s)	Test application time (ms)	CPU time (s)
d695	40	1132	0.44	1.0	0.47	8.4	0.47	12.7
		2265	0.36	0.9	0.37	8.7	0.35	11.8
g1023	70	1596	0.94	40.4	1.07	18.9	0.81	29.6
		3193	0.55	29.8	0.63	18.4	0.42	17.2
p34392	74	41784	16.43	437.7	19.32	26.6	15.34	50.5
		83551	10.93	1902.8	10.95	45.5	9.56	73.2
p93791	214	374267	n.s.	18000.0[a]	306.09	387.6	306.09	382.0
		748534	117.24	18000.0[a]	175.30	348.7	169.13	363.5

a. Terminated by time-out.