Heuristics for Adaptive Temperature-Aware SoC Test Scheduling Considering Process Variation

Nima Aghaee, Zebo Peng, and Petru Eles
Embedded Systems Laboratory (ESLAB), Linkoping University, Sweden
{nima.aghaee, zebo.peng, petru.eles}@liu.se

Abstract—High working temperature and process variation are undesirable effects for modern systems-on-chip. The high temperature should be taken care of during the test. On the other hand, large process variations induce rapid and large temperature deviations causing the traditional static test schedules to be suboptimal in terms of speed and/or thermal-safety. A remedy to this problem is an adaptive test schedule which addresses the temperature deviations by reacting to them. Our adaptive method is divided into a computationally intense offline-phase, and a very simple online-phase. In this paper, heuristics are proposed for the offline phase in which the optimized schedule tree is found. In the online-phase, based on the temperature sensor readings the appropriate path in the schedule tree is traversed. Experiments are made to tune the proposed heuristics and to demonstrate their efficiency.

I. Introduction

Two challenges for deep submicron integration are high power density and process variation [1]. The power density for a System-on-Chip (SoC) during test compared to its normal operation is high enough to put testing in trouble by considerably raising the overheating risk [2]. Efficient temperature-aware test scheduling techniques have been developed in order to minimize the test application time and avoid overheating [3, 4]. These methods neglect, however, the thermal consequences of the process variation and focus only on minimization of the test application time while maintaining the chips temperature under a given limit [3, 4].

The negative thermal consequence of process variation is unpredictability of the thermal behavior of the chip. It means that identical test vectors will result in a variety of different temperatures for different chips and cores. The difference between the expected temperature (estimated by simulation) and the actual temperature (measured by sensors) is called temperature error, which captures all errors generated due to different power/temperature-related effects. These negative effects include ambient temperature fluctuations, voltage variations, and process variation. For traditional technologies, temperature error is small enough to be negligible or to allow worst-case design with negligible performance penalty [3, 4]

The general trend of increase in power density and process variation will eventually lead to a situation where temperature errors cannot be ignored any longer. Therefore, the thermal consequences of the process variation should be taken into account in order to develop efficient test process. In [5], two process variation aware methods are proposed in order to maximize the test throughput by considering the thermal-safety as a part of the test cost. However, one of the proposed methods in [5] does not react to temperature deviations, and the other does not handle intra-chip process variation and rapid temperature error changes. In this paper an adaptive test scheduling method is introduced which navigates the tests according to the intra-chip process variation and temporal deviations in temperature errors. It makes use of multiple on-chip temperature sensors to provide on-line intra-chip temperature information.

A dynamic thermal-aware test scheduling technique using on-chip temperature sensors is proposed in [6] in order to cope with the power/temperature simulation inaccuracies in static scheduling. Thermal simulations are performed during the test in order to enable the earliest thermal-safe start of the next test [6]. This method does not handle the process variation and besides, it requires excessive ATE resources to run the thermal simulation during test. In this paper, we introduce a method to address process variation with ATE resources as a constraint.

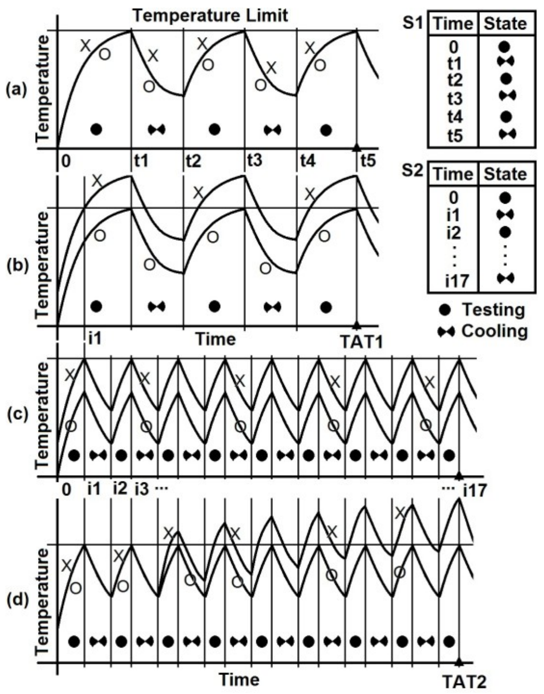
The proposed method in this paper generates a near optimal schedule tree at design time (offline-phase). During testing (online-phase), each chip traverses the schedule tree depending on the actual temperatures. The schedule indicates when a core is testing and when it is in the cooling state. The order of the test sequences is untouched and the schedule tree occupies a relatively small memory. Traversing the schedule tree requires a very small delay overhead for jumping from one point in the schedule tables to another point. This way, the complexity for the online-phase is substantially reduced. To our knowledge, this paper is the first work to present an approach which incorporates the on-chip temperature sensors data, repetitively during test, in order to adapt to the temperature deviations caused by process variation and to achieve a superior test performance.

The rest of the paper is organized as follows. A motivational example is given in section II. Section III provides the problem formulation and then it introduces the cost function. Section IV describes the temperature error model. The linear schedule tables are discussed in section V. The proposed method is presented in section VI. The proposed heuristics are discussed in details in section VII. Experimental results are represented in section VIII. Section IX presents the conclusion.

II. Motivational Example

Assume that there are two instances, *o* and *x*, from a set of chips manufactured for a given design. When the temperature error is negligible, the temperatures of *o* and *x* are equal and the same offline test schedule S1 is safely used for both of them, Fig. 1(a). Cooling periods for S1 are determined using thermal simulation. The simplest model of process variation only models the time invariant temperature errors. Assume that chip *x* is warmer than expected while chip *o* is normal; the result is overheating of chip *x* as shown in Fig. 1(b). To prevent overheating, a more conservative offline schedule S2 has to be designed considering *x* for both chips as illustrated in Fig. 1(c). S2 will lead to a longer test application time (TAT2 vs. TAT1). For chip *o*, S2 is unnecessarily long, since S1 was a safe schedule for *o*. In case of a set of manufactured chips with large temperature variations, a global thermal-safe offline schedule will be based on the hottest chip in the set. This test schedule will introduce unnecessary cooling time for most of the chips, leading to a very slow test.

We have proposed a technique, in [5], to address the above problem with the help of a chip classification scheme. This scheme consists of several test schedules for different temperature error ranges. After applying a short test sequence for warm up, the actual temperature is sensed and the proper test schedule is selected. Therefore, the hotter chips will test with a slower schedule, while the colder chips will test faster.



S1

Time	State
0	●
t1	⋈
t2	●
t3	⋈
t4	●
t5	⋈

S2

Time	State
0	●
i1	⋈
i2	●
⋮	⋮
i17	⋈

Figure 1. Test schedule examples (curves are only illustrative).

The overheating issue is solved and the test application time is not unnecessarily long. This approach works fine assuming the simplest model of process variation (time invariant temperature error), as shown in Fig. 1(a-c).

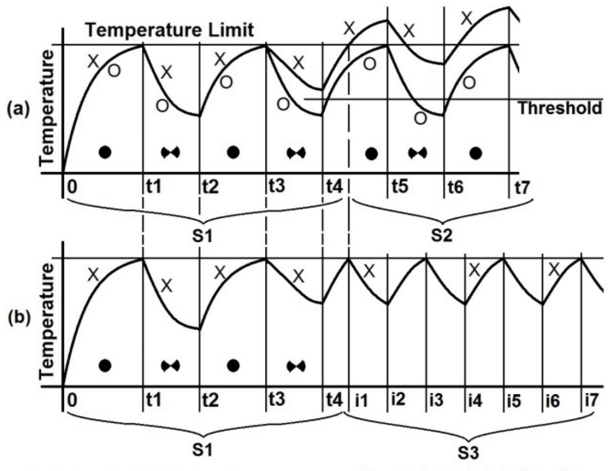
However, in the real world with large process variation, the thermal behavior is time variant and the technique presented in [5] will not be able to achieve high-quality schedules. The variation of thermal response with time is illustrated in Fig. 1(d). In this case, the temperature of chip *x* gradually lifts up compared to chip *o*. A scheduling method capable of capturing temporal deviations is therefore required. The temperature behavior shown in Fig. 1(d) is captured in Fig. 2(a) with more details. The lift up of the temperatures of chip *x* starts at t3, as shown in Fig. 2(a). Since *x* will only overheat after t4, both chips can be safely tested with schedule S1 up to t4. At t4, the actual temperature of the chip under test can be obtained via sensors. The actual temperature can then be compared to a *Threshold* and two different situations can be identified:

$$\begin{cases} x; & if\ Temperature(t4) > Threshold \\ o; & if\ Temperature(t4) \leq Threshold \end{cases}$$

For the rest of the test, after t4, two dedicated schedules, S2 and S3, are generated in the offline-phase for *o* and *x*, respectively. Therefore, in the online-phase the test of *o* continues using schedule S2, as in Fig. 2(a), and the test of *x* continues using schedule S3, as in Fig. 2(b). In this illustrative example, at the end of S1, the schedule does a branching to either S2 or S3 based on the actual temperature. This information and the branching condition can be captured in a branching table, B1 in Fig. 2. The segments of the schedule which are executed sequentially without branching are called linear schedules. An adaptive test schedule consists therefore of a number of branching tables in addition to multiple linear schedule tables.

III. Problem Formulation and Cost Function

Our goal is to generate an optimal adaptive test schedule, offline. The input consists of a SoC design with a set of cores and their corresponding test sequences. The floor plan, the thermal parameters, and the static/dynamic power parameters for the chip are given. The probability distributions that represent the deviations are also given. The desired adaptive schedule minimizes the test application time and overheating.



Linear Schedule Table S1

Time	State	Branch LUT ID
0	●	—
t1	⋈	—
t2	●	—
t3	⋈	—
t4	●	B1

Linear Schedule Table S2

Time	State	Branch LUT ID
t5	⋈	—
t6	●	—
t7	⋈	—

Linear Schedule Table S3

Time	State	Branch LUT ID
i1	⋈	—
i2	●	—
i3	⋈	—
i4	●	—
i5	⋈	—
i6	●	—
i7	⋈	—

Branching Table B1

Condition	Linear Schedule Table ID
Temperature <= Threshold	S2
Temperature > Threshold	S3

● Testing
⋈ Cooling

Figure 2. Schedule and branching tables (curves are only illustrative).

These objectives are encapsulated into a cost function which is introduced later in this section. The desired schedule satisfies the two following constraints. The first constraint is the available test bus width; it limits the number of simultaneously active cores. The second constraint is the available ATE memory which limits the schedule size; indirectly, it also limits the total number of sensor accesses.

In this paper, a comprehensive cost function is introduced by combining the cost of the overheated chips and the cost of the test application. These two contributors to cost go against each other. In order to prevent the overheating for chips with large negative temperature error, more cooling is required. The extra cooling cycles increase the test application time as shown in Fig. 1 (c) in comparison with (a), and lead to underutilization of the test facility. On the other hand, without enough cooling some chips will overheat. The Cost Function (CF) is defined as follows:

$$CF = \mathrm{TAT}/\mathrm{ATS} + BC \times TOP \quad (1)$$

The CF consists of two terms; the first one represents the test application cost and is equal to Test Application Time (TAT) divided by the Applied Test Size (ATS). This term shows what volume of test could be applied by test facility per time unit. The second term represents the overheated chips cost and is equal to Test Overheating Probability (TOP) multiplied with a Balancing Coefficient (BC). The TOP is the number of overheated chips per number of chips entering the test facility. The BC is used in order to balance the cost of the overheated chips against the cost of the test facility. Expensive chips will results in a larger BC and expensive test facility will result in a smaller BC. The exact reasoning behind the CF is not of our interest in this paper and is not explained here.

IV. Temperature Error Model

The temperature error has various sources including process variation, ambient temperature fluctuations, voltage variations, simulator errors, and the temperature dependent errors, e.g. static power (leakage). Temperature error is broadly categorized into spatial error and temporal error. A temperature error model gives the probabilities of the temperature error values for each core (spatial) and for each test cycle (temporal). The spatial temperature error model gives the initial error distribution and the temporal temperature error model is used to recursively estimate the error distribution for the next test cycle.

The spatial temperature error is a discrete statistical distribution, which assigns probabilities to temperature error ranges known as *error clusters*. The temporal temperature error is a discrete-time model, i.e., the temperature error is fixed during a period and then it changes discretely from one period to the next. Therefore, the temporal temperature error model has two pieces of information, the period which is called *temporal error period* and a table of error change probabilities.

For a SoC with as many as *C* cores, the error clusters divide the C-dimensional error space into *error cluster cells* indexed using Cartesian system, i.e. $(l_0, l_1, \dots, l_{C-1})$. For example assume that in a 2-core SoC, each core has 2 *error clusters*. The 2-dimensional error space is divided into 4 *error cluster cells*, indexed with (0, 0), (0, 1), (1, 0), and (1, 1).

V. Linear Schedule Tables

A linear schedule table, as discussed in section II, captures a schedule without branching (offline). The linear schedule table entries (time) should be optimized in the offline-phase. In order to simplify the search space, the possible times are assumed to be multiples of a constant, denoted by *linear scheduling period*. The *states* in the linear schedule tables are generated using the heuristic proposed in [4].

The estimated temperature is updated periodically with *linear scheduling period* by correcting the cores' simulated temperatures with *representative temperature error* value for each core. The estimated temperature is then used to compute the static power and to determine the "state" of the cores for the next *linear scheduling period*. The *representative temperature error* is updated periodically with *temporal error period* while the estimated temperature, static power, and state of the cores are updated periodically with *linear scheduling period*. After updating the state of cores, the dynamic cycle-accurate power sequence for the next *linear scheduling period* is computed. Having dynamic and static power sequences, the next *linear scheduling period* is thermally simulated. A number of linear schedule tables (edges) which are connected using a number of branching tables (nodes) will form the schedule tree, as shown in Fig. 3(a).

VI. Adaptive Test Scheduling

The adaptive method works as follows. During test, the actual temperatures (of carefully selected cores) are read (at carefully selected moments) and the gaps among sensor readouts are filled with thermal simulation. Chips are dynamically classified into one of the *chip clusters* and are tested using the corresponding schedule. At each adaptation moment the *chip clusters* change into a new scheme which is optimum for the new situation. The parameters that affect the

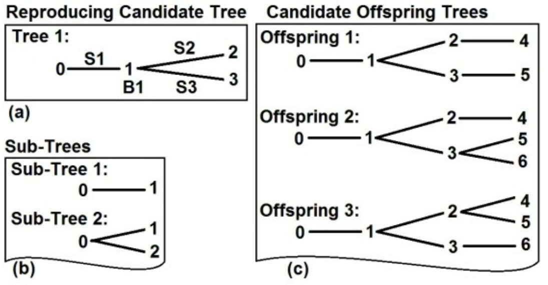


Figure 3. Reproducing tree, sub trees, and candidate offspring trees. For S1, S2, S3, and B1 in (a) refer to Fig. 2.

efficiency of the adaptive method are the moments when branching/adaptation happens, the number of branches (linear schedule tables), and the branching condition (*chip clustering*). For example in Fig. 2, the adaptation is happening at t4, the number of branches is 2 (two linear schedule tables), and the branching condition is a comparison with the *Threshold*. The problem is summarized into the two following sub-problems.

1. How many *chip clusters* (branches or linear schedule tables) at each possible adaptation point (node) are needed? One branch means no branching and no sensor reading.

2. What is the proper *chip clustering* into the given number of *chip clusters*? The number of *chip clusters* is known from question 1. Depending on the *chip clustering* some cores may do not need sensor readout.

When the answer to question 1 is one, question 2 is skipped. These two questions are then formulated into two different forms: the first question is described as a tree topology and the second question is to find the optimum *chip clustering* for the nodes of the specific tree topology.

A candidate schedule tree is generated by putting a possible tree topology together with a possible corresponding *clustering*. Since the number of candidate trees is the product of the tree topology alternatives and the *chip clustering* alternatives, the search space is unacceptably large. In order to reduce the search space, a constructive method is used. The schedule tree is constructed by adding sub-trees (small partial trees) to its leaves. A sub-tree consists of a small number of linear schedule and branching tables which makes it possible to be clustered and optimized (scheduled) at once. For example, assume that there is a reproducing tree, *Tree 1*, as shown in Fig. 3(a). The linear schedule tables of Fig. 2 correspond to the edges of *Tree 1* while the branching table corresponds to node 1, as shown in Fig. 3(a). Two sub-trees with 1 and with 2 edges are shown in Fig. 3(b). *Tree 1* has two leaves, which combinations of sub-trees are added to them in order to generate the offspring as shown in Fig. 3(c). *Offspring 2* is generated by attaching the *Sub-tree 1* to node 2 of *Tree 1* and attaching the *Sub-tree 2* to node 3 of *Tree 1*. The sub-tree scheduling is explained in section A. In section B, it is explained how the trees are constructed and selected.

A. Sub-tree Scheduling

A heuristic is used to find the near optimal sub-tree, by using the partial cost function of sub-tree *clustering* alternatives. When the schedule is a tree, the expected values of *test application time* (TAT), *applied test size* (ATS), and *test overheating probability* (TOP) which are denoted by ETAT, EATS, and ETOP should be used in the CF computation, Eq. (1), in order to utilize the temperature error statistics. The expected values are computed as each edge is being scheduled. The *chip clustering* at each node is done in a C-dimensional space and each *chip cluster* consists of a certain combination of *error cluster cells*. A candidate sub-tree *clustering* is a set of node *clustering* alternatives. For each candidate sub-tree topology there are a number of candidate *clustering* alternatives, which labels the nodes' *error cluster cells* with their corresponding *chip clusters*. Each *chip cluster* for a node corresponds to an edge branching out of that node (equivalent to a linear schedule table). Each node has its dedicated Error Cluster Cell Labeling (ECCL) as follows.

$$ECCL = \{ECCL(n, (l_0, l_1, \dots, l_{C-1}));\, n = 0, 1, \dots, N-1\} \quad (2)$$

Having the *error cluster cell labeling* corresponding to an edge, the edge is scheduled (linear schedule table is determined). The candidate sub-tree *clustering* is evaluated based on the optimized linear schedule tables and optimized branching conditions. A heuristic explores the candidate *clustering* alternatives to find the optimum *clustering*. The

Error Cluster Cell Probabilities of nodes and the nodes probabilities are computed using the temperature error model and based on the *clustering* of the ancestor nodes. This information is then used to compute ETAT, EATS, and ETOP. Since the way *Error Cluster Cell Probabilities* are computed, is not of our interest in this paper, it is not explained here. Having the ETAT, EATS, and ETOP values, the partial cost function is computed. Two different heuristics, a Genetic Algorithm (GA) and a Particle Swarm Optimization (PSO) are used to explore the *clustering* alternatives. The search space is the collection of different alternatives of Eq. (2). For example for a SoC with 2 cores and for a sub-tree similar to *offspring 3* in Fig. 3, an alternative solution is the following. (More details are given in section VII.)

$$\{ECCL(0,(0,0)) = 0, ECCL(0,(0,1)) = 0, ECCL(0,(1,0)) = 1, ECCL(0,(1,1) = 1, ECCL(1,(0,0)) = 1, ECCL(1,(0,1)) = 1, ECCL(1,(1,0)) = 1, ECCL(1,(1,1)) = 0\}$$

B. Tree Construction

The construction starts with a root node and in each iteration the reproducing candidate tree extends and multiplies by adding possible combinations of sub-trees to its active leaf nodes, as shown in Fig. 3. Then, a small number of promising reproducing candidates (similar to Fig. 3(a)) are selected out of the candidate offspring trees (partially shown in Fig. 3(c)). The selection process guarantees the ATE memory constraint and provides the freedom to put more clusters in the more beneficial regions. Such a freedom is provided by the virtue of a Scaled Cost Function (SCF) which is used as the selection criterion. SCF is defined as:

$$SCF = CF \cdot (adjusting\_offset + number\_of\_nodes) \quad (3)$$

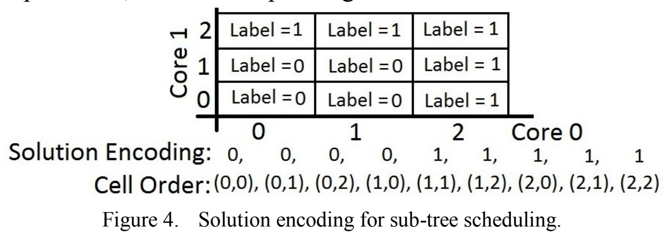
The *cost function* (CF) is scaled by the tree's number of nodes plus *adjusting offset*. Now, adding nodes to the tree is only beneficial if it gives a reasonable cost reduction otherwise a smaller tree may get a lower *scaled cost function* and be selected, while bigger trees are discarded. The effect of the number of nodes is adjusted by *adjusting offset*. A larger *adjusting offset* promotes having more branches, especially near the tree's root.

The number of the sub-tree topologies is controlled with the sub-tree length and the maximum allowed number of branches per node. Increase in the sub-tree length will improve the global optimality and increase in the allowed number of branches per node improves the *chip clustering* resolution, but both will increase the CPU time.

VII. Proposed Heuristics

This work is based on a number of heuristics, broadly categorized into tree construction and sub-tree scheduling. The tree construction is introduced in section VI-B in good details. The explanation of the sub-tree scheduling which is introduced in section VI-A needs more details which are presented here.

As it is introduced in section VI-A, for sub-tree scheduling a possible solution is coded by labeling the temperature *error cells* with a cluster label for each branching node in that sub-tree. An example of solution coding for a single node sub-tree similar to *sub-tree 2* in Fig. 3 (b) is illustrated in Fig. 4. The solution belongs to a 2-core design with 3 temperature *error clusters* per core and the number of branches, i.e. number of chip clusters, in the corresponding sub-tree is 2. As it is shown


Core 1
2
1
0
Label =1
Label =1
Label = 1
Label =0
Label =0
Label = 1
Label =0
Label =0
Label = 1
0
1
2
Core 0
Solution Encoding: 0, 0, 0, 0, 1, 1, 1, 1, 1
Cell Order: (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)
Figure 4. Solution encoding for sub-tree scheduling.

in Fig. 4, the cell order is static and there is no need to include it in every solution vector. (This is also true for the nodes and their order.) An example of solution coding for a sub-tree with more than one branching node, in contrast with the previous example, is given at the end of section VI-A. The possible solutions are then explored with GA or with PSO in order to find a near optimal solution. The implementations of these methods are discussed in the following.

GA mimics the evolution process of a *population*. Each individual member of the *population* is a possible *solution* which is represented by its *chromosome*. Therefore the *chromosome* should represent a solution in a comprehensive and unique way. The *chromosome*, ideally, is a minimal tuple of orthogonal quanta called *gene*. There are three kind of *populations* based on their origins, *elite*, *crossover*, and *mutated*. The *elite population* is initially generated using a simple heuristic and/or randomly. For next iterations, the *elite population* is selected to generate the new population, similar to *natural selection* phenomena. The *crossover population* is generated by mixing two *elite chromosomes* and the *mutated population* is generated by randomly altering some of the *genes* in an *elite chromosome*. The probability that a *gene* commits *mutation* is a characteristic of the GA. This probability is represented by *MutationProbability* in the following. The **genetic algorithm** is presented below as a pseudo code.

```
1. Generate the initial population of elites.
2. Generate crossover population as follows.
   a. Generate scrambled list of elites.
   b. Loop and traverse the scrambled list of elites.
      i. Loop for nodes.
         1. Generate a random crossover point.
         2. Crossover the solution which scrambled list index
            shows with the next solution in the scrambled list.
         3. If a chip cluster is missing go to 2-b-i-1 in order to try
            a new crossover point.
3. Generate mutated population as follows.
   a. Loop and traverse the elite list.
      i. Loop for nodes.
         1. Loop for cells (genes).
            a. Generate a random number smaller than
               (numberOfClusters - 1) / MutationProbability.
            b. If the random number is smaller than the number
               of branches, then copy the random number to this
               gene (cell), otherwise copy it from the current elite
               (given in 3-a).
         2. If a chip cluster is missing go to 3-a-i-1 in order to try
            a new set of random numbers.
4. Evaluate the solutions. This is the most time consuming step.
5. If the termination condition is met, exit with the best elite as final
   solution.
6. Select the new elite population; they are candidates with lower
   costs.
7. GO TO point 2.
```

The PSO mimics the social behavior of a *swarm* searching for food. Each individual member of the *swarm* is called a *particle*. A *particle* is represented by two attributes, its *location* and its *velocity*. The *location* in fact is a *solution* which, usually, is represented by Cartesian coordinate system. The dimensions in the coordinate system are analogous to the *genes* in a *chromosome*. The *velocity* keeps the *particles* moving in the search space. Each *particle* remembers its *previous best* location, and in addition to this individual memory, the *swarm* remembers the best location any of its *particles* have visited before, the *global best*. The *previous bests* and the *global best* are then used to give a hint to the random *velocities*. A canonical form of the PSO uses Eq. (4) to update the *velocity*. The coefficients in Eq. (4) are given as a part of the chosen canonical form [7]. The *random1* and *random2* are two distinct randomly generated numbers between 0 and 1. The *solution* and *velocity* on the right hand side of Eq. (4) are the current values, and the left hand side *velocity* is the next value. Since the *solution*, in this paper, is a natural number, the next *solution* is the rounded sum of the current *solution* and next *velocity*, as represented in Eq. (5). The **particle swarm optimization** heuristic is presented in the following as a pseudo code.

```
velocity = 0.7298*(velocity+
        +2.05*random1*
            *(previousBestSolutions-solution)+
        +2.05*random2*
            *(globalBestSolution-solution))        (4)
      solution = Round(solution+velocity)          (5)
```

1. Generate the initial *swarm*.
2. Generate random initial *velocities*
    a. Limit the range of the random number to the number of *chip clusters* for the corresponding node.
3. Evaluate the solutions. This is the most time consuming step.
4. Find the best solutions as follows.
    a. Loop for all *particles*.
        i. If the current *location* is better than the *previous best location* replace it and check if it is better than the *global best*, if so, replace the *global best*. (For the first iteration, copy the current solution as *previous best*, and find the *global best* among the *previous best* solutions.)
5. If the termination condition is met, exit with the *global best* as final solution.
6. Update the *Swarm* as follows.
    a. Loop for *particles*.
        i. Loop for *cells* (similar to *genes* in GA).
            1. Update the *velocities* according to Eq. (4).
            2. Update the solution (*particle's location*) according to Eq. (5).
            3. Saturate (limit) the solution. It means that if the *location* is outside the valid search space, make it equal to the corresponding extreme and reset the corresponding component of the *velocity* to 0.
7. Check if all clusters exist, as follows.
    a. Loop for all *particles*.
        i. Loop for nodes.
            1. If a *chip cluster* is missing, move the *particle* to its *previous best* location.
8. *GO TO* point *3*.

VIII. Experimental Results

In our experiments, the temperature simulation is done using HotSpot [8]. The static power is computed using the method given in [9]. Other elements of the experimental setup are the same elements used in [5]. Experiments are performed with one SoC build out of ITC'02 benchmark cores. In this section, the different sets of experiments (given in separate tables) are done with different settings and details which are selected in accordance with the requirements of that specific experiment. Separate tables are not meant to be compared.

Scheduling for a SoC, requires a large number of sub-trees to be optimized, however, only some of them are used in the construction of the finally selected tree. Each sub-tree is optimized using a GA (PSO) and as a result, a single SoC test scheduling includes a large number of executions of the GA (PSO). A number of experiments with different heuristics and population sizes are performed and are reported in Table I. The exact settings and details of the algorithms are not of our interest here and are not discussed. In the following tables, ***CF*** is the cost of the schedule as given in Eq. (1) and ***size*** is the ATE memory volume which is required to store the schedule (not the test sequences). The PSO is able to find the best schedule with the lowest cost equal to 5.409 and size equal to 1880. A commonly found schedule with medium quality has a higher cost equal to 5.497. The PSO is able to find it with a population size as small as 5 and with a CPU time as short as 5 hours in contrast with the GA which requires 11 hours.

TABLE I. Comparison of GA with PSO

Population	*Heuristic*	*CF*	*Size*	*CPU Time (H:M:S)*
50	*PSO*	5.497	1800	21 : 43 : 5
30	*GA*	5.497	1800	10 : 54 : 39
20	*PSO*	5.409	1880	11 : 19 : 56
10	*PSO*	5.497	1800	6 : 46 : 57
6	*GA*	5.835	2000	15 : 12 : 37
5	*PSO*	5.497	1800	4 : 54 : 22

TABLE II. Comparison of Traditional and Proposed Method

Methods	**Results**			
	CF	*Size*	*CF*	*Size*
			Relative to the Offline	
Offline	3.3875	460		
Hybrid	2.9389	920	86.76%	200.00%
Adaptive	2.7170	1320	80.21%	286.96%

TABLE III. Experiments with Different Memory Limits

Memory Limit	**Results**				
	CF	*Size*	*CF*	*size*	*CPU Time (H:M:S)*
			Relative to Limit = 500		
300	Aborted, Mem. Limit is too tight				1:03:42
500	3.3875	460	100.00%	100.00%	3:15:21
1000	2.9389	920	86.76%	200.00%	3:41:03
1500	2.7170	1320	80.21%	286.96%	3:53:52
2000	2.7170	1320	80.21%	286.96%	4:04:16

The traditional and the proposed test scheduling methods are compared and the comparison results are reported in Table II. The traditional methods include the Offline method (only one linear schedule is used) and the Hybrid method (similar to [5]). Our proposed adaptive method has reduced the cost to 80% relative to the offline method, while the cost achieved by the hybrid method is 87%. This difference demonstrates the advantage of the proposed adaptive method.

The reduction of the cost with the increase of the memory limit is shown in Table III. It is expected that the increase in the memory limit improves the cost before it saturates at memory limit equal to 1500. The CPU time increases with the increase of memory limit. This trend continues even after cost saturation because the algorithm has larger space to search (for example when memory limit is equal to 2000).

IX. Conclusion

This paper presents an adaptive SoC test scheduling technique in order to deal with spatial and temporal temperature deviations, caused by large process variations. A technique is proposed to generate an efficient test schedule tree, using a number of heuristics. During the test, on-chip temperature sensors are used to monitor the actual temperatures of the different cores and to guide the selection of proper test schedules accordingly. In this way, the overall test cost will be minimized. Experiments are made in order to select the proper heuristics and to tune them. The experiments, also, demonstrate the superiority of the proposed approach over the traditional methods.

References

[1] K.-T. Cheng, S. Dey, M. Rodgers, and K. Roy. "Test challenges for deep sub-micron technologies." pp. 142–149, DAC 2000.

[2] D.R. Bild et Al. "Temperature-aware test scheduling for multiprocessor systems-on-chip." pp.59-66, ICCAD 2008.

[3] C. Yao, K. K. Saluja, P. Ramanathan. "Partition based SoC test scheduling with thermal and power constraints under deep submicron technologies." pp. 281-286, ATS 2009.

[4] Z. He, Z. Peng, P. Eles. "Simulation-driven thermal-safe test time minimization for System-on-Chip." pp. 283-288, ATS 2008.

[5] N. Aghaee, Z. He, Z. Peng, and P. Eles. "Temperature-aware SoC test scheduling considering inter-chip process variation." ATS 2010.

[6] C. Yao, K. K. Saluja, P. Ramanathan. "Thermal-aware test scheduling using on-chip temperature sensors." VLSI Design 2011.

[7] R. Poli, J. Kennedy, and T. Blackwell. "Particle swarm optimization, An overview." Swarm Intell., Vol. 1, No. 1, pp. 33-57, 2007.

[8] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. "Compact thermal modeling for temperature-aware design." DAC 2004.

[9] W. P. Liao, L. He, K. M. Lepak. "Temperature and supply voltage aware performance and power modeling at microarchitecture level." IEEE Trans. CAD, Vol. 24, No. 7, pp. 1042- 1053, 2005.