THE LEFT EDGE ALGORITHM IN BLOCK TEST SCHEDULING UNDER POWER CONSTRAINTS

Valentin Mureșan, Xiaojun Wang
Dublin City University, Ireland
muresanv@eeng.dcu.ie

Valentina Mureșan, Mircea Vlăduțiu
"Politehnica" University of Timișoara, România
vmuresan@cs.utt.ro

ABSTRACT

A left-edge algorithm approach is proposed in this paper to deal with the problem of unequal-length block-test scheduling under power dissipation constraints. An extended tree growing technique is also used in combination with the left-edge algorithm in order to improve the test concurrency under power dissipation limits. Test scheduling examples and experiments are discussed highlighting further research directions toward an efficient system-level test scheduling algorithm.

1. INTRODUCTION

As the device technologies such as VLSI and MCM become mature, and larger and denser memory ICs are called for by the high-performance digital systems, the power dissipation becomes a critical factor and can no longer be ignored either in normal operation of the system or under testing conditions. VLSI devices running in test mode consume more power than when running in normal mode [1]. Thus, one of the major considerations in test scheduling is the fact that the heat dissipated during test application is significantly higher than during normal mode (sometimes 100 - 200 % higher). Test scheduling is strongly related to test concurrency. Test concurrency is a design property which strongly impacts testability and power dissipation. To satisfy high fault coverage goals with reduced test application time under certain power dissipation constraints, the testing of all components on the system should be performed in parallel to the greatest extent possible.

The current paper brings under focus the high-level power-constrained block-test scheduling problem which lacks of practical solutions. An efficient scheme for overlaying the block-tests, called extended tree growing technique is employed successfully together with classical greedy algorithms, e.g. left-edge algorithm, to search for power-constrained block-test schedule profiles in a polynomial time. The algorithm fully exploits test parallelism under power dissipation constraints. This is achieved by overlaying the block-test intervals of compatible subcircuits to test as many of them as possible concurrently so that the maximum accumulated power dissipation does not go over the given limit.

2. TEST SCHEDULING PROBLEM

The components which are required to perform a test (test control logic, test buses, test pattern generators, signature analyzers, blocks under test (BUT), and any intervening logic) are known as test resources and they may be shared among BUT's. Each activity or the ensemble of activities requiring a clock period during the test mode and occurring in the same clock period, can be considered as a test step. A block test is the sequence of test steps that correspond to a specific part of hardware (block). The testing of a VLSI system can be viewed as the execution of a collection of block tests. The steps in a step sequence belonging to the same block test can be pipelined and steps from different block tests can be executed concurrently, obviously if there are no resource conflicts between the steps.

Block tests and test steps have their resource sets used to build up their test plans. Depending on the test design methodology selected, once a resource set is compiled for each test ti, then it is possible to determine whether they could run in parallel without any resource conflict. A pair of tests that cannot be run concurrently is said to be incompatible. Each application of time compatible tests is called a test session, and the time required for a test session is often referred to as test length. Moreover, if PD(ti) is the power dissipation during test ti and PD(tj) is the one during test tj, then the power dissipation of a test session, consisting of just these two tests, is the sum of the instantaneous power dissipation of test ti and tj. These two tests cannot run their tests in parallel (are not compatible) if PD(tj) + PD(ti) > PDmax (the maximal accepted power dissipation). Pi will be the power dissipation considered in the current work as the maximum power dissipation over all test vectors applied in test ti. It is a pessimistic and simplistic definition, but it does not lead to undesirable test schedules which exceed the power dissipation allowance.

3. BLOCK-TEST SCHEDULING

The proposed algorithm deals with tests of blocks of logic, which do not have equal test length. Thus, it is an unequal-length block-test scheduling. It is meant to be part of a system-level block-test approach to be applied on a modular view of a test hierarchy. The modular elements of this hierarchy could be: subsystems, backplanes, boards, MCM's, IC's (dies), macro blocks and RTL transfer blocks. Every test node ti is characterized by a few parameters, which it has previously been assigned with, after the test scheduling optimization has been applied on it. These features are: test application time Ti, power dissipation Pi, and test resource set RESSETi.

4. PROPOSED ALGORITHM

Power dissipation during test scheduling was seldom under research so far. Approaches like [2] tackle the power dissipation problem during test application at gate-level. These approaches...
are not efficient at high levels. A theoretical analysis of this problem at IC level was proposed in [3]. It is, basically, a compatible test clustering, where the compatibility among tests is given by test resource and power dissipation conflicts at the same time. Unfortunately, from an implementation point of view the identification of all cliques in the graph of compatible block tests belongs to the class of NP-complete problems. Instead, a greedy approach is proposed in this paper. It has a polynomial complexity, which is very important for the success of the system-level test scheduling problem. A left-edge algorithm together with a tree growing technique are employed here to generate the block-test schedule profile at the node level, within the test hierarchy. The contribution of this paper is to solve for the first time the application of a polynomial complexity algorithm to the problem of power-constrained test scheduling from the time [3] defined it as belonging to the class of NP-complete problems.

4.1. TREE GROWING TECHNIQUE

In complex VLSI circuit designs, the block-test set is huge and ranges in test lengths. Thus it is possible to schedule some short tests to begin, if they are resource compatible, when subcircuits with shorter testing time have finished testing, while other subcircuits with longer testing time have not. The tree growing technique given in [4] is very productive from this point of view. That is because it is used to exploit the potential of test parallelism by merging and constructing the concurrent testable sets (CTS). This was achieved by means of a binary tree structure (not necessarily complete), called compatibility tree, which was based on the compatibility relations among the tests.

![Figure 1: Merging Step Example](image)

Nevertheless, a big drawback in [4] is that the compatibility tree is a binary one. This limits the number of children test nodes that could be overlapped to the parent test node to only two. In reality the number of children test nodes can be much bigger, as in the examples depicted in figures 1 and 2. Therefore an expanded compatibility tree (ECT), given by means of a generalized tree, is proposed here to overcome this problem. Figure 2 gives the test schedule chart and the ECT for the test scheduling example presented in section 5 and depicted in figure 3(a). The sequence of nodes contained in the same tree path represents an expansion of the CTS. Given a partial schedule chart of a CTS, a test $t$ can be merged in this CTS if and only if there is at least one tree path $P$ in the corresponding compatibility tree of CTS, such that every test contained in the nodes of $P$ is compatible to $t$. The compatibility relation here has three components. Firstly, tests have to be compatible from a conflicting resources point of view. Secondly, the test length of the nodes in a tree path have to be monotonously growing from leaf to root. Thirdly, if power dissipation constraint ($PD_{max}$) is given, the accumulated power dissipation on the above tree path should be less than or equal to $PD_{max}$.

A merging step example is given in figure 1. Partial test schedule charts are given at the top, while partially grown compatibility trees are given at the bottom. Suppose tests $t_2$, $t_3$, and $t_4$ are compatible to $t_1$, while they are not compatible to each other. Suppose $T_1$, $T_2$, $T_3$ and $T_4$ are, respectively, the test lengths of tests $t_1$, $t_2$, $t_3$, and $t_4$, and say $T_2 + T_3 < T_1$. Suppose now, a new test $t_1$ has to be scheduled in parallel to the partial test schedule depicted in figure 1(a). As can be seen, there is a gap $GAP_1$ given by the test length difference: $GAP_1 = T_1 - (T_2 + T_3)$. Thus a merging step can be achieved, if $T_4 \leq GAP_1$, by inserting $t_4$ in the partial test schedule and its associated ECT in figure 1(b).

![Figure 2: Tree Growing Example](image)
4.2. A PARALLEL TO THE LEFT-EDGE ALGORITHM

The high level of similarities between the register allocation problem given in [5] and the node-level test scheduling problem faced by the authors, led to the application of the left-edge algorithm to the block-level test scheduling of a single node within a test hierarchy. The input to the left-edge algorithm given in [5], is a list of variables to be allocated registers. A lifetime interval, with start time and end time, is associated with each variable. The list of variables is sorted on two keys: the start time of the variables as the primary key to sort them in ascending order, and the end time as the secondary key to sort in descending order the variables with the same start time. The algorithm makes several passes over the list of variables until all variables have been assigned to registers.

In the test scheduling algorithm proposed here, block tests take the place of variables, while the test sessions (subsessions) are the registers. The input to our algorithm is a list of block tests to be allocated to different test sessions (subsessions) with the goal to minimize the total test time, keeping the power dissipation within the given limits. The "variables list" in this algorithm has to be a list of block tests sorted by the following two keys: their test application time as the primary key to sort the list in a descending order, and their estimated power dissipation as the secondary key to sort in descending order as well. During each pass over the list, block tests are assigned to test sessions (subsessions) using the tree growing technique and generating other test subsessions in order to obtain better packing density. Throughout the algorithm, the power dissipation accumulated along each test session (subsession) has to comply with the given maximal power dissipation constraint. There are three travel approaches to be followed through the test session list:

1. traveling down the block-test list once for every newly generated gap (hatched or shaded, see subsection 4.1) until no further merging can be performed to it. This uses exactly the list travel approach from the left-edge algorithm. Every newly generated gap is considered a newly "allocated register" and represents a test subsequence;
2. allocating a new test session anytime a test from the list cannot be assigned to the existing gaps (hatched or shaded). Every newly generated gap is considered a newly "allocated register" as well;
3. an intermediate approach would be to consider only the test sessions as "allocated registers". In this case the algorithm travels down the block-test list once for every allocated test session. A new test session is allocated only when there are no more block-tests in the list compatible to the gaps belonging to the current test session.

The complexities of these approaches are $O(n^2)$. For space reasons, only the pseudocode of the first approach is given below in this paper. The data structures used in it are: the Growing Tree to model the ECT, GapsList to model the list of potentially expandable gaps (shaded and hatched gaps). The complexities of these approaches are $O(n^2)$. For space reasons, only the pseudocode of the first approach is given below in this paper. The data structures used in it are: the Growing Tree to model the ECT, GapsList to model the list of potentially expandable gaps (shaded and hatched gaps), BlockTestList to keep the ordered but not yet merged block-tests. CurTest is the block-test to be merged at a certain iteration. CurGap is the gap under focus at a certain iteration to see whether it is expandable (compatible) with the CurTest. In the pseudocode "used" means that the block-test has already been merged in the ECT. RestGap is the newly generated shaded gap at every iteration and it will not be inserted in the GapsList anymore after its generation, if its resulting compatibility list is null, i.e. it will not be an ETP. RestGap is meant to keep the hatched gap generated at every iteration if it is not null, i.e. CurTest covers completely CurGap.

The PSEUDO CODE of the FIRST ALGORITHM:

- Initialize the GrowingTree and the GapsList;
- while there are unscheduled block-tests
  /*BlockTestList is not empty*/
  
  • if(GapsList is not empty) then 
    - CurTest = head of BlockTestList;
    - insert CurTest as the tail of GrowingTree roots (new test section);
    - make CurTest "used";
    - remove CurTest from BlockTestList;
    - generate a TwinGap gap as the twin of CurTest;
    - insert TwinGap into GapsList;
  
  • else 
    - CurTest = head of BlockTestList;
    - CurGap = head of GapsList;
    - while CurGap is the head of GapsList AND the CompList[CurGap] is NOT empty 
      • if(CurTest =< CurGap AND PCCurGap + PCCurTest <= PCM AND CurTest NOT "used") then

Figure 3: Power-Test Scheduling Charts Of The First Approach
generate RestGap = CurTest - CurGap
if RestGap is not null:  
  $T_{RestGap} = T_{CurGap} - T_{CurTest}$
  $PD_{RestGap} = PD_{CurGap}$
  Comp.ListRestGap = Comp.ListCurGap,
  generate TwinGap as the twin gap of CurTest;
  $T_{TwinGap} = T_{CurTest}$
  $PD_{TwinGap} = PD_{CurTest}$
  Comp.ListTwinGap = Comp.ListCurGap
  remove CurGap from the GrowingTree;
insert CurTest and RestGap (if RestGap is not zero) in the place of CurGap inside the GrowingTree;
insert TwinGap into the GrowingTree as the unique offspring of CurTest;
remove CurGap from the GapsList;
insert RestGap (if RestGap is not null) as the head of GapsList;
insert TwinGap as the head of GapsList (if Comp.ListTwinGap is not null);
make CurTest "used";
remove CurTest from BlockTestList; /*if*/
  + else CurTest = CurTest next in the Comp.ListCurGap; /*while*/
  - if (CurGap is still the head of GapsList) then
    /*it means there are no compatible block-tests left for CurGap*/
    /*while*/
  + remove CurGap from the GapsList; /*else*/
/*while*/

5. EXPERIMENTAL RESULTS

The following example provides a graphical view of the working of this algorithm and its results. Suppose the following block-tests are to be scheduled and their parameters ordered in the order: power dissipation, test length and their compatibility list. For simplicity reasons, the block tests are already ordered by test length and power dissipation keys as described in section 4.2.

$$
t_1 (1, 2, \{ t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \})
$$

In figure 3 the results of the first approach are given both with (figure 3(a)) and without (figure 3(b)) power dissipation constraints ($PD_{max} = 15$). Table 1 gives the results of the same algorithm for a 50 block-tests set. Their degree of resource compatibility has been increased within a range from low to high: low (L) 10%, average-low (A-L) 30%, average (AV) 50%, average-high (A-H) 70% and high (H) 90%. The following abbreviations have been used in the table: test length (TL), accumulated power dissipation (APD), the number of iterations of the first loop (1LNb) and the second loop (2LNb) of the same algorithm. It can be seen in figure 3(b) that a tighter power dissipation constraint forces the test scheduling to a more balanced power dissipation throughout the test application time. At the same time obvious power dissipation spikes could be seen in figure 3(a) due to the lack of power dissipation constraints. That means the power dissipation is less balanced when it is loosely constrained. On the other hand when there are tighter power dissipation constraints (see table 1), the total test application time increases. Thus, it turned out to be a trade-off problem to be solved with more complex algorithms.

<table>
<thead>
<tr>
<th>$PD_{max\ LIMIT}$</th>
<th>$PD_{max\ LIMIT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td>TL</td>
<td>APD</td>
</tr>
<tr>
<td>40</td>
<td>139</td>
</tr>
<tr>
<td>45</td>
<td>186</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
</tr>
<tr>
<td>55</td>
<td>200</td>
</tr>
<tr>
<td>60</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 1: Power-Test Scheduling Results Of The First Approach

6. CONCLUSIONS

This novel greedy unequal-length block-test scheduling approach is based on the classical left-edge algorithm applied to an extended tree growing technique and its polynomial complexity is beneficial to the system-level test scheduling problem. Even though it does not guarantee optimal block-test scheduling solutions, its final result can be used as a starting point by near-optimal block-test scheduling approaches (e.g., simulated annealing, genetic algorithms, tabu search) to get an improved solution.

7. REFERENCES