A Generic Resource Distribution and Test Scheduling Scheme for Embedded Core-Based SoCs

Dan Zhao, *IEEE Student Member* and Shambhu Upadhyaya, *IEEE Senior Member*

***Abstract*— We present a novel test scheduling algorithm for embedded core-based SoCs based on a graph-theoretic formulation. Given a system integrated with a set of cores and a set of test resources, we select a test for each core from a set of alternative test sets, and schedule it in a way to evenly balance the resource usage, and ultimately reduce the test application time. Improvements to the basic algorithm are sought by grouping the cores and assigning higher priorities to those with smaller number of alternate test sets. The algorithm is also extended for solving the general test scheduling problem where multiple test sets are selected for each core from a set of alternatives to facilitate the testing for various fault models. A simulation study is performed to quantify the performance of the proposed scheduling approach.**

***Index Terms*— BIST, DFT, IDDQ, Resource balancing, system-on-a-chip test scheduling, test sets selection.**

I. Introduction

THE system level integration is evolving as a new style of system design, where an entire system is built on a single chip using pre-designed, pre-verified complex logic blocks called embedded cores. The system designers or integrators may use the cores which cover a wide range of functions from CPU to SRAM to DSP to analog, and integrate them into a system on a single chip (SoC) with their own user-defined-logics (UDLs). The SoC technology has shown great advantages in shortening time-to-market of a new system and meeting various requirements such as performance, size and cost of today's electronic products.

However, testing such core-based SoCs poses a major challenge for system integrators, as they may have limited knowledge of the cores due to IP (intellectual property) protection. On the other hand, various testing methods such as BIST, scan, functional and IDDQ for many kinds of design environments are provided by different core vendors. Future SoCs will see several hundreds of embedded components in a single package [1]. As a result, it increases system complexity and test cost in terms of test application time. Therefore, SoC manufacturing test becomes a bottleneck in SoC design cycle.

The overall test time of a testing scheme is defined as the period from the start time of test activity to the end time when the *last* test task finishes. Note that, only when all test sets in parallel test queues finish their tasks, we say it is the end of test. The test time may be reduced by using shorter test vectors or better scheduling schemes. Given a set of test sets and test resources, the goal of test scheduling is to schedule the tests in parallel so that those nonconflicting tests (which do not share the same test resource) can be executed concurrently, and thus to reduce the total test time for an SoC.

In this paper, we formulate the test scheduling problem for embedded core-based SoCs as a shortest path problem. We first consider a system where one test set needs to be selected for each core from a group of alternative test sets using different test resources, and propose a novel test scheduling algorithm to reduce the overall testing time. Then, we extend the algorithm to support multiple test sets selection for each core. The basic idea is to effectively construct a shortest path going through each core exactly once, while simultaneously balancing the parallel resource usage. Our major technical contributions are as follows:

(1) Formulation of test scheduling problem for SoCs as the single-pair shortest path problem by representing vertices as test sets, directed edges between vertices as a segment of a schedule sequence, and the edge weight as the test time of the test set at the end of the segment. Thereby, the problem of minimizing overall test time of a schedule becomes equivalent to the problem of finding a shortest path.

(2) Handling constrained scheduling by parallel resource usage queues. Resource conflict is the most commonly addressed constraint during scheduling, which arises due to the same DfT hardware shared among several cores. In addition, certain fault coverage should be achieved when testing an SoC. One method or a combination of several methods may be needed to test a core in order to attain the required fault coverage. In this work, we define m queues in parallel corresponding to m resources, thus the test sets competing for the same resource will sequentially enter the resource usage queue.

The rest of this paper is organized as follows. In Sec. II, we discuss the existing scheduling schemes and the motivation behind SoC modeling and scheduling. Sec. III describes a general SoC model, in which each core may have multiple test sets using different resources. In Sec. IV, we formulate the test scheduling problem as the shortest path problem and propose a novel scheduling scheme based on effective balancing of resource usage. Furthermore, we propose a grouping scheme and all-permutation scheduling to further reduce the overall test time. In Sec. V, we show the experiments based on randomly generated systems and compare the results with other scheduling approaches. Sec. VI extends the algorithm by selecting multiple test sets for each core. Finally, Sec. VII concludes the paper and presents the future work.

Manuscript received December 7, 2002; revised July 18, 2003. This work was supported in part by a NYSTAR grant from Microelectronics Design Center, University of Rochester. The paper was published in parts at IEEE GLSVLSI 2002 and IEEE ISCAS 2003.

D. Zhao and S. Upadhyaya are with Department of Computer Science and Engineering, State University of New York at Buffalo. Please send comments to {danzhao,shambhu}@cse.buffalo.edu.

II. Background

A. Related Work

A number of approaches have been proposed recently for test scheduling. Chakrabarty has shown in [2] that the test scheduling decision problem for SoCs is equivalent to m-processor open shop scheduling, which is known to be NP-complete. In this model, m test resources (such as external test bus and BIST) corresponds to m processors, while each test for a core using a resource is viewed as a task for a job running on a processor. The finish time of a schedule is the latest complete time of individual processor schedule. Mixed Integer Linear Programming (MILP) technique is used to minimize the finish time. However, the computation time of MILP grows exponentially with the number of cores and test resources, and makes this approach unscalable to large systems. Chou et al. [3] have analyzed the test scheduling problem with resource conflicts and power constraints using graph theory. Clique identification and covering table minimization technique are applied on the Test Compatibility Graph (TCG). However, this work is limited to a theoretical analysis rather than proposing an algorithm to solve it. Huang et al. [4] have transformed the problem to bin-packing and adopted a heuristic Best-fit algorithm to map the pins of embedded cores to SoC I/O pins. Iyengar et al. have advanced the rectangle packing approach in [5] to design wrapper scan chains and configure test access mechanism (TAM) buses at the same time. However, these approaches mainly focus on test access architecture configuration and test compatibility is not effectively utilized. Larsson and Peng have proposed a test parallelization combining scheduling scheme to minimize test time under power limitation [6]. But the problem is quite simplified by the assumption of linear dependence of test time and power on scan chain subdivision.

The readers may also refer to [7]–[12] for other scheduling algorithms.

B. Rationale

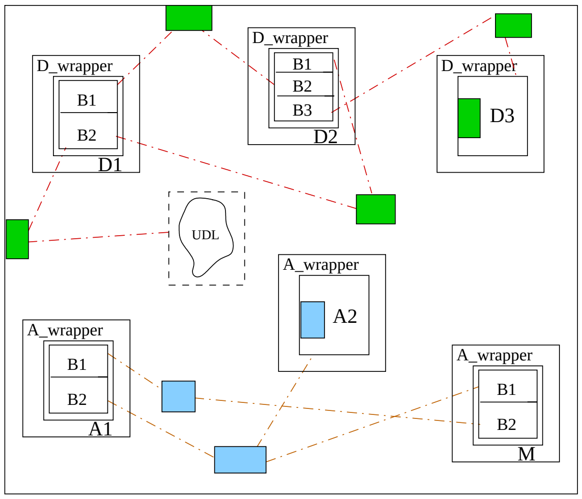
Most of the existing approaches assume that all of the given test sets have to be used in testing. Although test scheduling with multiple test sets has been introduced in [8] and an MILP model has been developed in [2], their work focuses on selecting a test set for each core from a set of alternatives with a varying proportion of BIST and external test patterns, which is just a special case of the problem to be studied in this approach. We assume that a core may be provided with several test configurations, each corresponding to a test set, or a core may consist of several functional blocks or submodules, each of them requiring a different test method in order to meet the fault coverage requirement. For example, a core may be provided by core vendors with several precomputed test patterns to provide flexibility for different system needs. Moreover, as system integrators can purchase cores from various core vendors, multiple core vendors may provide cores with similar functionality but different test configuration. In this case, we may consider the test sets for a core with similar functionality as a group of candidates (i.e., alternate tests). The system integrators need to select one from each group for their system. To our knowledge, this is the first paper to address such an SoC testing problem.

In this paper, we propose a scheduling algorithm for the case where only one from a group of test sets may be selected for each core to perform testing, and take into consideration the test conflicts and the fault coverage requirements. Our method subsumes the problem of constrained scheduling, where some tests may not be executed concurrently due to resource conflicts. In those existing approaches, for example, the MILP formulation, additional constraints have to be added to formulate the problem thus increasing complexity. However, we map the test resources into parallel queues, and the nodes competing for the same resource will sequentially enter the particular resource queue, thus no additional constraints are necessary. The goal behind our formulation is that we expect to minimize the overall testing time by shortening the usage time for any particular test resource. Thus we view test resources as queues and the test to be scheduled as the job entering corresponding queue. The test scheduling problem is deduced to minimizing the longest queue length which represents the overall testing time. In order to solve this problem, we formulate it as a single-pair shortest path problem by representing vertices as test sets, directed edges between vertices as a segment of a schedule sequence, and the edge weight as the test time of the test set at the end of the segment. Thereby, the original problem becomes finding a shortest path from the source to the destination by going through each core exactly once.

III. Preliminaries

A. SoC Modeling

A general SoC model is shown in Figure 1, which consists of digital cores (D_1, D_2 and D_3, for example), analog cores (A_1 and A_2) and mixed-signal cores (M) as well as UDLs which can be treated as cores so as to unify the formulation. In order to facilitate test reuse, a test access architecture, which consists of test wrappers, test access mechanism (TAM) and test source and sink, is constructed for individual cores embedded in the SoC so that the tests can be applied and the responses can be observed at the chip level. The wrappers are logic structures that surround the cores to support both core isolation and test access to IP cores during test operation. The TAM works as “test data highway” which propagates test patterns from the test pattern source to the core-under-test (CUT) and test responses from the CUT to the test pattern sink, as well as the control signals to perform system chip test in a predetermined schedule. In addition, each core may include several functional modules, and each block may be tested by one or multiple test sets using one or multiple resources, thus to provide flexibility for test scheduling. As we can see, if analog, digital and mixed-signal cores do not share resources (for example, the mixed-signal tests must be executed on a special mixed signal test bus like IEEE 1149.4 test bus), they can be separated and tested in parallel using the same scheduling technique, as shown in Figure 2.



r1 − r8 are different test resources
D1 − D3 are digital cores
A1 − A2 are analog cores
M is a mixed−signal core
B_i is denoted as different functional blocks in a core

Fig. 1. A General SoC Model.

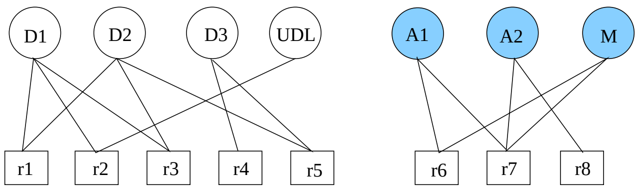


Fig. 2. Graph Representation of Resource Sharing.

B. System Definition and Assumptions

Before introducing the system definition, we list the assumptions made for this work.

(1) It is assumed that an SoC is embedded with n testable cores with m test resources. A core may need to perform one test (or several tests) by using one resource (or several resources) to meet the required fault coverage.

(2) Test resources are defined as test buses, BIST, or any specific set of circuit blocks for certain test configuration. For instance, the circuit blocks, i.e., the test control logic, TPG, compressors/analyzers, and any intervening logic, needed to execute BIST test on core c_i are grouped as test resource r_j for c_i.

(3) A collision occurs when the tests sharing the same resource or the tests for the same core are performed in parallel. Therefore, a core can be tested by one test set by using certain resource at one time. A resource can be shared among several cores, but only one resource can be used by a given core at a given time.

(4) Each test set includes a set of test vectors. Different test sets may have different test times by using different test resources. In other words, core vendors may have provided a set of alternative tests, and one test from each group needs to be performed to achieve the required fault coverage.

Given the test times and the required fault coverage, the goal of the scheduling technique is to efficiently determine the start times of the test sets to minimize the total test application time.

T_{ij}	r_1	r_2	r_3	· · · · · ·	r_m
c_1	T_{11}	T_{12}	n/a	· · · · · ·	n/a
c_2	n/a	T_{22}	n/a	· · · · · ·	T_{2m}
c_3	T_{31}	n/a	n/a	· · · · · ·	n/a
c_4	n/a	n/a	T_{43}	· · · · · ·	n/a
·	·	·	·	·	·
·	·	·	·	·	·
·	·	·	·	· · · · · ·	·
·	·	·	·	·	·
·	·	·	·	·	·
c_n	T_{n1}	T_{n2}	T_{n3}	· · · · · ·	T_{nm}

Fig. 3. Matrix Representation of Test Sets.

Formally, we define the SoC model as *TM*={*C*, *RSC*, *T*, *FC*}, in which *C*=$\{c_1, c_2, ..., c_n\}$ is a finite set of cores, *RSC*=$\{r_1, r_2, ..., r_m\}$ is a finite set of resources, *FC* is the fault coverage required to test each core, and *T*=$\{T_{11}, T_{12}, ..., T_{1m}, ..., T_{n1}, T_{n2}, ..., T_{nm}\}$ is a finite set of tests, which is shown as an $n \times m$ matrix in Figure 3. Test set T_{ij} represents a test set for testing core c_i by using resource r_j, and has a test time of t_{ij}. The entries with n/a indicate that such test sets are not available.

IV. The Proposed Test Scheduling Algorithm

We introduce a new scheduling algorithm for SoC testing. The basic idea of the proposed approach is to map the test sets to a directed graph with weighted edges, and apply the shortest path algorithm to obtain the best testing scheme.

A. Problem Definition

We consider a system discussed in Sec. III-B and assume that one core needs only one test set (selected from a number of candidates) to achieve the required fault coverage (however, this assumption will be nullified in the extended approach to be discussed in Sec. VI). According to the matrix shown in Figure 3, we can construct a graph with $m \times n$ vertices, one for each entry in the matrix (see Figure 4).

- If an entry is n/a (which indicates that the test is not available), it is mapped to a dummy vertex (see the shaded circles in Figure 4).
- A vertex T_{ij} (i representing the core index and j representing the resource index) is connected to all vertices (except the dummy ones) in the next column (i.e., $T_{i+1,k}$, $1 \leq k \leq m$) with directed weighted edges.

Definition 1: *The weight of an edge connecting vertices T_{uv} and T_{ij}* is defined as a vector, $w(T_{uv}, T_{ij}) = (0, ..., t_{ij}, ..., 0)$ (only the jth entry, corresponding to resource r_j, has a value of the test time for T_{ij}, while other entries are zeros). The major motivation behind using the weight assignment is to allow the shortest path algorithm to consider, and moreover, balance the usage of test resources.

- In addition, two special nodes, the source s and the destination d are added. Node s connects the vertices T_{1k} ($1 \leq k \leq m$) with the weight of $w(s, T_{1k}) = (0, ..., t_{1k}, ..., 0)$. The vertices T_{nk} ($1 \leq k \leq m$) connect to node d with a weight of (0,0,...,0).

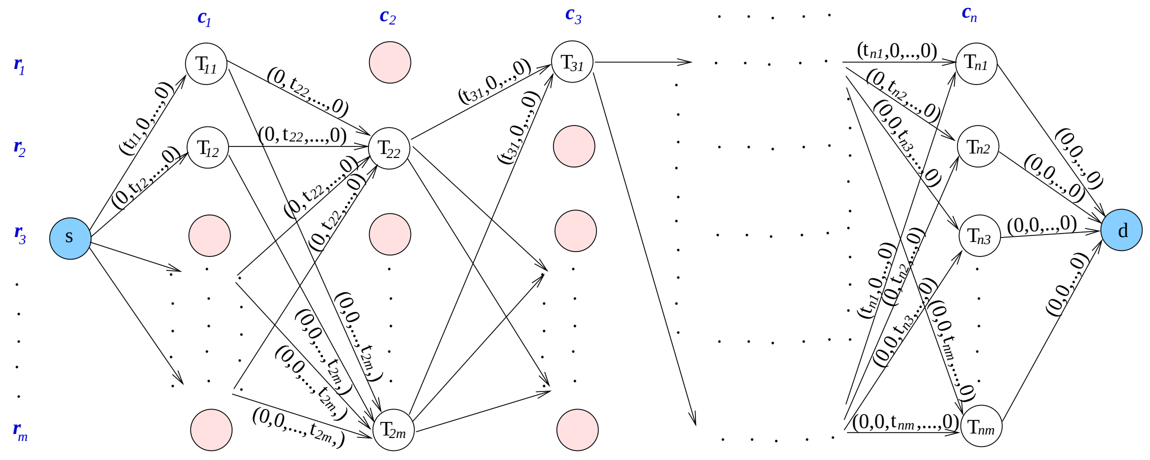


Fig. 4. The Graph Constructed From the $n \times m$ Matrix.

Definition 2: For a path p from a vertex $T_{i+1,j}$ to s including the vertices $T_{1k_1}, T_{2k_2}, ..., T_{ik_i}$ ($k_1, ..., k_i$ can be any value between 1 to m), *the length of the path* $p = (s, T_{1k_1}, T_{2k_2}, ..., T_{ik_i},\ T_{i+1,j})$ is denoted by the distance vector $W(p : T_{i+1,j} \to s) = (D_1^p, D_2^p, ...D_k^p, ..., D_m^p)$, where $D_k^p = \sum_{i=1}^{n} t_{ik}$ is the sum of the test time shown in the kth entry of the weights of all edges along the path p. In addition, *the predecessor of* $T_{i+1,j}$ *on the path* p *to* s is recorded in $\tau_{i+1,j}$.

We define m queues in parallel corresponding to m resources that may be used at the same time independently [13] (see Figure 5). The length of the queue denotes the total testing time of all test sets using the resource. For example, as we can see from Figure 4, the nodes on the first row enter resource queue r_1 depending on whether the path is going through them, and their weights contribute to D_1^p of $W(p : s \to d)$. Since the longest queue length dominates the overall test time of a schedule, the absolute value of the path distance $|W(p : s \to d)|$ is defined as $\max\{D_1^p, D_2^p, ...D_k^p, ..., D_m^p\}$. Accordingly, the test scheduling problem can be converted to the problem of finding a shortest path from s to d. More specifically, the SoC test scheduling problem $STS([T_{ij}], s, d)$ can be formulated as follows.

$\underline{STS([T_{ij}], s, d)}$: Given an SoC represented by an $n \times m$ matrix, construct a weighted, directed graph $G(V, E)$ (where V includes $m \times n$ vertices), with m-tuple weight function $w(T_{uv}, T_{ij}) = (0, ..., t_{ij}, ..., 0)$ for some edges. The length of a path p is the sum of the weights on corresponding resource tuples of its constituent edges. We define the shortest-path weight from source s to destination d by

$$\delta(s, d) = \begin{cases} \min\{|W(p : s \to d)|\}, & if\ exists\ path\ s \overset{p}{\leadsto} d \\ \infty, & otherwise \end{cases} \tag{1}$$

The objective is to find a path p from source s to destination d such that $W(p : s \to d) = \delta(s, d)$.

B. The Schedule With Modified Single-Pair Shortest-Path (SPSP) Algorithm

Dijkstra's algorithm [14] is a well-known approach to solve the single-source shortest path problem when all edges have nonnegative weights. A variation of this algorithm can be used to find the shortest path from s to d of the graph shown in Figure 4. More specifically, each vertex T_{ij} maintains a m-tuple vector as the distance to the source s, $W(p : T_{ij} \to s) = (D_1^p, D_2^p, ..., D_m^p)$, and τ_{ij} to record its predecessor on the shortest path to s. Each vertex may be in one of the following three states:

- **state 1**: not updated; the distance vector is $(\infty, \infty, ..., \infty)$.
- **state 2**: updated; the distance vector has been updated at least once.
- **state 3**: finalized; the distance vector is the shortest distance to s, and it will not be updated in the future.

Initially, s is finalized (i.e., in state 3), with the distance vector $W(p : s \to s) = (0, 0, ..., 0)$, and all other vertices are not updated (i.e., in state 1), with the distance vector $W(p : T_{ij} \to s) = (\infty, \infty, ..., \infty)$. s will update the distance vectors of all its neighbors (T_{1k}, $1 \le k \le m$, in Figure 4). More specifically, $W(p : T_{1k} \to s) = w(s, T_{1k})$, $\tau_{1k} = s$, and the states of these vertices will be changed to *state 2*. Then, one of the vertices in state 2 with the smallest $|W(p : T_{1k} \to s)|$ will be selected and finalized. Again, it will update the distance vector of all of its neighbors. In general, when a vertex T_{ij} (with the smallest $|W(p : T_{ij} \to s)|$ among all vertices in state 2) is finalized, it will update the vertices $T_{i+1,k}$ ($1 \le k \le m$). If $W(p : T_{i+1,k} \to s) > W(p : T_{ij} \to s) + w(T_{ij}, T_{i+1,k})$, then $W(p : T_{i+1,k} \to s) = W(p : T_{ij} \to s) + w(T_{ij}, T_{i+1,k})$, and $\tau_{i+1,k} = T_{ij}$. This algorithm will continue until the vertex d is finalized. The pseudocode is provided in Appendix. It can be shown that, the worst-case time complexity of the initialization step is $\Theta(V)$, and the computation is dominated by Priority Queue operation $\Theta(E \log V)$, where V is the number of vertices and E is the number of directed edges in Graph G. Therefore, the worst case time complexity of this algorithm is $\Theta(E \log V) = \Theta(m^2 n \log(mn))$, where m is the number of resources and n is the number of cores in the system, respectively.

Figure 6 shows an example of applying the algorithm for a core-based system with 7 cores and 4 resources (as shown in Table I). We first construct a graph (see Figure 6(a)) as described in Sec. IV-A, then we apply the modified SPSP algorithm to find the shortest path from s to d (see Figure 6(b)).

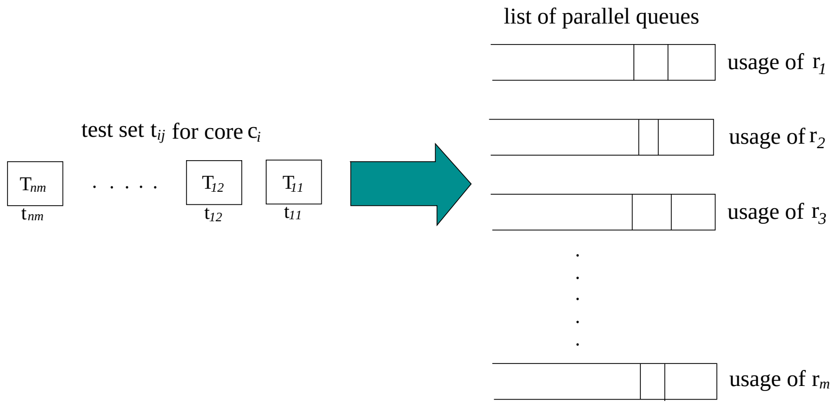


Fig. 5. Parallel Usage of Test Resources.

TABLE I
THE MATRIX OF TEST SETS FOR AN EXAMPLE SYSTEM

T_{ij}	r_1	r_2	r_3	r_4	P
c_1	12	7	n/a	6	3
c_2	n/a	4	1	n/a	2
c_3	3	n/a	8	12	3
c_4	n/a	n/a	13	9	2
c_5	5	7	6	2	4
c_6	5	n/a	1	n/a	2
c_7	4	6	n/a	n/a	2

As we can see from Figure 6(b), a shortest path from s to d includes tests T_{14}, T_{23}, T_{31}, T_{43}, T_{52}, T_{61} and T_{71}, within which T_{31}, T_{61} and T_{71} are the tests sequentially entering resource queue r_1, T_{23} and T_{43} sequentially enter resource queue r_3, and T_{52} and T_{14} are the only test sets using resource r_2 and r_4, respectively. The test sets in different resource queues can be applied concurrently. The distance vector of d, $W(p : d \rightarrow s) = (12, 7, 14, 6)$, represents the total test time on the corresponding resource queue. As we can see, queue r_3 is the longest resource queue which results in an overall test time of $|W(p : d \rightarrow s)| = 14$. We convert the shortest path from s to d into a way of resource usage of the cores as shown in Figure 7(a). As a matter of fact, the shortest path from s to d is constructed by balancing the resource queue lengths. In addition, as we have noticed, one of the advantages of the proposed approach is that there is no idle time between successive tests (namely, *explicit dead time*) in any of the queues.

Proposition 1: A shortest path from source s to destination d is constructed in a way that balances the resource usage queues.

Proof: As shown in Figure 5, we have defined m queues in parallel corresponding to m resources. Meanwhile, the graph on which the modified shortest path algorithm is applied is constructed in a way that the rows are corresponding to the resource usage queues while the columns are corresponding to the cores in the SoC. In other words, the vertices in column i represent all the tests for core c_i and the vertices in row j are the candidate tests entering resource queue r_j. As we have discussed earlier, each vertex maintains a m-tuple vector as the distance to source s, $W(p : T_{ij} \rightarrow s) = (D_1^p, D_2^p, ..., D_m^p)$. Each vertex T_{ij} in the graph is updated when a shorter longest queue length can be reached when going through vertex $T_{i-1,k}$, i.e., $W(p : T_{ij} \rightarrow s) > W(p : T_{i-1,k} \rightarrow s) + w(T_{i-1,k}, T_{ij})$. Vertex T_{ij} is finalized when its longest queue length $|W(p : T_{ij} \rightarrow s)|$ is the smallest among updated vertices. That means, a shortest path is constructed from s to T_{ij}. Thus, the shortest path from s to d is constructed by balancing the resource queues to minimize the length of the longest queue which dominates the total test time. ■

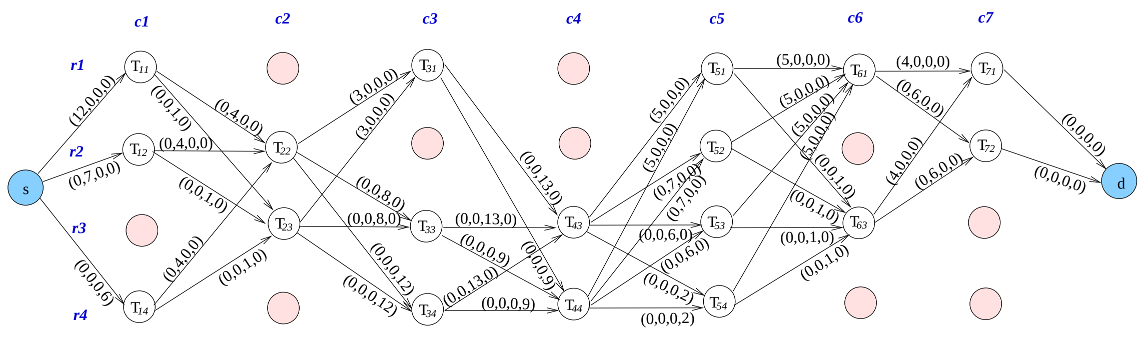
Proposition 2: There is no "explicit dead time" in this resource balancing approach.

Proof: Explicit dead time arises due to resource conflicts. There are two types of resource conflicts defined in our system. 1) Several tests compete to use the same resource; 2) Different tests for the same core are executed at the same time. Conflict of the first kind is totally overcome by the resource balancing approach, since the tests competing for a resource sequentially enter the resource queue. Although for each core a set of tests is provided, only one of them will be executed to test the core. So the conflict of the second kind is eliminated. ■

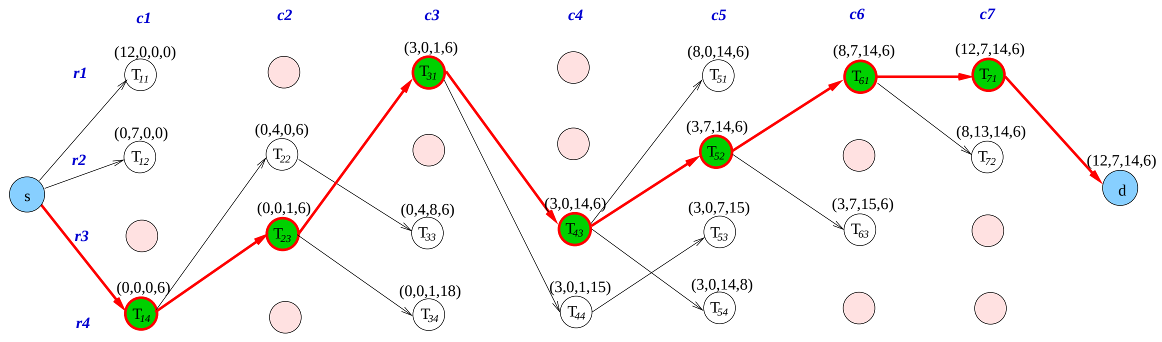
C. Grouping Scheme

As there is no explicit dead time in resource balancing, our purpose is to effectively reduce the *implicit dead time* (i.e., the idle time appearing at the beginning or the end of a schedule) at the end of the resource queues (obviously, no implicit dead time appears at the beginning in this approach). Because the shortest path from s to d is set up by going through certain test set of each core from left to right, different ordering of ready-to-schedule cores (i.e., the cores before entering the resource queues) results in different schedule of tests, and accordingly the total test time.

We group the cores based on the number of available tests they have, such that in a group G_p, all cores have P alternate test sets. This is a one time effort. For example, the right most column of Table I shows the P value of each core. The cores in the group with smaller P value will be scheduled earlier, because these tests have to be put into certain queues (i.e., the corresponding cores have to be tested by using certain



(a) The Graph of The Example System



(b) The Schedule Without Grouping

Fig. 6. The Scheduling With The Modified SPSP Algorithm.

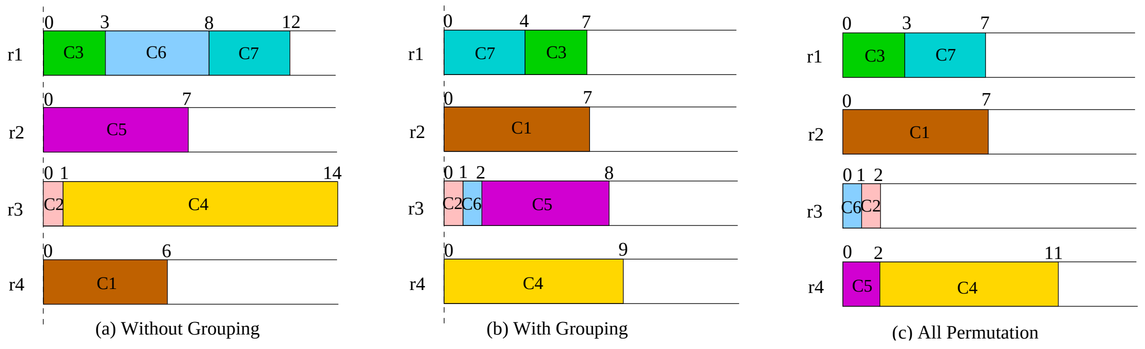


Fig. 7. The Final Schedule Illustrated on Parallel Queues.

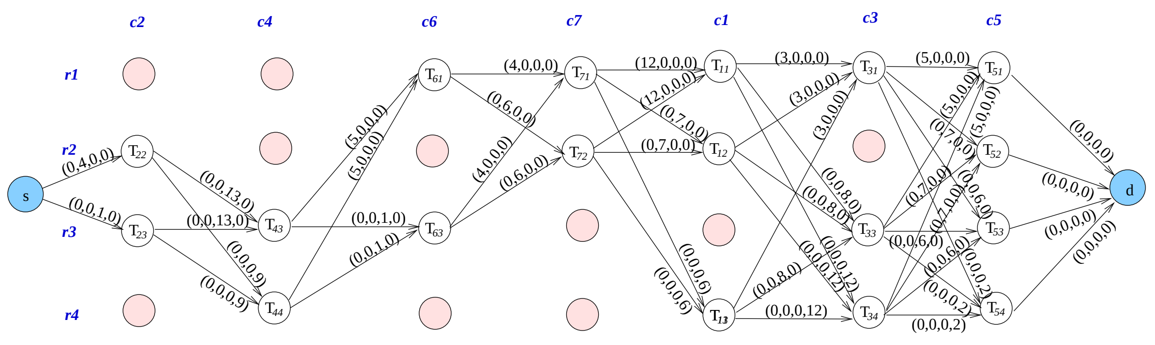
resources). Then, we schedule the test sets in the group with larger P value to balance the lengths of the resource queues, and accordingly shorten the longest queue length. Balancing queue length ultimately results in shorter overall test time.

Figure 8 illustrates the execution of the algorithm with grouping for the given example. Core c_2, c_4, c_6 and c_7 have 2 alternate test sets and are scheduled first. With the P value of 3, c_1 and c_3 are the cores to be scheduled next, and finally c_5 with P of 4 is scheduled. After the graph is constructed, the shortest path algorithm described in the previous subsection can be employed here to find the shortest path. Figure 7(b) shows the resource usage of the final schedule. Compared to the case without grouping, the total test time is reduced by 30% by using the grouping scheme. As we can see, grouping the cores properly before scheduling can reduce the total testing time and achieve better balancing of resource usage, while the worst case time complexity remains the same.

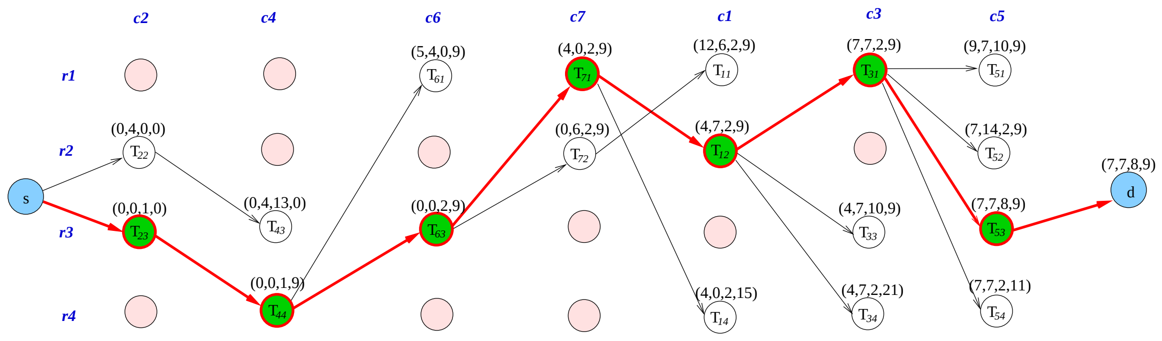
D. *All Permutation Scheduling*

As we have discussed above, different ordering of the cores will affect the performance of the schedule significantly. To perform test scheduling on a system embedded with n cores, there are $n!$ ways for the ordering of the cores. Thus the optimal schedule can be determined after running $n!$ times of the SPSP algorithm on all $n!$ different ordering of cores. Clearly, the computation is quite excessive (the worst case time complexity is $\Theta(n!m^2n\log(mn))$).

One way to reduce computational complexity while eliminating the effect on core ordering is to construct a single graph with all possible permutations of the cores and running the SPSP algorithm once. We call this kind of scheduling as *All-Permutation Scheduling*. In this model, the graph is constructed with the size of $m \times n \times n$. We list all the tests (including the dummy nodes) for the cores in one column and copy by n times, thus there are $n!$ possible ways for the



(a) The Graph After Rearranging The Cores



(b) The Schedule With Grouping

Fig. 8. The Scheduling With Grouping Scheme.

TABLE II
THE TEST SETS FOR ALL PERMUTATION SCHEDULING

T_{ij}	r_1	r_2	r_3	r_4
c_1	T_{11}	T_{12}	n/a	T_{14}
c_2	n/a	T_{22}	T_{23}	n/a
c_3	T_{31}	n/a	T_{33}	T_{34}

ordering of these cores. Note that, when we connect a vertex in column i ($1 \leq i \leq n-1$) to a vertex in column $(i+1)$, they should not belong to the same core. We use a simple example for illustration, which includes 3 cores and 4 resources as shown in Table II. We first construct the graph as shown in Figure 9. In this way, we consider all $3! = 6$ permutations of the three cores in one graph: (1) c_1, c_2, c_3; (2) c_1, c_3, c_2; (3) c_2, c_1, c_3; (4) c_2, c_3, c_1; (5) c_3, c_1, c_2; (6) c_3, c_2, c_1. With the graph ready, the shortest path algorithm discussed in Sec. IV-B can be employed on this graph with a minor modification that a vertex T_{ij} cannot update those neighbors in the next column if the cores they belong to have already been included in the shortest path of T_{ij} to s. Therefore, for each vertex T_{ij} we maintain a record that traces the cores which consist of the shortest path from T_{ij} to s. Note that, the all-permutation scheduling doesn't result in an optimal schedule, because when we construct the shortest paths for the vertices to s, some among the $n!$ permutations will not be taken into consideration anymore since those finalized vertices will not contribute to the shortest path from s to d.

We apply the all-permutation scheduling (AP, for short) on the same example used by the schedules with/without grouping (WG and WOG for short, respectively), the schedule result is shown in Figure 7(c) (We don't show the graph of all-permutation scheduling due to its complexity). When we compare the results of the three approaches, an interesting observation is that although AP approach considers all possible permutations of the cores at one time, it doesn't result in a better performance than WG approach. It's because balancing the resource usage queues at the earlier stage does not guarantee the final result to be well balanced. For example, Figure 10 shows the shortest paths constructed by WG and AP approaches on the example system. Although, AP results in more balanced queues from stage 2 to 6 (for instance, when in stage 2, the longest queue length in AP is 2 while in WG 9), it leads to a longer longest queue length (11) than that in WG (9). As we can see, till the last stage, WG balances the queues by inserting tests into other queues than the longest queue. While in AP, it balances the resource queues well at the beginning (till stage 6, the longest queue length in AP is 7), but in the final stage, it cannot result in more balanced queues rather than increasing the length of the queue of r_4 (which results in the longest queue length finally). In addition, all-permutation scheduling results in much higher complexity, $\Theta(E \log V) = \Theta(m^2 n^3 \log(mn^2))$.

V. SIMULATION STUDY

We evaluate the proposed scheduling algorithms by implementing them in C and running simulations on Sun Enterprise 450 Workstation with four 450MHz UltraSPARC-II CPUs. We define the balance ratio as G as given below:

$$G = \frac{L_{wog} - L}{L_{wog}}$$

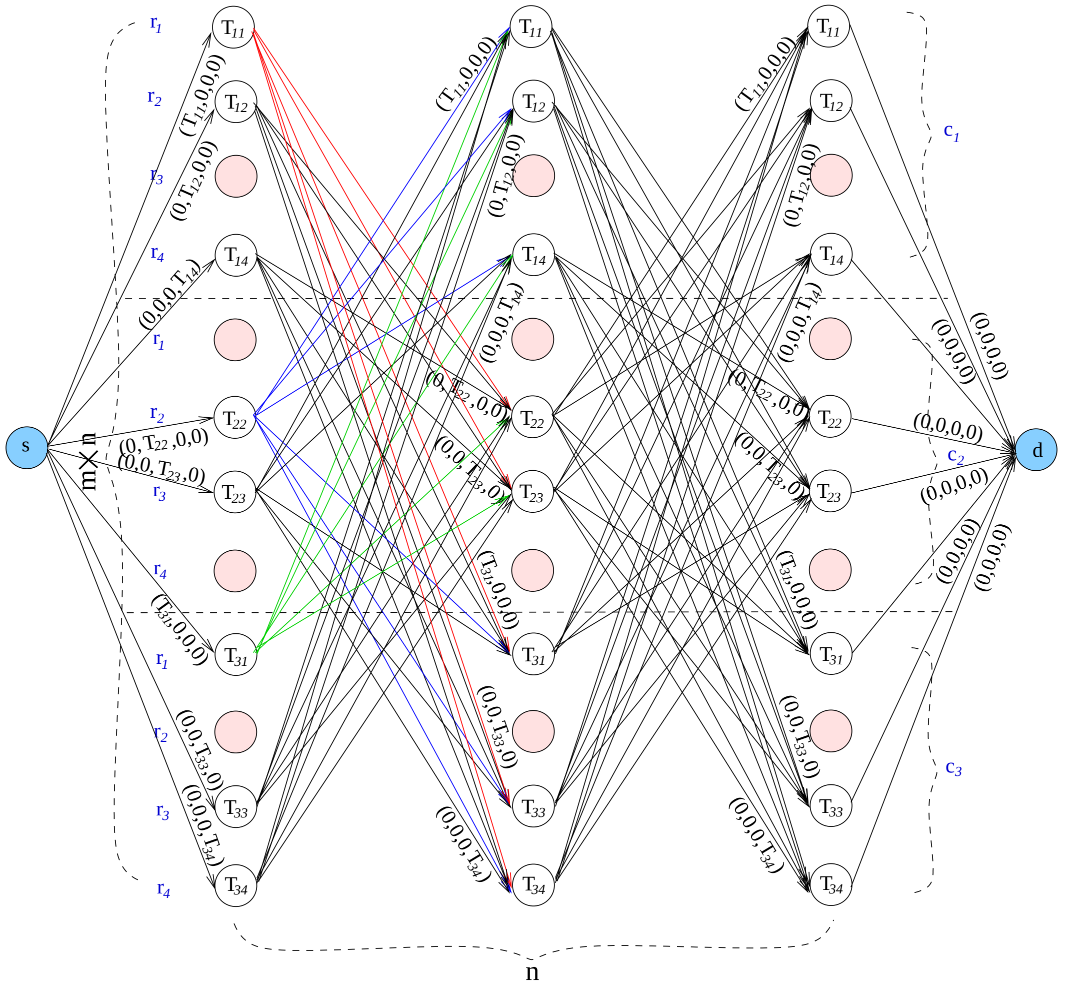


Fig. 9. The Graph Constructed for All Permutation Scheduling.

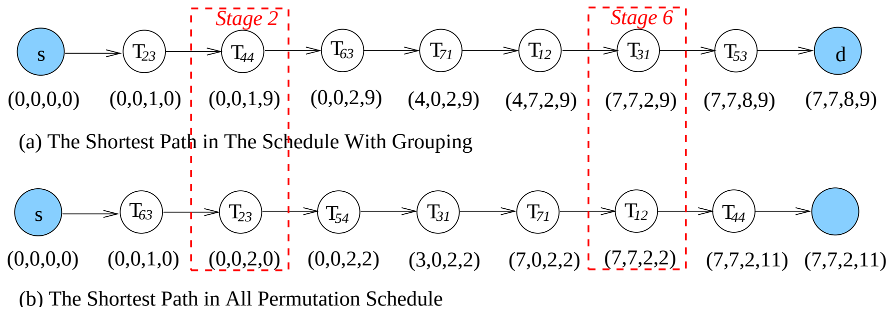


Fig. 10. Comparing The Shortest Paths in The Schedules With WG and AP Approaches.

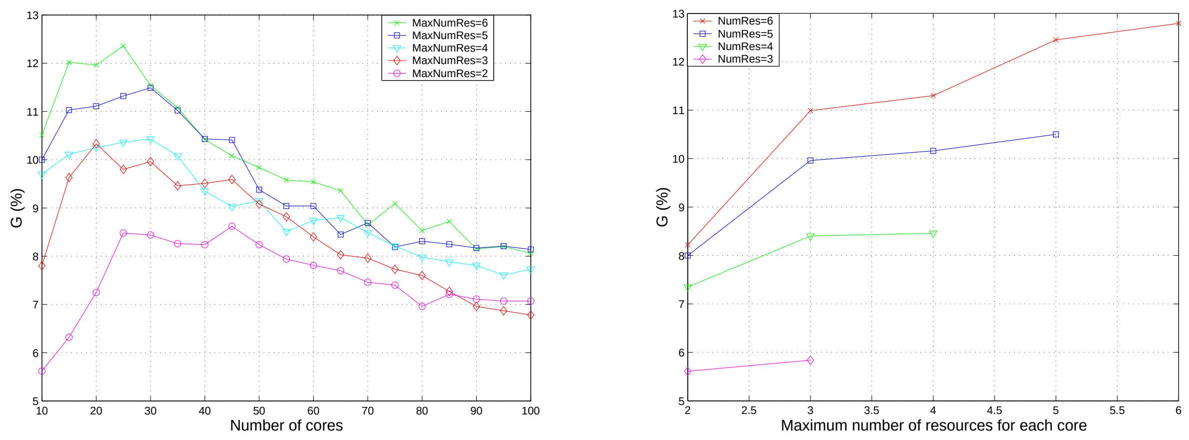
where L_{wog} is the total test time of a schedule without grouping while L can be either L_{wg} or L_{ap}, i.e., the total test time of a schedule with grouping or all-permutation scheduling.

In simulation scenario 1, we assume that there are 5 resources in the system and each core may be provided with 1 to 3 test sets using corresponding resources to meet the fault coverage requirement. Table III shows the comparison results of the performance of WG over WOG, as well as AP over WOG. The number of cores (*NumCore*) in the SoCs changes from 10 to 45. TL in WOG/WG/AP represents the total test time by using WOG/WG/AP approach and the balance ratio of WG to WOG is represented by G_{og} in percentage, while the balance ratio of AP to WOG in G_{oa}. ET in WOG/WG/AP means the corresponding CPU execution time in these approaches, represented in milli seconds. As we can see, WG and AP perform better than WOG since both G_{og} and G_{oa} are greater than zero. When Comparing WG with AP, WG achieves better performance than AP in terms of the balance ratio and the CPU execution time. G_{og} reaches as high as 9.48% when *NumCore* is 40, while G_{oa} is 4.97%. When *NumCore* is 45, AP needs $158427ms$ CPU time to execute the algorithm while WG only needs $5ms$.

In scenario 2, we study the effect of the number of cores

TABLE III
THE COMPARISON BETWEEN WOG, WG AND AP APPROACHES.

# of cores	TL in WOG	TL in WG	TL in AP	G_{og} (%)	G_{oa} (%)	ET (ms) in WOG	ET (ms) in WG	ET (ms) in AP
10	136592.84	126900.89	131228.16	7.10	3.93	0	0	117
15	186448.02	170924.75	179253.49	8.33	3.86	1	1	837
20	224807.08	205515.81	216476.31	8.58	3.71	1	1	3394
25	270352.56	246014.63	258553.52	9.00	4.36	2	2	10005
30	318043.99	290777.63	307681.87	8.57	3.26	3	3	24359
35	365636.25	332672.08	350888.77	9.02	4.03	3	3	51968
40	413141.74	373992.82	392622.08	9.48	4.97	4	4	94022
45	455018.06	417752.12	434622.70	8.19	4.48	5	5	158427



(a) G_{og} Changing With *NumCore*.

(b) G_{og} Changing With *MaxNumRes*.

Fig. 11. G_{og} Changing With The Resource Distribution.

on the test time and the maximum number of resources (*MaxNumRes*) provided for each core on the test time. We first assume that the total number of resources in the system is 6. Figure 11(a) shows the G_{og} values with number of cores ranging from 10 to 100 and maximum number of resources ranging from 2 to 6. As we can see, with the same maximum number of resources, G_{og} increases when the number of cores increases. After it reaches a peak, it drops slowly when the number of cores increases further. For example, when *MaxNumRes* is set at 5, G_{og} increase from 10% when *NumCore* is 10. It reaches a peak of 11.49% when *NumCore* is 30. Then it drops slowly, G_{og} decreases to 8.14% with *NumCore* 100. This is reasonable because, when there are small number of cores, the total number of tests is also small and we could not balance the resource queues more evenly due to less flexibility. As the number of cores increases, the flexibility increases, and accordingly, G_{og} increases. On the other hand, when there are a large number of tests, the benefit of grouping will be dominated by the randomness, which in turn results in the dropping of the curve.

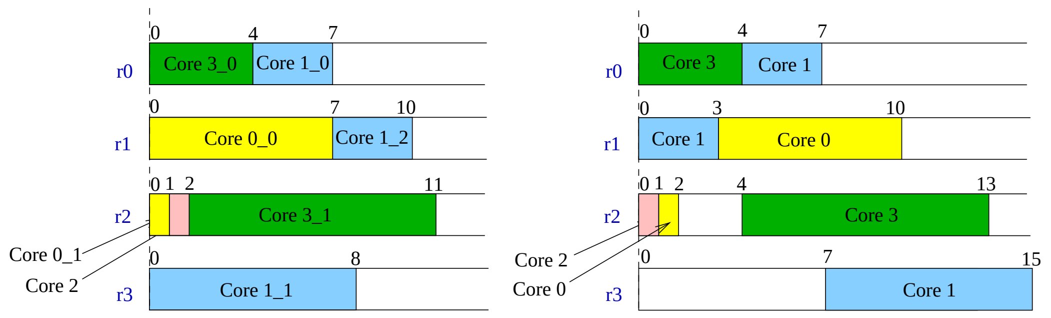
Moreover, we choose the number of cores to be 25 (for example), and change the total number of resources in the system from 3 to 6. Figure 11(b) shows G_{og} with various maximum number of resources for each core. As we can see, with the same total number of resources, G_{og} increases with the maximum number of resources for each core, while with the same maximum number of resources for each core, G_{og} increases when the total number of resources increases. This is again due to the change in flexibility of choosing test resources as discussed above.

Based on our simulation, we have the following result.

Proposition 3: Grouping always helps balance the resource usage queue lengths fast and efficient.

VI. FAULT-MODEL ORIENTED MULTIPLE TEST SETS SCHEDULING

In the previous section, we assumed that one core needs only one test set. However, it is possible that a core may need multiple (say L) test sets to achieve a certain fault coverage. For example, in an embedded core-based SoC, several test methods are used to test the embedded memory. As we know, in addition to stuck-at, bridge, and open faults, memory faults include bit-pattern, transition, and cell-coupling faults. Parametric, timing faults, and sometimes, transistor stuck-on/off faults, address decoder faults, and sense-amp faults are also considered. [15] lists various test methods for embedded memory, i.e., direct access, local boundary scan or wrapper, BIST, ASIC functional test, through on-chip microprocessor,



(a) Test Set Selection for Each Fault Model of Cores

(b) Rescheduling to Avoid Conflicts

Fig. 13. Multiple Test Sets Scheduling.

Core ID	Fault Model	Candidate Test set	Resource Usage
c_0	f_{00}	$t_{00} = 12$	r_0
		$t_{01} = 7$	r_1
		$t_{02} = 6$	r_3
	f_{01}	$t_{03} = 4$	r_1
		$t_{04} = 1$	r_2
c_1	f_{10}	$t_{10} = 3$	r_0
		$t_{11} = 8$	r_2
		$t_{12} = 12$	r_3
	f_{11}	$t_{13} = 13$	r_2
		$t_{14} = 8$	r_3
	f_{12}	$t_{15} = 5$	r_0
		$t_{16} = 3$	r_1
		$t_{17} = 6$	r_2
		$t_{18} = 11$	r_3
c_2	f_{20}	$t_{20} = 5$	r_0
		$t_{21} = 1$	r_2
c_3	f_{30}	$t_{30} = 4$	r_0
		$t_{31} = 6$	r_1
	f_{31}	$t_{32} = 18$	r_1
		$t_{33} = 11$	r_3
		$t_{34} = 9$	r_2

Fig. 12. A Fault Model Based System.

etc. Different test methods may require different test resources, use different test times, and provide different fault coverage. In this case, we can simply make L virtual cores and convert the 1-L mapping to a 1-1 mapping. The only difference between this and the single test selection we discussed earlier is that, when choosing the shortest queue, one has to check if the selected test set conflicts with others which are for the same core and overlap the running time. Figure 12 shows an example system, in which the tests are to be performed using the corresponding resources, for instance, test t_{10} to be applied using resource r_0, test t_{11} using resource r_2, etc. For each fault model, we need to select one test method by applying certain test from the candidates. Figure 13 illustrates the multiple test sets scheduling for the system, which can be performed in two steps.

First, we create L virtual cores for each core corresponding to L fault models. For example, in Figure 12, two virtual cores, $Core$ 0_0 and $Core$ 0_1 are generated for Core c_0 according to the two fault models, f_{00} and f_{01}, respectively. For each fault model, a group of test sets with various test times are provided for the required fault coverage. This means, each virtual core has a group of test sets available and we select one of them to perform testing. For instance, in order to cover fault f_{00}, we need to select one test set from the alternate test sets, t_{00}, t_{01} and t_{02}. Thus we map the multiple tests selection model to the single test selection case. We select the tests in a way that we balance the queues in order to avoid the situation where all the test sets will only use some of the resources and thus result in long length in these queues. In the second step, we need to reschedule the tests for the same core which overlap the running time. The shortest-task-first procedure is adopted here for rescheduling [16]. The worst case complexity is $O(r^3)$, where r is the number of virtual cores.

VII. Conclusion and Future Work

Optimal test scheduling for embedded core-based problem is a NP-hard problem. In this paper, we have formulated the SoC test scheduling to the single-pair shortest path problem, and presented efficient test scheduling heuristic algorithms for embedded core-based SoCs. With the flexibility of selecting a test set from a set of alternatives, we have proposed to schedule the tests for a given system in a way that balances the resource usage queue as evenly as possible, thus reducing the overall test time. Moreover, we have presented a grouping scheme and all permutation scheduling to optimize the schedule and evaluated the proposed approaches via simulation. Our simulation results have shown that there is no explicit dead time in our approach and we can further reduce the implicit dead time by proper grouping. We have also extended the algorithm to allow multiple test sets selection from a set of fault model based alternatives. We expect that the proposed approach can be properly extended for testing the mixed-signal SoCs as well. In our future work, we will discuss the modeling of mixed-signal SoCs for testability analysis, scheduling and diagnosis, and present efficient test scheduling algorithms to minimize the test cost.

APPENDIX
THE PSEUDOCODE OF THE SHORTEST PATH ALGORITHM

```
structure CORE: $id$;            /* core id */
                $num\_t$;        /* the num of test sets */
                $t[m]$;          /* the test time of each test set */
                $r[m]$;          /* the corresponding test resources */
structure NODE: $dist[m]$;       /* a vector of distance to $s$ */
                $wt[m]$;         /* the weight on the incident edge */
                $pred$;          /* the predecessor */
                $max\_dist$;     /* the max among $dist[m]$ */

Modified SPSP (int $m$, int $n$, struct CORE $core[n]$, struct NODE $vertex[mn]$)
begin
/* initialize $V[G]$ */
for each vertex $v \in V[1..mn]$        /* $m$ is the num of resources, $n$ is the num of cores */
    for the distance on each resource queue $j \in r[1..m]$
        $vertex[v].dist[j] = \infty$;
    for the weight of the incident edge on each resource queue $j \in r[1..m]$
        $vertex[v].wt[j] = vertex[v].t[j]$;
    $vertex[v].pred = NIL$;
    $vertex[v].max\_dist = \infty$;
/* initialize source $s$ */
for the distance on each resource queue $j \in r[1..m]$
    $vertex[s].dist[j] = 0$;
$vertex[s].pred = NIL$;
$vertex[s].max\_dist = 0$;
/* initialize destination $d$ */
for the distance on each resource queue $j \in r[1..m]$
    $vertex[d].dist[j] = \infty$;
for the weight of the incident edge on each resource queue $j \in r[1..m]$
    $vertex[d].wt[j] = 0$;
$vertex[s].pred = NIL$;
$vertex[s].max\_dist = \infty$;

$S \leftarrow \{\phi\}$;
Enqueue all vertex $v \in V[G, s, d]$ into priority queue $Q[1..mn + 3]$;

while $Q \neq \phi$
        /* $u \leftarrow$ Extract-Min(Q) */
        Dequeue($u, Q, m, n$);        /* remove node $u$ from $Q$ with minimum $max\_dist$ value */
        $S \leftarrow S \bigcup \{u\}$;
        if (node $u == s$)
            for each node $v \in Adj[s]$
                for the distance on each resource queue $j \in r[1..m]$
                    $Q[v].dist[j] = Q[v].wt[j]$;
                update $Q[v].max\_dist \leftarrow max\{Q[v].dist[j]\}$;
                $Q[v].pred \leftarrow s$;
                Enqueue($v, Q, m, n$);        /* add node $v$ into $Q$ */
        else if (node $u == d$)
              print minimum distance vector;
              print path from $s \rightarrow d$;
              break;        /* a shortest path from $s$ to $d$ is found */
        else
              for each node $v \in Adj[u]$
                  /* relaxation */
                  for the distance on each resource queue $j \in r[1..m]$
                      $A[j] = Q[v].dist[j]$;
                  for the distance on each resource queue $j \in r[1..m]$
                      $B[j] = Q[u].dist[j] + Q[v].wt[j]$;
                  if ($getmin(A, B, m) == 1$)        /* if $A > B$ return 1; else return -1 */
                     for the distance on each resource queue $j \in r[1..m]$
                         $Q[v].dist[j] = Q[u].dist[j] + Q[v].wt[j]$;
                     update $Q[v].max\_dist \leftarrow max\{Q[v].dist[j]\}$;
                     $Q[v].pred \leftarrow u$;
                     Enqueue($v, Q, m, n$);        /* add node $v$ into $Q$ */
end
```

REFERENCES

[1] A. Allan, D. Edenfeld, J. William H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 technology roadmap for semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42–53, January 2002.

[2] K. Chakrabarty, "Test scheduling for core-based systems using mixed-integer linear programming," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 19, no. 10, pp. 1163–1174, October 2000.

[3] R. Chou, K. Saluja, and V. Agrawal, "Scheduling tests for VLSI systems under power constraints," *IEEE Trans. on VLSI Systems*, vol. 5, no. 2, pp. 175–185, June 1997.

[4] Y. Huang, W. T. Cheng, C. C. Tsai, N. Mukherjee, O. Samman, Y. Zaidan, and S. M. Reddy, "Resource allocation and test scheduling for concurrent test of core-based SoC design," in *The 10th Asian Test Symposium*, October 2001.

[5] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, "On using rectangle packing for SOC wrapper/TAM co-optimization," in *Proc. IEEE VLSI Test Symposium*, 2002.

[6] E. Larsson and Z. Peng, "System-on-chip test parallelization under power constraints," in *Proc. of IEEE European Test Workshop*, May 2001.

[7] V. Muresan, X. Wang, V. Muresan, and M. Vladutin, "A comparison of classical scheduling approaches in power-constrained block-test scheduling," in *Proc. of ITC*, October 2000, pp. 882–891.

[8] M. Sugihara, H. Date, and H. Yasuura, "Analysis and minimization of test time in a combined BIST and external test approach," in *Design, Automation and Test in Europe Conf.*, March 2000, pp. 134 – 140.

[9] S. Koranne, "On test scheduling for core-based SoCs," in *Proc. IEEE Int'l Conf. on VLSI Design*, January 2002, pp. 505–510.

[10] Y. Zorian, "A distributed BIST control scheme for complex VLSI devices," in *Proc. IEEE VLSI Test Symposium*, April 1993, pp. 4–9.

[11] V. Iyengar, K. Chakrabarty, and E. J. Marinissen, "Test wrapper and test access mechanism co-optimization for system-on-a-chip," in *Proc. of ITC*, 2001, pp. 1023–1032.

[12] D. Zhao and S. Upadhyaya, "Adaptive test scheduling in SoCs by dynamic partitioning," in *IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, November 2002, pp. 334–342.

[13] D. Zhao, S. Upadhyaya, and M. Margala, "Minimizing concurrent test time in SoCs by balancing resource usage," in *Proc. of the 12th ACM Great Lakes Symposium on VLSI*, April 2002, pp. 77–82.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[15] R. Rajsuman, "Design and test of large embedded memories: An overview," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 16–27, May-June 2001.

[16] R. Conway, W. Maxwell, and L. Miller, *Theory of scheduling*. Addison-Wesley, 1967.

Dan Zhao received the B.S. degree in Biomedical Instrument from Zhejiang University, China in 1996 and the M.S. degree in Computer Science and Engineering from the State University of New York at Buffalo in 2001. She is currently a Ph.D. candidate in Computer Science and Engineering at the University at Buffalo. She is working in the area of embedded core-based System-on-Chip testing in the Electronic Test Design Automation Laboratory. Her research interests include VLSI design and testing, CAD, fault tolerance in real-time embedded system.

Shambhu J. Upadhyaya received his Ph.D. degree in Electrical and Computer Engineering from the University of Newcastle, Australia in 1987. He is currently an Associate Professor of Computer Science and Engineering at the University at Buffalo. He is a recipient of the 2000-01 IBM Faculty Partner Fellowship. He served as the Program Co-Chair of the 1995 IEEE Great Lakes Symposium on VLSI and the 2000 IEEE Symposium of Reliable Distributed Systems. His research interests are VLSI Testing, fault diagnosis, fault tolerant computing, information assurance techniques and diagnostic reasoning. He is an associate editor of IEEE Transactions on Computers and is a senior Member of IEEE.